

Apple

Jaguar Software

Design Specification  
Special Projects

May 1, 1990



Return Comments to:  
Russell Williams, DS Coordinator  
Phone: x4-7662  
AppleLink: WILLIAMS.R  
MS: 82D

Apple Registered / Restricted



# Table of Contents

This Table of Contents covers the entire Jaguar Software DS document. Note that each chapter's page numbers include an acronym of the chapter's contents. This will allow us to update the chapters without changing the numbering of all the others.

---

---

## **Jaguar Software Overview**

This chapter describes the key concepts and modules making up the Jaguar Software and describes how they relate to each other and to Pink.

---

---

## ***Jaguar Human Interface Guidelines***

This chapter describes the human interface elements that are common to system and application software: the menus, cursors, dialogs, etc.

---

---

## **Digital Assistant**

This chapter describes the fundamental Jaguar system software human interface and its underlying implementation — the equivalent of the Macintosh desktop, Finder, and folder/file structure.

---

---

## **Integrated Media**

This chapter describes the Jaguar's user and application tools for manipulating the various media types.

---

---

## **Communications**

This chapter describes the Jaguar's application tools for communication over networks and telephones to other Jaguars and non-Jaguar computers.

---

---

## **Low Level Software**

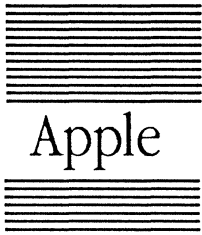
This chapter describes the software tied most directly to the Jaguar hardware: such as drivers and boot code.

---

---

## **Miscellaneous Software Issues**

This chapter contains miscellaneous information that doesn't fit anywhere else, unresolved issues, and software needs not yet completely understood.



Apple

Jaguar Software

## Jaguar Software Overview

Design Specification  
Special Projects

May 1, 1990

Return Comments to:  
Russell Williams, DS Coordinator  
Phone: x4-7662  
AppleLink: WILLIAMS.R  
MS: 82D

Apple CONFIDENTIAL

---

---

## Purpose of This Document

This document is a design overview of the Jaguar Software exclusive of Pink. It is not a comprehensive DS in the sense of containing complete class descriptions and user interface specifications. Many of these details will only be known after some prototyping has been done, and we wish to avoid putting details into this document that would almost certainly be changed. This document is meant instead to describe the functionality we expect to provide and the overall design we will use to provide it.

There is also a HyperCard 2.0 based version of this document. It interactively demonstrates some of the Jaguar software concepts and features, particularly in the Media area. The HyperCard stack may have more recent information than this document in some areas. Contact Galyn Susman to obtain a copy.

---

---

## Structure and Conventions

The DS is divided into four major sections: Digital Assistant, Media, and Communications, and Low Level Software. The first three sections reflect the three key areas of emphasis for Jaguar. The entire document is evolving from outline form toward a document which specifies the user and programmatic interfaces in detail. As it evolves, issues for future elaborations are described *in italic underline type*. Appendix A includes miscellaneous issues such as Wankel and Wilson management.

---

---

## Relationship To Pink

Jaguar Software is built on top of Pink. It differs from basic Pink in three main ways:

- It contains software for areas not addressed by Pink (e.g. Voice Recognition).
- It contains software needed to support hardware present only in Jaguars (e.g. the ISDN interface).
- It supports a different model of user interaction (e.g. the Digital Assistant vs. the Pink Finder).

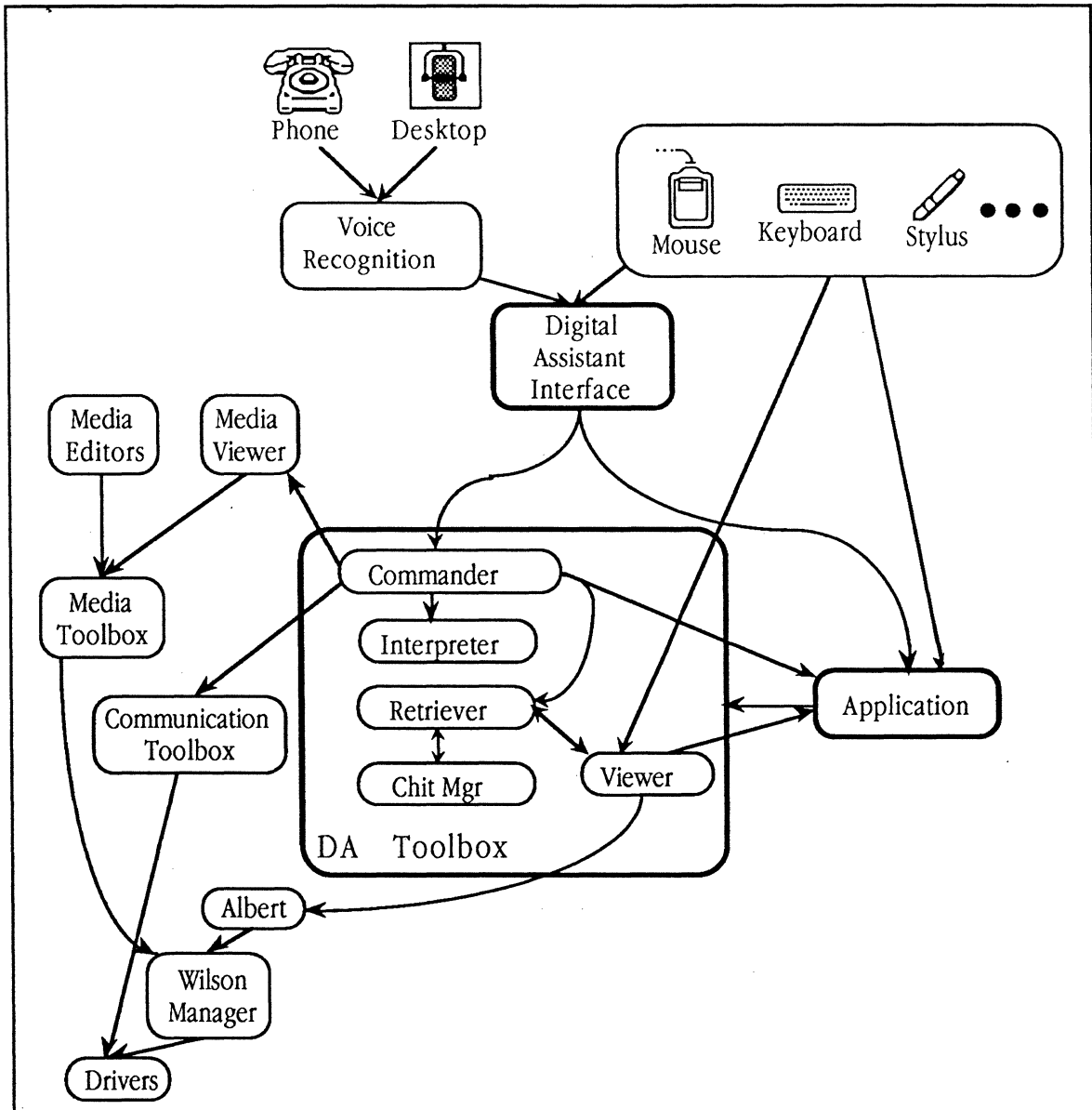
Thus Pink must be ported to the Jaguar hardware and Jaguar Software must be integrated with it. Some of the Jaguar Software, such as the Digital Assistant, can be developed fairly independently, while other portions, such as the Sound Server, are elaborations of software included in Pink and must be developed in close cooperation with the Pink team.

---

---

## Overview

The diagram below shows some relationships among some of the major components of Jaguar Software.



Key Jaguar Software Modules (not to scale)

**Jaguar Human Interface Guidelines**

The Human Interface Guidelines will be the evolving description of what the Jaguar looks like across all application and system software, though not every feature here will be found in every application. These guidelines, analagous to the Macintosh User Interface Guidelines in Inside Macintosh, describe how menus, windows, cursors, dialogs, scroll controls, and other common interface elements will look and behave. This behavior will be fully supported by the toolbox.

Some of these features have direct analogs in the Macintosh, but look or act differently in order to take advantage of Jaguar hardware or software. For example, graphic elements will be redesigned to take advantage of more powerful displays, and input standards will have to change if we use a stylus and/or multi-axis mouse. Some features have no direct counterparts in the Macintosh guidelines: some of these have also been to take advantage of new Jaguar functionality, while some represent an attempt to achieve consistency in areas where the Macintosh has none. For example, the Macintosh guidelines include no discussion of the behavior of 3-D or time-based data displays, sound input, or telephone access. While the functionality will evolve rapidly over the next few months, the final appearance of the various features will probably not be set for quite a while.

---

## Digital Assistant

The Digital Assistant replaces the desktop as the primary metaphor for interacting with the Jaguar. In the Blue and Pink worlds, the Finder is a tool with which the user explicitly organizes data and controls the machine. Computer users don't like the clerical work of organizing and filing information; the computer should be able to take on that task at a higher level. Instead of requiring the user to performing the clerical tasks of naming files, placing them in folders, rearranging and searching for them and so forth, the Digital Assistant allows the user to treat the computer more like a human assistant. The Digital Assistant is simply told to file information without specifying the details of how to do it. Just as with a human assistant the user asks for information by content and context, not by specifying which set of folders to search. The Digital Assistant can also take care of other aspects of controlling the user's machine, automatically handling common tasks such as performing regular backups or collecting AppleLinks first thing in the morning.

The Digital Assistant performs these functions by calling on several layers of subordinate software:

- The Interpreter translates user requests from English into the more formal language used by the Commander (English is not the only way to interact with the Digital Assistant).
- The Commander processes user requests and controls the appropriate software to handle the request.
- The Retriever searches for information by content, context, and attributes.
- The Chit Manager is a relational database providing the means for storing and retrieving information.
- The Viewer presents retrieved information to the user, both directly and by calling on various pieces of software which know how to deal with the content of different chits.
- The Pink File System provides the underlying data store needed to support the Chit Manager.
- The Communication Toolbox gives the Digital Assistant the ability to reach out over local and wide area networks as well as the global phone system.

## Chits

In a system characterized by greater connectivity and communication, a Digital Assistant should also keep track of assorted bits of information: phone calls, software updates, email addresses, and miscellaneous notes. In Blue or Pink such data are stored by separate applications which cannot easily share it: for example, information about a person may be stored separately in a phone list application, an email program, and in unstructured form on a note pad. Larger collections of information are stored separately as files.

The Digital Assistant unifies the concept of information storage through the concept of *chits*: arbitrary-sized pieces of (optionally) structured information which can be accessed throughout the system. For instance, all the information the user has about a person would be stored in a chit for that

person, and the user or any piece of software could easily retrieve information about that person (e.g. "Joe Smith's phone number", or "the email addresses of all people in department 5875"). Chits can be thought of as records in a dynamically extensible relational database of all the information in the Jaguar.

---

## **Media**

One of Jaguar's primary goals is to support a rich set of media types. Interest in and demand for "multimedia" is growing rapidly, and any new personal computer family introduced in the 1990s must be prepared to make a strong showing in this area or risk being born obsolete. Existing computers such as the Macintosh and PCs will have had media support added on. Jaguar's competitive opportunity lies in fully integrating the media support so that these new media types can be manipulated as easily and fluidly as bitmapped text and static graphics are on the Macintosh. These new media types include sound, text-to-speech, 2D and 3D animation, and video. Jaguar software provides comprehensive toolbox support for these media types in the form of the Sound Server, the Animation Toolbox (included in Pink), and the Video Toolbox.

Media can be input and output in a variety of ways. All media can be input, stored and transmitted in digital form using any of Jaguar's storage and communication facilities. Once input, the user can manipulate these media through the various media editors and the Media Sequence Editor and Media Sequence Player. Applications can manipulate the media through appropriate toolbox routines.

Each Media Editor is focussed on the job of editing a single medium, and the Media Sequence Editor ties them together, allowing the media to be synchronized in space and time. Media data are stored in a standard extensible format called Jaguar Data Format. This format allows media data to be easily interchanged among applications. Media data in this standard format can be displayed by the Media Sequence Player, either on user command through the Digital Assistant, or within an application using the Player's toolbox interface.

---

## **Communications**

Another of Jaguar's key goals is to increase the utility of the computer at a distance, allowing the user access to his or her data from remote locations, with or without a computer. Jaguar's communication software consists of a set of tools allowing the Digital Assistant and applications to provide this remote accessibility in addition to more traditional datacomm functions. These tools support ISDN, AppleTalk, fax, and other protocols as appropriate over Jaguar's built in ISDN, SCC, and analog phone connections.

To support communication over the analog phone network, various modems are implemented in software, including v.29 (fax). Addition of future modem protocols thus requires only software changes, not additional hardware.



---

## Low Level Software

Jaguar's low level software encompasses aspects of the system devoted to booting and maintaining the software and hardware configuration parameters, and the hardware-specific support code needed as an interface between the kernel and the hardware.

Code stored in the EEPROM or ROM is used to test the hardware and boot the system from SCSI disk, M/O, devices attached to the BLT, or possibly a network. It also handles reawakening the system from its low-power Standby Mode. The kernel debugger supports debugging of the Opus kernel, either locally or remotely, as well as the taking of crash dumps either locally or remotely. Drivers are required for all the motherboard I/O devices and for handling of optional BLT cards (the equivalent of the Macintosh slot manager). Finally, this category includes software for handling hardware errors and system shutdown and reboot.

Most of the low level software is invisible to the user; those facilities such as configuration management which are visible to the user are accessed through the Digital Assistant.

---

## Miscellaneous Software Issues

This section of the document includes software issues which don't fall cleanly into one of the other section and/or which are less well defined — sort of a kitchen sink or system heap. Wankel and Wilson management and real time scheduling fall here, as does 68K Emulation. Wilson management includes the necessary changes in Albert and the Layer Server to use Wilson's DMA, blitting, blending, and pixel munging services, as well as scheduling the limited bandwidth available through the various portions of Wilson.

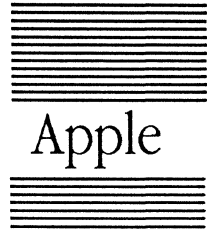
Much of the functionality of Jaguar requires real time response: producing frames of animation, compositing animation with video to avoid frame tears, synthesizing audio, producing modem data. As presently designed, the Opus kernel does not provide the requisite scheduling facilities, so Jaguar software must add it. The real time scheduling mechanism must handle dynamically changing real time tasks, it must degrade gracefully when there is insufficient cpu time to completely handle the computational load, and it must maintain the responsiveness of the user interface. Even though this functionality is being added by the Jaguar team, it is probable that software integral to Pink (such as the animation toolbox) will use it.

Jaguar will be able to run the huge base of Macintosh software by emulating the 68K instruction set and supporting the Blue Adapter (which is part of Pink).

---

## Summary

As a personal computer for the 1990s, Jaguar will live in a highly interconnected, media rich world in which people have little time or desire to deal with the technical details of computers. These factors interact in that the increased communication capabilities and media types will allow access to a vast quantity and variety of data, and more powerful tools are required to manage it. Jaguar's hardware provides the requisite resources. Jaguar's software must deliver that potential to the user.



Apple

Jaguar Software

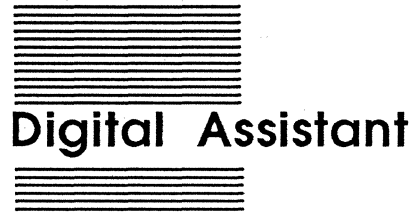
Digital Assistant

Design Specification  
Special Projects

May 1, 1990

Return Comments to:  
Phil Goldman  
MS: 82D

Apple CONFIDENTIAL



---

---

## Intro -

The Digital Assistant (DA) is the generic term for an entire subsection of system software that handles the data organization needs of the user and applications. It is composed of six components: the DA user interface, the Natural Language Interpreter, the Query Retriever, the Viewer, the Commander, and the Chit Manager.

The interface component is responsible for communicating with the user with the help of the Interpreter, and possibly the voice recognition and synthesis components. If the user wishes to retrieve data then the Retriever and Chit Manager provide it. The results will be displayed, "Finder style", by the Viewer. If the user wishes the computer to perform some actions the Commander determines what applications are involved, and delegates authority to them.

---

## Motivations

What is the Finder?

It is the graphical view of a database, a database in which the atomic objects are files.\* The operations that the database provides on these objects are few: find (via folder open), open, print, view, copy/move, rename, and delete. Further organization winside of the file is left to applications.

What is a file system?

The file system is the actual database engine for the Finder and applications. It allows the apps and the user (via the Finder and the Standard File dialog) to organize his data as file-sized units. Taken without the file system (and various other services) beneath it, the Finder is merely a presentation level application.

So, the file system is a database...but not a particularly good one. It manages only one type of object. It can do fairly few and fairly weak operations on that type. And most of all, the hierarchical structure is an extremely poor choice for data management, especially for a system of the size we are building.

The very first databases were know as *flat file* databases. That is, they had no inherent structure, other than the ability to sort records based on different attributes. These database worked very well on small amounts of data, but started to become more cumbersome as the amount of data grew — as anyone who has used the phone book realizes.

---

\* Files, as opposed to documents. The Finder blurs the distinction, and actually never uses the former as a term.

The popular solution to this was provided by the next great wave of database technology, *hierarchical* databases. They allowed the user to structure his data so that he could look at only pieces of it at one time. This allowed the user to deal more reasonably with larger amounts of data. Unfortunately, the tree structure does grow with the number of records, and thus traversal becomes a problem.

The next big wave, the *network* databases, were optimized for this traversal. Not only did they allow the user to categorize his data, they also let him define the links between the categories. The user could create a full graph instead of tree, and navigate through it. Unfortunately, the user still had to do exactly that — navigate. This forced a modal view of data, one set of it at a time.

The most recent wave of databases are the *relational* databases. These databases allowed the user to make complex queries that combined multiple sets of data in different ways. They alleviated the navigational problem because they allowed multiple views of the same data.

Apple's system software has followed this same relative history in the context of the Mac, about twenty years after the fact. The original Finder used a flat file structure, the old MFS (Macintosh File System). This worked well for small (400k) floppy disks, because the user was forced to physically juggle (literally) the data in 400k blocks anyway, so most of the data organization was actually taking place outside the machine.

With the advent of larger floppies and hard disks, the Mac was forced into a hierarchical file system, HFS. Although it initially caused compatibility problems, HFS allowed the user to solve much of his management needs, even on those "large" 20 Megabyte disks. The Finder visually supported HFS's tree structure by allowing folders to be placed within other folders. The key word here is "allow". HFS allows much, but provides little help.

Repeating database history, HyperCard sought to use a new method of data management for its smaller, more plentiful datatypes. The method of choice was, of course, the network database model, the model that has become synonymous with the "Hyper-" prefix. HyperCard could not have used the hierarchical model for two reasons. First, the average datatype is fairly small, the number of them is large. Also, the typical user is much less proficient at using computers than the typical heavy-duty file system user; he simply cannot manage any of the structring process.

Alas, history is not repeated, completely. Apple is asking its users to use a hierarchical database and a network database in order to manage their data. We can get away with this only because no system software vendor yet has provided a decent user interface on top of the current database technology of choice, the relational database. And to be fair, most Macs only contain a small to fair amount of data in the machine.

This changes with Jaguar. We have several advantages and needs that are not cogent to the Macintosh. They include:

- **Large Storage.** The typical Jaguar will have 1 or 2 orders of magnitude more storage than the typical Mac. The other software components, Media and Communications, will *easily* fill that space.
- **Small Data.** Although many of the data chunks, such as the Media-filled ones, will be large, the Chit Manager (see below) will make possible very small pieces of data, such as a telephone number. There will be a plethora of these.
- **External Data.** Every Jaguar will be connected to every other one, via the analog phone lines. Because of this degree of physical connectivity and the attendant system software support (e.g.

mail, data sharing) the percentage of data on a user's Jaguar that someone else created will be much higher than it would be on a Mac. If you wonder how this is relevant, take a look at how well your AppleLink messages are organized.

- **Natural Language.** This is essential for making the system accessible remotely via the phone. The problem with the hierarchical and network approaches is that they require navigation, so that over the phone the user would be "flying blind". With the query model the data comes to the user, rather than vice versa.

Of course, most of the motivation for this is that the relational model, or more specifically the query and view aspect of it, is an inherently nicer one for a file system. One of the big wins is that it does not force the user to be the database administrator too. He may view and modify the data, but he need not worry about the structure. The user will see this as an automation of the system; he no longer needs to put all of his MacWrite files in the MacWrite folder; the Digital Assistant will do it for him.

---

---

## Metaphor

The metaphor we have chosen is that of an assistant, an intelligent entity that performs, metaphorically, many of the same functions as the stereotypical secretary. It will file and retrieve your documents, based on just a little information about them. It will handle your personal schedule. It will take calls when you are away, forwarding them to you if appropriate. It will learn new actions if you wish.

This is in direct contrast to the desktop metaphor. What a terrible object to model! The Mac simulates a piece of wood with paper on top of it. The user is responsible for keeping his desktop neat, for throwing away old papers, and for constantly shuffling papers into the correct piles. As well, the desktop is unusable if the user is not sitting attentively in front of it. The most interesting feature of the desktop of the future is that it will have a Jaguar atop it.

---

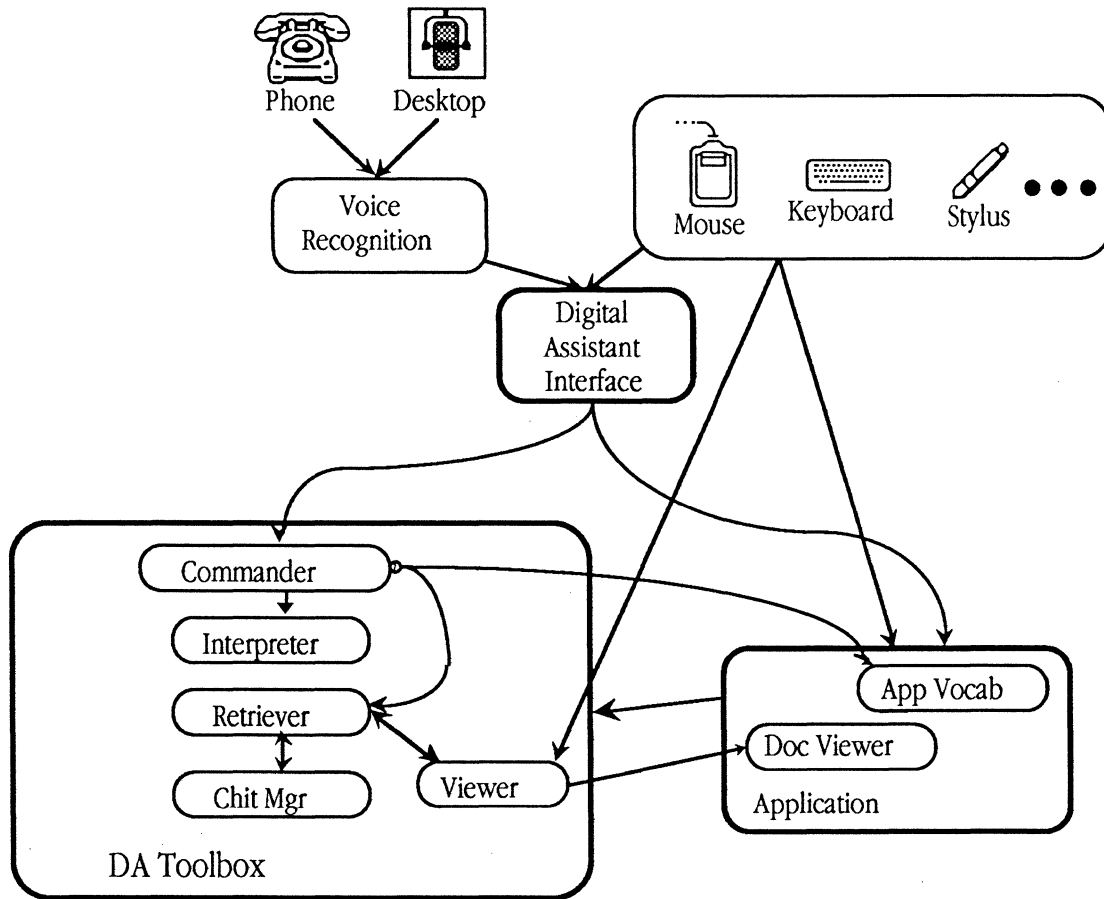
## Design Overview

The six components interact as shown below. They can be categorized as the DA Interface and the DA Toolbox. The distinction between the two is that while applications and users may use the toolbox, users are the sole clients of the toolbox<sup>1</sup>. The arrows in the picture represent components' dependencies upon other

---

<sup>1</sup>It is not quite this simple, because applications can in fact affect the DA User Interface, but not directly. Rather, the toolbox may request changes of the Assistant based on hints from the app.

components.



Digital Assistant Overview

Section 1

## DA User Interface

This section describes the Digital Assistant User Interface component. It also discusses various portions of general interface that are relevant to the relationship between the DA, the user, and applications.

---

---

## Metaphor

The DA User Interface will make very tangible the metaphor discussed above. It presents the user with the most appropriate simulation of a human agent, without unrealistically raising user expectations. From this point on we will also call this element "Bayles" in order to drive home the metaphor, and also to distinguish it from the DA system as a whole.\*

Implementing Bayles will be a tricky endeavor, as numerous studies have shown that expectations are closely tied to an agent's visual implementation. Therefore, we will adapt to the image to fit the abilities, based on our own user testing. For now, we assume that the most accurate image, using video, is most appropriate. Note that this whole issue is much more of a problem at the desktop than over the phone.

The interface component consists of two subcomponents, the desktop and the phone, which pertain to the method by which the user accesses the DA. Another large difference between the two is that over the phone the user *must* use the DA, whereas it is strictly an aide at the desktop.

---

---

## Desktop Interface

The Digital Assistant will be accessible over the phone and at the desktop. The former will be the sexy feature that draws attention to the Jaguar, but it is really the latter that is the bread and butter of the DA's user interface — 99% of the time the user will be using the machine at the desktop.

---

---

## Purposes

The Bayles will be portrayed as a human figure, using canned, compressed video. He will serve the following purposes:

**Help.** Centralizing help within the DA will improve usability; the user will know exactly where to go to find help. Conversely, it will improve the user's view of the DA, which is important if we believe in living up to (raised) expectations.

**Input.** As with help, we can make Bayles a convenient place for user input, no matter what the situation. Through the use of the DA Toolbox (primarily the Interpreter) Bayles will be able to feed the text to either the current app, or else the relevant one, depending on the text. If the user so desires, Bayles will listen for input, rather than require entered text.

**Feedback.** Depending on user preferences, Bayles may give feedback to user actions. This is different than notification (below), because it refers to the current app and is direct response to user action. It is clear that Bayles should sometimes respond to input to it, but it is not clear that it should respond to user input directly to an app, such as playing the role of the Mac Dialog Manager. An example of feedback is displaying the commands that are the translated user actions.

**Notification.** Bayles will handle notification in a manner similar to (but superior to) the Mac Notification Manager. Depending upon the urgency of the notification request and user preferences, Bayles may come to the front, display a message, read a message, or all of these.

---

\* Not my idea, really.

Notification is an example where apps interact with Bayles itself. We mentioned above that this interaction is indirect. The apps have no notion that Bayles exists, it is just the means by which the system expresses the notification.

Note that for all of these purposes Bayles is not the only option. For example, applications themselves may still provide access to help, since there may be cases where it is more efficient and intuitive to do so (e.g. an object that explains itself). Another example is notification. For very low priority notification an app may wish to simply highlight an important object. This occurs for actions that straddle the line between notification and status reporting (e.g. connecting to AppleLink).

---

## Look and Feel

*This section is completely up in the air. The following description is given just for the sake of argument.* Bayles exists in its own set of three objects\* in its own layer. One window will contain a human bust. The other two windows are for user input and output. The input object is shaped like a cartoon thought bubble. The output object is shaped like a cartoon speech bubble. None of these objects contain any other visual structure (e.g. title bar, scroll bars). The user may drag the objects by clicking in any of the three and dragging; they move as a group.

Bayles may create other objects in its layer, as appropriate. For instance, it may open up a preferences window. Typically though Bayles indirectly causes windows in other layers to open, in response to user actions.

Bayles has the ability to climb out of its window and move around the screen under its own volition. This feature is used primarily with the help system, so that Bayles may “point” while explaining. *Will this work for 3D pointing, or does the DA need to enter windows?* While Bayles moves about, an inactive (*visually illustrated how?*) version of it remains where the user last dragged it. The movement is cancelled when the user switches apps or activates (e.g. clicks on) the inactive objects in Bayles’s layer.

If user testing shows that users need to bring Bayles to the front often, we will look at adding an “Assistant ⌘A” menu item. We could also always keep the layer above all others, but this causes problems with determining keyboard targets.

---

## Aside: Limits to Desktop VR

We plan to use voice recognition (VR) sparingly at the desktop. Although the full power of the phone interface will be available, it will be turned off as a default. In addition, when the feature is enabled we will warn the user about its appropriateness. There are three reasons for we do so:

1. **Raised Expectations.** VR will be a powerful tool, but it is not yet at the point where it is sufficiently reliable. It is fine for the phone, where the alternatives are worse and the need is great, but it cannot (currently) compare with direct manipulation at the desktop. Therefore, if users find

---

\* Object here means visual object, ordered front to back with others within a layer. A window is an example of such an object.



the VR cumbersome and unreliable at the desktop they may grow annoyed, and even stop using the Assistant over the phone.

**2. Audio Clutter.** We expect Jaguar to be used most commonly in the workplace. A large group of people constantly talking to their computers could cause a large amount of noise. Not to mention that the ideal response from the computer is through voice output, which its own microphone would pick up. If we cannot find a way to filter this then the computer would be unable to speak and listen at the same time; it could not be interrupted. Not quite the ideal model for an assistant.

The obvious solution is a headset, which will only be successful with a great deal of creative product design. Otherwise, it would be awkward and uncomfortable.

**3. Lack of Application Support.** While we can expect applications to allow easy access to documents via VR, so they can be used over the phone, we do not expect them to customize every single feature they provide to be used in this way. This would cause too many headaches for third party developers and for users, who would be forced to memorize the verbal nuances of every app. Instead, developers should concentrate on the few features that make voice most usable.

Despite all of the above, we do believe that VR has a valuable limited role at the desktop. It is extremely useful as a means of input when the user has focused his eyes and/or hands on his work, and does not want to be interrupted. An example of this is a user drawing with a black paintbrush in JagPaint. He is concentrating on his HeloCar, but wishes to change to the paint bucket tool. He simply tells the computer, "paint bucket!", and the tool changes. At no time has he had to stop drawing.

Of course, the user will have unlimited voice access at the desktop, as an option. At some point during the evolution of the Jaguar this will be useful for a large portion of our users. Immediately it will be useful for two small segments: the curious and the visually impaired.

---

## DA User Interface API

### Notification

```
enum{kVeryLowPriority, kLowPriority, kNormalPriority, kHighPriority,
kVeryHighPriority} NotificationPriority;

typedef void (TNotificationResponseFnc)(Boolean isOkay);

class TNotificationRequest
{
public:
    TNotificationRequest(const NotificationPriority, const char *text,
        const TSequence *pSequence = nil,
        const TNotificationResponseFnc
            *pNotificationResponseFnc = nil,
        );
    ~TNotificationRequest();
};
```

---

---

## Phone Interface

The addition of an analog phone jack in the Jaguar makes a tremendous impact on the nature of the machine. In the context of the Digital Assistant, it provides a way for the user to employ his Jaguar almost anywhere in the world.

---

### Goals

There are three major goals we have for the phone interface:

1. **Retrieval.** The user should be able to retrieve any textual piece of information that he could see at the desktop. At a minimum this includes that info on the computer's disks.
2. **Entry.** Likewise, the user should be able to add to the information within the computer. This will not be as common as information retrieval.
3. **Control.** Finally, the user must still maintain control over his machine. He must be able to give it commands and get feedback from the results of these commands. The specific goal here is to handle commands such as "Fax me the '89 revenues at 415-348-1966".

---

### Preserving the Metaphor

One of the big advantages to providing a command-based interface at the desktop is that that very same metaphor can be employed over the phone. The syntax of the interface language, direct manipulation and text entry vs. spoken commands and feedback, changes but the semantics of the interface remain identical. The user comfortable at the desktop will immediately be able to converse Bayles over the phone.

---

### Efficient Blind Interaction

The obvious interface constraint of remote access is that the Assistant can provide no visual cues. Not only will the user be directing Bayles solely by voice, but that will also be the principal method of response, along with other appropriate audio clues.

Because of this constraint the amount of interaction is necessarily higher. Also, the amount of control the user needs over retrieval is higher. The user can instantly see twenty files in a view, but it would take Bayles perhaps a minute to recite their names.

The solution is to provide a standard set of access commands that apply to Bayles's recitations, as well as the reading of document sections. The access commands allow for an approximation of random access on this slow sequential data. They also provide for the execution and monitoring of time-consuming commands.

There are several issues that Bayles must deal with over the phone:

- **Large Data.** What if a list of documents contains hundreds of them? What if the user is looking for information in a 40 page document?
- **Multidimensional Data.** Not all data follows sequential progression. The most obvious example is a spreadsheet. The phone interface does not support any multidimensional access, but apps can add on to the standard access commands in the same way it can add on to Standard File on the Mac. The

only requirement is that it support the standard commands, and thus implicitly a method of sequential access, as well.

- **Atextual Data.** Obviously, not all data is text. The onus is on apps that create other data types to either translate them into text or else skip them.
- **Asynchrony.** The user may wish to execute a time-consuming query. The DA cannot "go away" while the command is running. Instead, the user will be able to issue more commands to Bayles. This means that there must be some way for the user to identify, query, and switch among command contexts.

This is analogous to the problem that the Unix C Shell solves with its interface for job control. The user must explicitly force a job to run in the background when he starts it, although he can switch it between foreground and background whenever he wishes. If it is in the background then it waits when it needs information from the user. Jobs are identified by job number, numbered from 1. The user can query as to the state of the outstanding jobs, receiving for each a report as to its state and an optional description why it is in that state.

This is a good general model for our interface as well, with a few exceptions. Asynchrony is undoubtedly a power user option, so the level of complexity the Unix model has is not excessive.

We do have different constraints than a shell session though. For example, whereas the C Shell will not let the user logout while jobs are active we have no such luxury with the phone. Therefore, Bayles must use the Commander to send a message to the apps handling the pending commands when the user hangs up the phone. This message has the audio cue flag turned off (see Commander below). Also, users will have trouble remembering job numbers without a visual explanation of each number. So, the power user can identify commands by content (e.g. "Stop the search"). This disambiguation is unimportant for the novice, who makes at most one command at a time.

We also have different features of which we can take advantage. For instance, we could have audio hints that identify an active command, such as different voice characteristics for Bayles. Perhaps its voice gets higher corresponding to the higher priority jobs, or perhaps it gets (a bit) slower for the less important jobs, to reflect how fast they go. Perhaps we superimpose the name of the job over the output.

There is also a question of whether or not we wish to make the foreground/background schism explicit. Our plan right now is to hide this, and merely let the user interrupt the DA whenever he wishes. An interrupted active command runs in the background until the user holds it or "enoughs" it (see below).

Another question is, is this scheme better than having multiple assistants that each operate synchronously? The functionality of each scheme is identical; it is only the metaphor that changes. One problem with assistants is the notion of identity of each, especially over the telephone. At least the jobs are named by what they do, whereas the names of the assistants would have to be arbitrary. Perhaps on the desktop they are more useful than just for jobs?

No question that if we run out of time this whole portion of the DA (command control) deserves to be pushed off to a future release, but the basic standard access system cannot be.

The DA will understand the following standard access commands:

“Stop (<x>)” — Stop telling the user the info, or executing the command. The command name (e.g. “the search”) is specified optionally.

“Continue (<x>)” — Stop telling the user the info, or executing the command. The command (e.g. “the search”) is specified optionally.

“Read It” — Start telling the user the info, from the last place stopped.

“Faster (by <x>)” — Tell him faster. The number after “by” is how much faster.

“Slower (by <x>)” — Tell him slower. The number after “by” is how much slower.

“Start Over” — Start reading data from the beginning.

“Skip Ahead (<units> <unit type>)” — Move the reading position ahead by one unit, or more if specified. The units are defined by the application and the context, if not specified. For example, the units in the DA will typically be chits and documents. In a spreadsheet they would be cells. In a word processor they would be pages, although the user could specify chapters, paragraphs, lines, words, or characters.

“Go Back (<units> <unit type>)” — Move the access position back by one unit, or more if specified.

“How many (<unit type>)?” — How many units are there in whatever the user is accessing?

“What Units?” — Tell the user what units exist in the current access context.

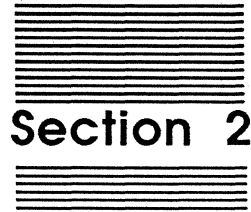
“End (<x>)” — The user is done listening or waiting. The command name is optionally specified.

“Status” — What commands are currently executing? The DA will respond with a message to the user giving the name of each command, its status, and a brief description of that status. Like all other messages, standard access may be performed upon this too.

“Quiet (<x>)” — Don’t tell the user everything, just the important stuff. An optional command name may be specified.

“Noisy (<x>)” — Tell the user as much as possible. This includes giving status on every conceivable action, and updating pending statuses more often. It also causes the DA to echo back the command the user just gave, to prove that it understood. For example, when the user says “Read It”, the DA prefaces the reading with “Playing from <foo>...” This is set as default. An optional command name may be specified.

What else?



## Section 2

# Interpreter

---

---

## Intro- What do it do?

The Interpreter translates the natural language command into an intermediate command language that the Commander can then use to delegate authority to the appropriate entities. It is also used by applications to parse commands given them.

Because of these different requirements, the Interpreter machinery will be separate from the grammar it parses, so that it can handle many grammars simultaneously. This separation is also important in order to allow for localization.

When we discuss the Interpreter below, we assume the grammar that DA uses.

---

---

## Interface Model

---

### Vocabulary

*The vocabulary of the Interpreter is highly dependent on the recognition rate and extensibility of the voice recognition system. So once again the description below is conditional. The key question is whether we have a rich vocabulary or a sparse one.* The Interpreter has a large basic vocabulary. In addition, users can customize it to suit their needs. Most of this customization will be extension, so that one user's Assistant is still useable by another.

Applications determine the vocabulary of the Interpreter to a great extent. Each application registers the verbs that it can handle with the Commander. The Interpreter has access to this list; it need only notice that the word is a verb since the Commander is ultimately responsible for delegating the sentence based on the type of verb.

---

### Syntax

Ideally, the Interpreter would be based on a semantic grammar, a grammar in which the syntactic categories are based upon the semantic concepts they describe, such as chits, documents, people, phone calls, etc., as opposed to theoretical categories like "noun phrase". The advantage to such a grammar is a higher "hit rate" since it is intrinsically tied into the domain. The typical reason that a semantic grammar is not used is because of a concern about portability of the grammar to other domains, and this is not our concern.

One problem does present itself with any interpretation of a semantic grammar. Although portability is not a concern of ours, extensibility is. How can we hardwire the grammar if the objects beneath it are not fixed? For example, the fax manager would want to define a certain sentence structure based around the verb "fax". It would want a construct such as

S: "Fax" recipient-object data-object "at" fax-destination

But this construct must be added on the fly. *This* is the problem.

There are two ways in which it can be solved. First, the parser can allow additions to the grammar on the fly. Or, it can define broad classes of subcategories, most importantly for the verbs, and force each verb handler to fit into one of the classes. So for example the grammar would contain a construct of the form

S: Send-verb recipient-object data-object "at" destination

and then the Fax manager would merely notify the Interpreter, or rather the Commander. Note though that it is optimal for the app not to know anything about the word "at", for transportability. On the other hand, the app will have to be localized anyway, so it could conceivably include the construct. On the third hand, if we do allow the app to include the word "at" then it will require immense cooperation from all apps in order to switch languages on the fly,

Therefore, we will attempt to keep all natural language knowledge within the Interpreter, but we will allow apps to extend that knowledge if absolutely necessary. Even in the rare cases where this is done the app should try to use one of the builtin grammar rules. This will allow the app to work better in certain languages, but work correctly in all.

By the far the most important aspect of the Interpreter, other than that it work, is that it be easily transportable to other languages. The goal is to allow someone with little or no programming skill to be able to make most conversions. Therefore, we will separate the grammar rules from the Interpreter code (via chits — see below) and have the Interpreter understand how to manage multiple grammars at once. This latter feature should actually make it possible to have a multilingual Interpreter (which is probably a redundant term).

Another key to the Interpreter is that it makes clear to the user what types of sentences it understands. This will be accomplished via the Grammar Book (which will be part of Bayles's help system), which contains example sentences that illustrate the different types. If possible, the grammar book will be used to transport the grammar to other languages, and customize it within one.

---

## Semantics

The semantics of the Interpreter will be dictated completely by the basic environment that the Assistant provides, along with the extensible one provided by applications via the Commander.

---

## User Access to Retriever

The user has access to the Retriever through the Interpreter. Therefore, the user never need worry about the intermediate form of the language, only the natural one. This allows us one of two options. First, we could hide the intermediate form so that later releases of the software can mold it to support additional constructs or are more efficient. On the other hand, the intermediate form could provide a way to have a more expressive command language for power users, since it could allow for constructs that are indescribable in our English subset. Most likely we will just make the natural language powerful enough to do this; we get a large gain by keeping the intermediate language in a parsed form.

---

## Localization Issues

As mentioned above, localization is of the utmost concern in this area. The Interpreter will be created with this expressly in mind. The strategy will be to parametrize and "chitify" (see **Chit Manager** below) the Interpreter in each of its subcomponents. This should allow languages closely related to English (e.g. romance languages) to be altered sufficiently. For languages less similar the alternative will be to replace the parser. This will be facilitated by the development of a set of extensible tools we provide. This level of conversion will require programming skills, but should still take little enough time that other portions of the localization dominate it.

---

## Classes

```
enum {kAll, kEnglish, kFrench,...} TLanguageType;

class TGrammarSet : public TSet
{
public:
    virtual void          Add(TGrammarRule);
    virtual void          Add(TGrammarSet);
    virtual void          Delete(TGrammarRule&);
    virtual void          Delete(TGrammarSet&);
    virtual void          Use(TLanguageType);
    virtual void          Disuse(TLanguageType);
    virtual Boolean       Interpret(const TText&, TParse&,
                                  TLanguageType = kAll);
    virtual Boolean       Contains(TLanguageType);
    virtual TIterator&    GetLanguageIterator();
};

class TGrammarRule
{
public:
    TGrammarRule(const TText& pattern,
                 const TText& rewrite, );
    virtual void        ~TGrammarRule();
};
```



## Section 3

## Commander

---

### Intro-Why needed

The Commander is the traffic cop of the Digital Assistant. It is responsible for delegating the natural language command to the different extremities of the software. It is furthermore responsible for handling the feedback from the responsible software, thus determining the outcome of the operation.

In addition, the Commander must access the Assistant's memory, the Chit Manager, in order to provide the data to fuel these operations. This is nowhere more evident than when the operation itself is a request to view data.

---

## Command Path

When the user types or speaks a sentence, typically to Bayles, the frontmost application has the first shot at interpreting it. It may do this with the help of the Interpreter. If the application determines that it cannot do a local parse of the text then it sends it to the Commander, which itself uses the Interpreter to parse it. If this fails as well then the Commander returns the suitable error message to the app.

---

## Hand Verbs to Apps

The subset of English that the Interpreter provides is command-based. That is, the verb in the sentence is the most important word. The Commander architecture reflects this; it is the verb from which the Commander decides to which applications it should hand the command.

Each application registers the verbs it handles with the Commander. More than this, it registers a template for the entire sentence that is based on a particular verb. This template includes the types of objects it will accept, defined in terms of chits. For example, the Fax Manager would register the following template:

Fax: <document>,<location>,<person>

Notice that the template is language independent; it is the Interpreter's job to fill in the slots. Also, one command may map onto many templates, such as if the users says "Fax me all documents about Erich at 415-348-1966" or "Fax it to the Jaguar Interest Group." However, it does not appear as though one command can generate templates for different apps.

Also, notice that multiple applications can register for the same verb. In that case, the rest of the templates are compared for a match. If many match, the most specific one will be used. The only criteria for specificity currently is in comparing chit types. In case of a complete tie the Commander will ask the user once, by notification via Bayles, and then remember.

As an example, the Retriever registers the following template:

Show: <chit>

And a mail application may register this template:

Show: <message chit>

The mail application would be invoked when the user requests "Show me all mail".

When the user makes a command, the Commander sends a message to the responsible app with the following information: filled template, visual cue flag, audio cue flag. *Should we include the unparsed sentence too?* The cueing lets the app know whether to respond visually, audibly, or both. The app replies with a status, which may include *pending*. If the status is *error*, then the DA will report the error back to the user, and the app may elaborate if the visual cue flag is set (and elaboration is definitely needed). If the status is *pending* then the app may send a message later with the eventual status.



---

## Get Nouns from Retriever

The Commander uses the Retriever as its noun handler. Chits and documents are the basic units of operation. Thus they are typically the objects acted upon.

The Commander passes through the noun phrases representing these objects to the Retriever, which then returns a set of objects. The Commander does this before verb handling (see above), so that it may send the verb handling apps the actual objects, rather than a description of them (e.g. the actual set of documents about Erich). Should we allow other apps to be noun handlers?

---

## Command Language

Although the user's command language will be English, it will be the Commander that is ultimately responsible for executing the command. These commands may be typed in by the user, sent from a command chit that the user double-clicked on, spoken by the user, or even sent from an application.

In addition, commands may be triggered by a preset time trigger, a user-defined condition, or invoked from another command. The Commander is responsible, once again, for finding the appropriate commands, executing them, and then returning to the previous context.

The Commander will serve four basic purposes:

- Allow a user to control his Jaguar intuitively
- Help a user to customize his working environment
- Allow for limited vertical programming
- Automate the user interface

The Commander will be similar in spirit to HyperTalk, in the functions it serves within the Jaguar user interface. It will also be similar to HyperTalk in that commands written using it will be distributed amongst the objects upon which it operates. We see it as "HyperTalk for the system".

For example, say the user attaches a script to a disk icon. Then, he can create a set of actions that will be performed when an appropriate disk event (e.g. eject) takes place. In addition, the script can access and modify other objects, perhaps another disk.

One of the problems with HyperTalk is one of its very strengths: the ability to distribute a program over a variety of objects. Besides the obvious, it causes us headaches in our attempts to support multiple users (not necessarily simultaneously!) on the same Jaguar. We wish to allow each user to have his own environment, including his set of Commander scripts, which can be brought to a different Jaguar if necessary. Perhaps the Interpreter can simply support a view of the user's environment from which he can copy from a floppy or over the network.

The very existence of the Commander implies a commitment for each application. In order for the Commander to support a rich and integrated set of actions, applications must cooperate by

registering the verbs they understand, and then accepting the Commander's message to handle the command.

---

## Scripts

The Commander will allow the user to create scripts of commands, and to execute these scripts either manually or automatically. Although the Commander plays a major role in user scripting, it is by no means the only portion of the DA that is needed to support scripting.

### Creation/Editing

Scripts will be chits, so that they can be created and modified using the Chit Application. In addition, they will show up as visual objects within the Viewer, and like all other chits they may be queried for by the user. When double-clicked upon, they will execute. A new script will automatically be named with the name of the object that refers to it, with a "script" appended.

### Attachment to Objects

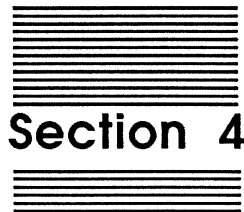
Command scripts may be attached to any object — not just in the Viewer but in any application. The user can do this by dragging the script chit icon on top of another icon. Note that this does not put the script inside the object, it merely makes the object's script field point to that script. Therefore, multiple objects can share one script.

---

---

## Localization Issues

There should be no localization issues with the Commander, since it is the Interpreter's responsibility to translate the user's request into the Commander's language.



## Section 4

### Retriever

---

---

#### Intro

One of our greatest concerns is that the user be able to view his data in any way he sees fit, with little or no administrative overhead heaped upon him. The Retriever is at the heart of this approach. It provides for context- and content-based retrieval on documents, and to chits in general.

The Retriever is a portion of the Digital Assistant never seen by the user. It is an internal engine that maps a standardized query to a set of chits that satisfy the query criteria. It is used by the Digital Assistant (via the Interpreter), although it will be accessible to the rest of system software, and to applications. Applications use the Retriever as their lowest view of the file system (or more accurately the chit system).

The content-based retrieval will provide full indexing on all text data, so that users can query for any documents and chits containing certain strings.

---

## DBMS

The Retriever is involved in database management to the extent that is used as the textual/spoken interface to the Chit Manager. It adds value to the Chit Manager in that it is fully able to select chits based on some criteria. Also, it is active, a server, whereas the Chit Manager is a passive system, a library. This activeness is necessary in order to support dynamic views on the Chit Manager's data.

---

## Content/Context-based Retrieval

The Retriever has the capability to retrieve chits based on any attribute that the type chit for it describes. This allows it to support constructs such as date, size, etc.

It can also retrieve chits based on information kept in their text fields. Each Retriever (see Multiple Retrievers below) will define its own implementations of CBR (Content-based Retrieval) for the disks it manages. This implementation will obviously be transparent to the Commander; all it cares about is receiving the set of matching chits. Typically, though, a small (40-100 MByte) hard disk would contain a list of digital signatures for its text, and a floppy disk might not contain any index.

The typical Retriever for local hard disks will use digital signatures for text indexing. A large disk (> 100 MBytes, perhaps optical or CD) would contain a structure for more efficient retrieval, such as the PLS System. The typical Retriever for floppies will do no text indexing. Of course, the size of the disk is not so important as is the amount of text and the likelihood of it being used for CBR. This is the very reason why the Retriever may make the decision.

An important question is, how does the Retriever know what type of scheme is being used. Well, it will continue to use the one on the disk until it is reformatted. At that point, it will use a heuristic based on the disk size (*and also the media access speed?*).

When an index needs to be updated, the Retriever will do so in the background. If a request is pending completion of the indexing it will be raised to a higher priority than the retrieval.

---

## Multiple Retrievers

Typically, the Viewer will maintain a connection with one Retriever for maintaining all of the views — the Retriever on the local Jag. However, for remote machines that support it (at least all Jags) the Commander can assign multiple Retrievers to each view. With the correct underlying system software (e.g. location transparent ipc) this should be trivial.

The difficult parts are in allowing the user to pick Retrievers, and in making clear the cost of the connection to the remote ones, in time and money. The obvious solution is to allow the user to say "Show me all Retrievers". This may need to be organized hierarchically (gasp!) if the number is large. This whole topic is currently under examination by the user interface group.

The Viewer/Retriever connection is set up by the Commander. Initially, the connection is between the Commander and the Retriever — "...all files about Erich". Then the Commander contacts the Viewer — "Show me...". The Commander also sends the Viewer enough information to maintain an ongoing connection with the Retriever. This is necessary in order to ensure that the view is dynamic.

---

## Offline Storage

The Retriever will have some knowledge of offline volumes.

---



---

## Classes

The call set the Retriever supports is:

Chit\*            **GetRetriever**(Chit \*disk);

Returns a chit that represents the Retriever for the given disk. The chit has pertinent information about the Retriever, including which disks it handles, and whether or not the Retriever is executing on the local computer.

void            **AddDocument**(Chit \*Retriever, Chit \*document);

Given a Document chit, the Retriever adds it to its CBR (Content-based Retrieval) index.

Set of Chit\*    **Retrieve**(Chit \*commandChit);

The commandChit is a chit of type Command, with target == the chit for the Retriever. It also refers to two set of chits. The first is a set of chits, that represent the chits that must be matched. This set is ordered as a tree with and and or nodes, in order to describe the boolean expression to be evaluated for each chit. The second set of chits parallels the first, and explains how the individual fields are to be matched (e.g. ==, !=, >, etc.) The chits returned can be of any type.

Chit\*            **TimeRetrieve**(Chit \*Retriever, Chit \*commandChit);

Returns a time chit that represents a good guess as to how long the retrieval should take, based on the type of query, how remote the disk is, etc.

---



---

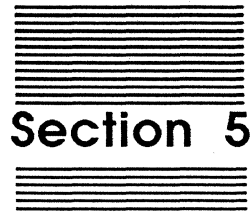
## Query Language

A constraint chit has a chit reference to any other chit, which is called its value chit. For every field in the chit it points to, the constraint chit has a parallel field that describes the constraint on the other field.

When we mentioned the and/or tree constructed above, it is actually constructed over the constraint chits, so that the value chits used in constructing the query may be actual preexisting chits (so that queries like "Show me all documents older than this one" are efficient).

The fieldtype of each fields in the constraint chit is dependent on the corresponding fieldtype in the value chit. Below is a table for some of the allowable realtions:

<u>Value Chit Field Type</u>	<u>Constraint Chit Field Values</u>
integer	T, F, ==, !=, >, ≥, <, ≤, divides, relatively prime
string	T, F, ==, !=, >, ≥, <, ≤, contains, is word
chit reference	==, !=, is same type
set of chit	==, !=, subset, superset



## Section 5

# Viewer

---

## Intro- Finder lives

*The Viewer is currently undergoing complete prototype. This section will be revamped during this process to reflect that state of the prototype.*

Once the Retriever has returned the set of chits that match the query, the Viewer is used to display the results. The Viewer is responsible for display, and manipulating the chits in a Finder-like, iconic manner. Our plan is to have an icon for each chit, inside of a view window that corresponds to the query. The windows can be moved and resized by the user, and the icons can be moved as well, inside of their respective views. Other direct manipulations, such as double clicking, dragging to a disk icon, and dragging over an open document, will cause either intuitive actions, or none at all if we cannot find one that fits our paradigm. In other words, the Viewer provides the Finder's direct manipulation user interface.

However, the similarities to the Finder end at this point. The windows are each only views onto the chits, rather than the chits themselves. In the parlance of the current Finder, every icon is an alias in that it can be seen in more than one window. Unlike the current aliases, however, the objects in the view do actually represent the object. For example, if the object is thrown in the trash it actually will be deleted. The visual distinctions between Viewer views and Finder windows will be large enough so that there will be little confusion, even between users that switch between Macs and Jaguars.

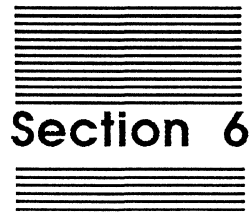
---

## Effective use of Direct Manipulation

One of our biggest challenges is to provide a large amount of direct manipulation for an interface method, natural language, that is inherently non-visual. The primary way in which the Viewer will accomplish this is via command chits. These will manifest themselves as iconic objects that can be double-clicked to create the appropriate command, be it a query or otherwise.

When combined with the schedule view, the commands can be given a temporal element. This combination is done via direct mapulation; the user merely drags the command chit to the time he wants it to happen.

Of course, all of this assumes that the command chits are accessible. They will be — as a default the Assistant will provide a "Show common commands" view. Our hope is that 95% of the user's actions can be captured by the command chits in this view. For those other 5% the full power of natural language is needed anyway, to handle the user's sudden inspiration (e.g. "Show me all letters from Jean-Louis about earrings").



## Section 6

# Chit Manager

---

## Intro-

The “Chit Manager” is a free-form database system which handles the semantic objects which define the user’s experience with the system. It essentially serves as the lower half of the Digital Assistant: maintaining the user’s data, and providing to the user access to the underlying files and hardware.

---

## Motivation

The reason for providing the chit manager (as opposed to using existing tools) is the need to manage a large number of small, highly interrelated, flexibly formatted data objects. The small size of the typical datum makes using the filesystem inefficient due to the typical overhead implied by “a file”. The high degree of interrelationship makes the Resource Manager and/or ADF inefficient. And the flexible structure (as well as the need to manager some large data objects) make traditional database systems inefficient.

---

## Goals

There are three major functional goals of the Chit Manager.

### Support Digital Assistant

The first and foremost goal of the Chit Manager is to support all of the database management needs of the Digital Assistant. This implies the ability to store diverse kinds of information in a very flexible manner, and the ability to access that information quickly. There are three broad classes of information which the Chit Manager needs to maintain for the Digital Assistant.

#### Access to Machine Information

All of the information on the user’s machine and about the user’s machine needs to be accessible through the chit manager. The Digital Assistant uses the Chit Manager as the user’s primary access channel to the “filesystem” and to any peripherals connected to the machine. This implies that the chit manager needs to provide rapid access to large volumes of data, and that at the lowest level it needs to provide some form of random access metaphor for large data structures/files.

#### Access to Global Information

All of the information about the machine’s (and user’s) environment needs to be able to be accessed through the chit manager. This includes Phone/Address lists, networking information, and the like. This implies that the chit manager needs to efficiently support highly structured data.

### Access to Personal Information

All of the user specific information which the user wishes to be accessible through the Digital Assistant needs to be able to be maintained in the Chit Manager. In this sense, the chit manager is also a Personal Information Manager. This implies that the structure of the database needs to be very flexible and needs to handle the irregular nature of "Personal Information".

### **Provide Shared Database for Applications**

The secondary goal of the Chit Manager is to provide a mechanism for Applications to share common information such as user preferences, address books, style information and the like. This implies that a "program friendly" interface needs to be provided which is general, and not tied specifically to the needs of the Digital Assistant.

### **Provide a Useful Tool**

The final goal of the Chit Manager is to provide a useful and interesting metaphor for information management which third party programs can use for their own purposes. Since it is impossible to guess in advance how the Chit Manager might be used by other developers, this implies that we need to provide as general a tool as possible, with as many aspects of the system being extensible as possible.

---

## **High Level Model**

This section describes the high level "user" model for the Chit Manager. One should however keep in mind that the "user" in this case is more often than not a program (like the Digital Assistant itself) and not a human being.

### **Overview**

The user's personal dataspace maintained by the Chit Manager is composed of persistent, sharable data objects called chits. All of the information in the dataspace is stored in chits, including all of the information which describes the chits themselves.

Chits are in turn broken down into fields. Each field has at least a datatype and a value. Fields may also have a label and/or a fieldtype to aid a user in interpreting the data. The datatype of a field determines how the value is to be interpreted. The Chit Manager will provide a robust set of standard datatypes including chit reference, rich text, video, and sound as well as a mechanism for third parties to add their own datatypes.

The value of a field can either be fixed size or variable size, as defined by its datatype. For program "users", the values for all fixed size fields, along with descriptive information for all the fields in a chit, are stored in a contiguous block. The values for variable size fields are stored separately and require some additional code to be accessed either directly or in a random access manner. Conceptually, if not literally, variable length values can be thought of as "files".

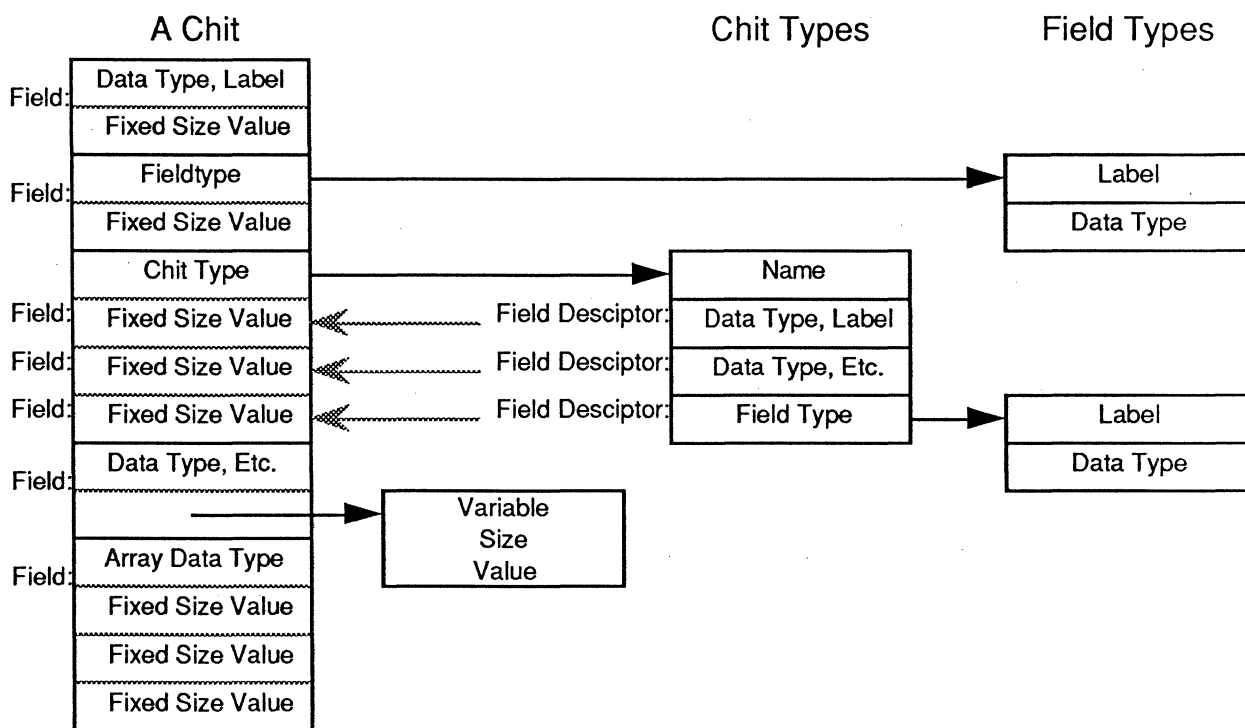
A field may also have a fieldtype which relates that field to equivalent fields in other chits. For instance there may be many different chits which contain fields with a person's name in it. To insure that the Chit manager understands that all of these fields contain equivalent information, there should be a "Person Name" fieldtype defined in the system, and all of appropriate fields should be defined to have that fieldtype.

A fieldtype definition (which is itself stored in a chit), includes a datatype, a name, a description, a default label, and optionally a script. The description provides human users with a more lengthy explanation of how the fieldtype is intended to be used. The default label can be used by programs for fields which do not supply their own labels. The script allows the user to provide additional intelligence to the fieldtype for such things as validity checking and correlation.

Finally, an entire chit can be given a chittype. A chittype defines a class of related chits, and enforces the existence of a set of fields within the chit. The chittype definition (which is itself stored in a chit) includes a name, a description, a set of field definitions, and a script. Every chit of a given chittype is forced to have at least the set of fields described in the chittype definition. They may however have additional fields. Chits are not required to have a chittype.

The chit manager will provide a number of standard chitTypes to store information which the Digital Assistant needs to do its work. These will include Person, Group, Schedule, Meeting, Task, Message, DocumentType, Document and Application. The latter two are used to provide access to information and code which is not part of the chit manager.

Putting it all together, you get a structure which looks something like the following. Keep in mind this is not a description of the actual implementation.



### Concepts

This section goes into somewhat more detail on some of the concepts used by the high level model of the Chit manager.

#### Data Types



The chit manager will provide a very robust list of datatypes which the user can use to build chits. These built-in DataTypes will include: Byte, Number, String, ChitName, RichText, Video, Sound, Time, Sequence, ChitReference, Opaque, and Domain as well as Arrays of each of these types. A couple of these data types have special properties. Fields with "ChitName" dataType provide a name for the chit itself (as opposed to the name of whatever is being described by the chit). Fields with the dataType "Opaque" can not be interpreted by the chit manager, and require another application to open and/or manipulate them.

In addition, a mechanism will be provided to allow the user to add new data types to their personal dataspace. This will work in much the same way as adding CDEV's to the Mac system and/or adding RSSC's to ResEdit. The user will copy a DataType Definition Chit into their dataspace, and then they can add/edit fields of that type. Conceptually, if not literally, a datatype definition chit is a code object. It will provide the ability to display values, edit values, and to translate values into other datatypes. It will be able to provide these capabilities either through template/script mechanism, or through actual code. Obviously datatypes which use code will not be portable to other systems.

In general, fields of a particular type will have references to the datatype definition chit for that type. In addition, there will be a mechanism for having variations within a datatype. This is particularly useful for datatypes such as rich text which want to support both fixed and variable sized values.

### Values

Values come in two types: fixed length ones and variable length ones. That terminology however hides some of the usefulness of the distinction. Fields with fixed length values have a fixed amount of space allocated for the value; and that space is allocated directly in the chit structure, so it can be accessed quickly. Fields with variable length values do not have fixed space allocated for them, and have the added advantage of being able to be manipulated in a random access manner. This makes them particularly useful for fields with very large values.

If you need a familiar frame of reference for using the chit manager, you can view each chit as a "file", with the fixed length fields making up the directory entry, and the variable length fields being the forks of the file.

### Fields

In addition to the obvious roles of providing a datatype and a name for every value in the system, Fields are also responsible for helping the chit manager display the value. This means that when they exist, any additional "hints" about the display of the field (such as font, size, minimum rectangle, access, etc.) should be stored as part of the field description.

### Field Types

Field types in the chit manager serve an similar purpose to indexed fields in traditional databases. They provide quick access to those chits which have a particular kind of information in them (but not necessarily a particular value). You will still be able to get at the other fields which do not have a declared field type, but they will be less efficient both in terms of speed and storage.

In addition, we probably want to examine different levels of actual value indexing. Since we have already committed to full content based retrieval, that may be sufficient for quick look-up of particular values. If however more is needed, there is a range of options we can pursue. The current low level model provides for one medium level solution.

## Chit Types

Chit types actually provide several different advantages/capabilities. First, they provide structure and regularity for those types of information which need it. For instance, "address book" chits, which are accessed by several programs as well as the user, need to require certain types of information for the applications to work. Second, for technical reason chits which get their fields from chit types are more space efficient than those with just local fields. Third, Chit types provide a means for classifying chits. Thus it is possible and useful to have a chit type which imposes no fields just to classify a set of chits.

One of the trickier aspects of the type mechanism is giving a type to chits which already have fields. When an untyped chit is assigned a type, the chit manager first looks at all of the existing fields in a chit to see which ones match the fields which the new type requires. These fields are consolidated into the type. Then, the additional required fields are added.

In addition, the current low level implementation easily supports chits with more than one chittype in a manner not unlike multiple inheritance in C++, with some added advantages and problems. This is useful both for imposing multiple sets of fields as well as for multiple classification. The problem comes in dealing with the possible overlap between multiple types. Whether and/or when we choose to make this feature available to the user can be decided at a later date; it is not strictly necessary for providing the goal functionality of the chit manager.

## **References, Domains, and Scope**

Two of the most important datatypes in the system are the ChitReference datatype and the ChitReferenceArray datatype which allow chits to contain links to other chits. Chits are the lowest level of information which can be passed around by the system, but "no chit is an island". Many fields in most chits will be references to other chits. Thus what the user may see and think of as a single object may in fact be a small network of chits which are themselves a subset of the larger network of chits which is the user's database.

There is in fact far more information which needs to be available to the user through this network than the user actually wants to deal with at any one time. Therefore the chit manager needs to provide a mechanism to limit the set of chits which are "visible" at any one time. This is done by providing "domains" of chits. Each domain of chits can contain its own datatypes, chittypes, fieldtypes, and chits. Chits within a domain can contain references to chits in the same domain or can refer to chits in a different domain, the only difference is the speed/difficulty in following the reference. To follow a reference into another domain, the user/program can either open the destination domain, and thus make available all of its chits; or a call can be made to read the specific chit and make a temporary copy of it in the current domain.

The domains themselves can exist in several different ways. Information which is stored remotely from the user's machine will usually be considered separate domains. Domains can be a simple subset of the user's dataspace which the user has created to organize his information. For instance the user might want to keep work information in a separate domain from personal information. Or, domains can be created as fields in chits. The Domain datatype allows the user to encapsulate a particular set of chits for easy high level manipulation.

When the chit manager is started, it will open the user's personal domain which contains all of the user's personal chits as well as chits which contain references into all of the other domains which the

user can access. Over the course of a session, the user can then open additional domains to get at the information he needs.

## **Documents and Applications**

While a wide variety of information can be accessed and manipulated directly by the Digital Assistant through the Chit Manager, there will still be other applications in the system as well as the documents they manipulate. The Chit Manager needs to provide access to these Documents and Applications so that they can be launched by the Digital Assistant. This is accomplished through three of the standard chit types: Application, DocumentType, and Document. An Application chit contains the code for an application as well a list of the document types which that application can read and which it can write. A DocumentType chit represents a format of data which can be stored in a document (it corresponds to the "file type" used by Macintosh documents). Amongst other things, each DocumentType chit contains a list of references to applications which can read that particular type. The list can be ordered by the user to represent his preference for using each application for that kind of data. A Document chit contains the document type and data for one of the user's "files" as well as a reference to the user's preferred Application for that document.

## **Performance Bottlenecks**

In order to support the performance required of the Chit Manager, the following features will be provided to user programs: direct access to the chit structure including the field definitions and the fixed length field values; a fast call for iterating over all fields in a chit; a fast call for iterating over all chits of a given chittype; and a fast call for iterating over all chits which contain fields of a given fieldtype (which will also provide the offset/index of the field within the chit); calls which provide both direct and random access to variable length field values.

In addition, there needs to be a minimum of overhead per chit for chits which get most of their fields from a chittype. And there needs to be a minimum of overhead for fields which just have a type and no additional display information.

---

---

## **Implementation Model**

This section describes the current low level implementation model for the chit manager. This is only a model of how we expect it to work, and not a byte-by-byte, call-by-call description. It is included to demonstrate the feasibility of the design.

---

## **Assumptions**

The current design assumes that the underlying operating system supports virtual memory and Memory Mapped files.

---

## **Domains**

A Domain is a collection of data types, chit types and chits which are tightly grouped together. Implementation wise, a domain is a combination of a disk file and (when open) a memory heap. The disk file contains a representation of all of the objects in the domain, and the memory heap is used to load the data into memory.

The Domain Manager is responsible for allocating storage in the file/heap, loading/storing chits, and dereferencing chit references. It is expected that a fairly standard storage allocation scheme will be

used. Given current structures, an allocation quantum of 16 bytes seems ideal. There is a 12 byte per object overhead, so a single quantum object would have 4 bytes left over for data.

In the current prototype, the domain's file and the heap are identical, and work as if the file were mapped into the program's memory. This makes loading/unloading chits trivial, but complicates crash recovery and network operation. In the final design some form of domain manager controlled paging will be used.

All objects in a domain will be referred to through a double indirections scheme not unlike Handles in the current Mac Memory Manager (with a few exceptions). Object references will map into offsets into a single Master Block. The master block entry will then map into the object's position in the file/heap. The exact nature of these mappings are yet to be determined. In the prototype, the object reference **is** the index into the master block, and the master block entry is the **offset** of the object within the file and the heap (since the two are identical).

For each Domain, the Domain Manager maintains four data structures: The Master Block, The Free List, the Modification List, and the Backup List. The the first two of these server the obvious functions and need no explanation. The Modification List is a singly linked list of all allocated objects in the heap, in order of modification, with the more recently modified objects towards the end. This is used by asynchronous and "background" tasks (such as the chit daemon and the content indexer) to process changed objects. The Backup List is used to maintain data integrity and to handle crash recovery, as described in the section on that topic.

---

## Objects

Objects have a header which contains 3 values: the object's length, it's "next pointer" in the Modification List (which is also used for the Backup List), and a Modification ID which indicated when the modification was made. (The Modification ID will either be a timestamp, or a sequence number depending on trade-offs to be determined later).

From the domain manager's point of view, all objects in the domains are of one of seven types:

### User Chits

User Chits are used to store the information which the user thinks of as "chits". The structure of a user chit is described at length below under the heading "Internal Chit Structures". In general, each user chit is composed of a header, followed by some number of pairs of field descriptors and values.

### Type Chits

Type Chits are used to store structural information about user chits and are referenced by field descriptors in either User Chits or other Type Chits. The structure of a type chit is described at length below under the heading "Internal Chit Structures". In general, each type chit is composed of a header, followed by some number of field descriptors. In some respects one can think of a Type Chit as a User Chit without the data. Type Chits are used to implement both "Chit Types" and "Field Types", since the only low level difference between the two is that "Field Types" contain one and only one field descriptor and "Chit Types" can have any number.

### Data Blocks

Data Blocks are the primary means of storing variable length values for User Chits. Fixed length values are stored directly in the User Chit structure; but since variable length values need to be dynamically

sized, they are stored in their own blocks. The User Chit will then contain a reference to the Data Block as its "value". Data Blocks have a header which is a back-reference to the chit/field whose value they contain.

## File References

File References are the alternate way of storing variable length values for User Chits. In some cases, for reasons of size, sharing, remoteness, or user/program preference, variable length values will not be stored in the Domain at all, but will instead be stored in their own files. In these cases the User Chit will contain a Reference to a File Reference Object. The File Reference Object will contain a back-reference to the chit/field, and some form of "universal pathname" to find the file. How the Universal pathname will work is not known at this time, but it will probably include a file ID for local files, and some variation of the globally-unique ID's used by the proposed Apple Document Format to handle non-local files.

## Array Blocks

Array Blocks are used to implement array-of-value fields. The current plan is to implement these in terms of a linked lists of buckets. Array Blocks are used to hold the buckets. Given expected overheads, a bucket size of between 8 and 16 entries seems ideal.

## BTree Blocks

BTree Block are used to store the nodes of the BTrees used to index field values in the Domain. How the BTree indexes relate and or interact with the content based retrieval system is yet to be determined.

BTree's will not be implemented in the initial prototype. Instead, each Type Chit will have an Array of pairs where each pair contains a chit/field back reference and a hashed version of the value. The Array may be maintained in sorted order.

## External References

In many cases, chits will refer to other chits which are not in the same domain. When this happens, the chits will contain a Reference to an External Reference Object which serves as a placeholder for the other chit. The External Reference Object will contain some form of "universal pathname" to find the domain's file, and a chit reference within the Domain to find the specific chit. How the Universal pathname will work is not known at this time, but it will probably include a file ID for local files, and some variation of the globally-unique ID's used by the proposed Apple Document Format to handle non-local files.

---

## Internal Chit Structures

This section provide some details of how User and Type Chits might be implemented. Some of the details presented here are based on the current prototype and may change before the final design

## Field Descriptions

Field descriptors are used to define the structure and content of fields in a chit. For every value in a User Chit there exists somewhere a field descriptor which provides information about that value. The descriptor can either be in the User Chit with the value (in the case of local fields) or will be in a Type Chit which is references by the User Chit. Thus Field Descriptors appear in both User and Type Chits.

## Components

Field Descriptors can have the following components. Note that not all of these components are required in every field descriptor, and as noted below, there will be more than one Field Descriptor format to account for different levels of information.

### *Flags (required, 1 byte)*

The first byte of every field descriptor is a flag byte, which amongst other things indicates the format of the rest of the Field Descriptor. The Flag byte also indicates whether the value of the field is to follow the descriptor or if a reference to the value is to follow (is the value Direct or Indirect).

### *Type (required, 1-4 bytes)*

The type in a field descriptor can be one of three things: it can be a built-in type, an add-in type, or a reference to Type Chit. There will be at most 128 built-in types, and so allowing for Arrays of each type, a built-in type definition will fit into 1 byte. For add-in types, the field descriptor must include a reference to the Datatype Definition Chit for that type, and so requires 4 bytes in the descriptor. The same is true for field descriptors types which are references to Type Chits.

If the type of the field is a built-in or add-in type, then the data type of the value is known. But if the type is a reference to Type Chit, then the chit manager must interrogate the Type Chit to determine the data type of the value. The Type Chit may in fact contain several type descriptors in which case several values are associated with the single field descriptor in the original User Chit. This allows for greater space efficiencies for strongly typed chits. In addition, the descriptors in the Type Chit may in fact contain references to other type chits, and so on. However, at some level of indirection, every field descriptor must be able to be resolved into a list of built-in and add-in types which indicate the kind(s) of value(s) which are associated with that field.

### *Length (required 1-4 bytes)*

The length of the value associated with the field descriptor. Since many of the built in types are very small, the field descriptor can be optimized to allow for as little as 1 byte to store the length. On the other hand, the value can be arbitrarily large (*in the prototype, I have assumes that only 3 bytes are necessary. is this in fact enough???*). In the current implementation, the length includes the length of the descriptor as well, and as such serves as the offset of the next Field Descriptor in the Chit.

In the case where the field descriptor type is a Type Chit reference, the length is the total length of all of the values represented by the Type Chit.

### *Label (optional, 32 bytes)*

Every field descriptor can have a label associated with it to provide a human understandable name for the field. Type Chits can also provide labels, and if both are provided, the label in the User Chit will override the one in the Type Chit.

### *Display Hints (optional, 12 bytes?)*

Display hints are provide to help applications understand how to display the field. This allows user/developer to create Chit Types which understand how to display themselves as forms, tables, or whatever is appropriate. This concept is current not well defined, but the type of information

provided by the the display hints might include: preferred font/size/style for value and/or label. Minimum size of display areas, margins for display areas, relative placement of display area, etc.

*Permissions (optional, 4 bytes)*

Who has the right to read and write the associated values. Normally Field Descriptors with this component will appear in Type Chits, but the user should be able to choose to protect individual fields of individual chits. Protection is one of the least thought out sections of the Chit Manager, for information on what is known, see the section below on Data Security.

Structures

As noted already, there will be several different Field Descriptor formats. What is provided here are the formats used in the current prototype. They **will not** be the ones actually used (if for no other reason than they do not account for permissions); but are useful as examples of what might be done.

In the prototype, there are 3 basic Field Descriptor formats. In addition, each basic format can optionally be extended to include a 32 character label to make a total of 6 Field Descriptor formats. The lengths of the three basic formats are 4 (+32) bytes for the short format, 8 (+32) bytes for the medium format, and 20 (+32) bytes of the long format.

*Short*

The short format is used in cases where the type is a built-in type and the length is short. It will be largely used in Type Chits and for local fields.

Flags	Type	Length
Label		

*Medium*

The medium format is used when the type is an add-in type or (more typically) then the type is a Type Chit Reference. It will typically be used in typed User Chits.

Flags	Length
Type	
Label	

*Long*

The long format contains the kitchen sink. It will be used when the user/program/developer has provided display hints.

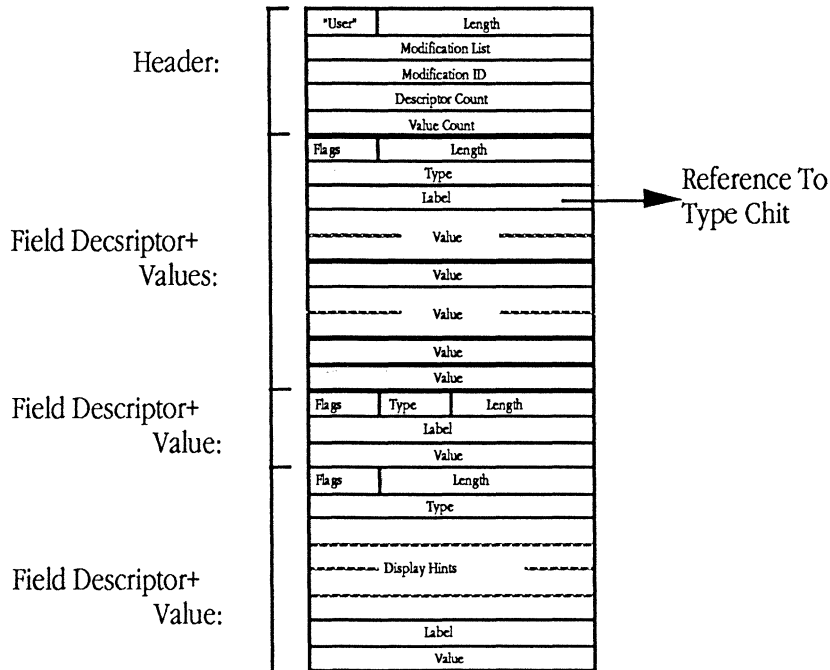
Flags	Length
Type	
-----	
Display Hints	
-----	
Label	

### User Chit Header

The header of a User chit will contain two additional values (beyond those in the standard object header): the Descriptor Count and the Field Count. The Descriptor Count is the number of field descriptors in the chit itself. The Field Count is the total number of field values in the chit. Since a single descriptor may reference a Type Chit which contains several field descriptors, these two numbers may (and often will be) different.

### User Chits

Putting it together, a User Chit is made up of a header, followed by a field descriptor, followed by some number of values, optionally followed by another field descriptor and more values, and so on. Graphically a User chit might look something like:



### Type Chit Header

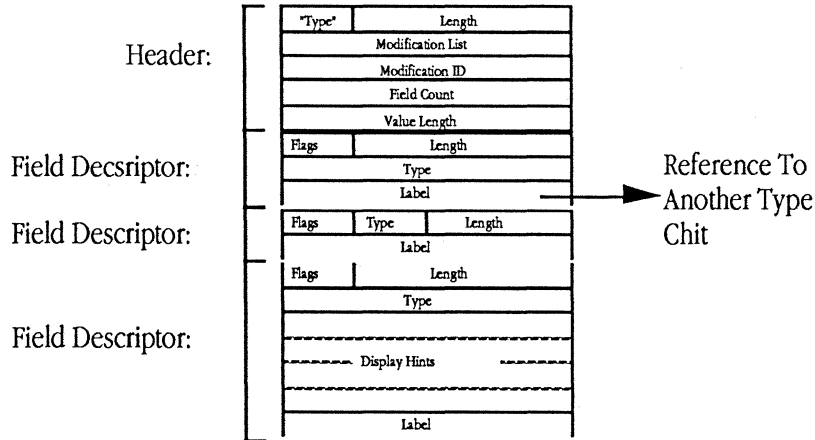
The header of a User chit will contain two additional values (beyond those in the standard object header): the Field Count and the Value Length. As in the User Chit header, the Field Count is the total number of fields represented by the chit, including any from other Type Chits which are referenced.



The Value Length is the total amount of storage which needs to be allocated in a chit to store all of the values for those fields.

### Type Chits

Putting it together, a Type Chit is made up of a header, followed by some number of field descriptors. Graphically a Type Chit might look something like:




---

### Interface

This section provides a summary of the major function calls and procedures which will be provided by the chit manager code. Since we have yet to decide what language we will be developing in, actual interfaces will not be given. Instead a textual description of the functionality provided by the key routines is provided.

On critical part of the interface is however not procedural in nature, and that's the chits structures themselves. While we will not generally publish or support the storage and memory management scheme used by the domain manager, we will publish the structure of User and Type Chits as described above, so that programs can handle these structures directly.

### Domains

The Domain related calls just deals with the domain as a whole

#### New Domain

New Domain will create a new domain. A "pathname", or whatever serves as one in the target OS, will need to be provided for the base file of the domain. Depending on some implementation details yet to be determine, this may actually create a "director" with the given pathname, and a domain file within that directory.

#### Open/Close Domain

Open Domain creates a local workspace for the domain in memory, including chit buffers, a Changed Chit List, and a Change Master Block List. details of the latter two are given below in the section on Crash Safety and Recovery. If the domain is shared, the it registers its use with the appropriate chit server. Close Domain does an Update and then frees the workspace.

### Set/Push/Pop Domain

Set Domain sets the current domain. In general, most chit calls requires an open domain specification; but if the specification is 0, then it uses the current domain as specified by this call. Push and Pop also alter the current domains by pushing and popping open domain specifications on a stack.

### Update Domain

Update Domain writes out all of the current changed blocks to the domain and causes any other registered users of the domain to be notified of the changes. More details of this call are provided in the section Crash Safety and Recovery. The user can optionally specify that all chits in the domain are to be released. There is some question as to where this can be safely done to a single domain or if it must be applied to all open domains.

### Revert Domain

Undoes all of the changes made to a domain since the last Update.

### Recover Domain

Applies the Crash Recovery methodology described in the section below in an attempt to insure the integrity of the database.

## **Chits**

### New User Chit

Creates a new user chit. The user will optionally be able to specify an initial field/type and an initial value for that field/type.

### New Type Chit

Creates a new type chit. The user will optionally be able to specify an initial field/type.

### Copy Chit

Makes a duplicate of a Chit

### Shadow Chit

Makes a Proxy Chit to reference a given chit. The Domain in which the Proxy is to be created needs to be specified.

### Set Name,ID

Adds a Name and/or ID field to the chit (if one doesn't exist yet), and set the values. Either parameter is optional.

### Add/Remove Field

Add Field add a new field to the chit. The user can optionally provide a default value for the field. If the type of the field is a reference to a Type Chit, then it calls Add Type below. Remove Field

removes the field. The current database design is highly optimized for adding information at the cost of making removing a field somewhat more time consuming.

#### Add/Remove Type

Add Type adds a new type to the chit. This is the routine which handles consolidating existing fields into the type and then adds the new type.

#### Lock/Unlock Chit

Locks/Unlocks the chit in memory.

#### Dereference Chit

Given a chit reference, give the user a pointer to the chit in memory. If the chit is in a different domain, it handles chasing the external reference.

#### Dereference Chit to Modify

Given a chit reference, give the user a pointer to the chit in writable memory, and put it on the change list.

#### Release Chit

Indicates that the user is done modifying the chit.

#### Snarf Chit

gets a copy of a chit from out of a closed domain. This chit can not be modified, and is not guaranteed to stay current.

#### First/Next Field

Get a pointer to the first/next field in the chit. it actually returns a pointer to the value, and fills in some call-by-reference arguments with the type, size, etc. of the field. Since this dereferences the chit, the user needs to be concerned about locking the chit into memory.

#### Get/Set Field

Gets/Sets the information about a field which is specified by its label and/or index into the chit.

#### Get/Set Value

Gets/Sets the value of a field which is specified by its label and/or index into the chit.

#### Extract/Parse Structure

While we will publish the internal format of chits so that users can use the chit objects directly; it is also very useful for the user to be able to see the chits without all of the header and field descriptor information. Extract Structure provides this functionality. As a default, it creates straight C structure with just the field values in it. Optionally, the user can specify a label of a type in which case Extract will just extract the fields associated with that type. Parse Structure does the inverse operation. Taking a C structure and filling in the values of a Chit with appropriate information from the

structure. When using this call the user must take great care to insure that the C structure matched the structure of the Chit.

An additional pair of calls may be provided which take a format string to describe the format of the C structure involved. These would be the equivalent of printf/scanf. They would allow fields to be skipped and/or be in different orders.

#### First/Next Access

Gets First/Next access specification for the chit.

#### Add/Delete/Replace Access

Adds a new access specification to the chit, Deletes one, or both.

### **Arrays**

There are a set of calls which are used to support Array type fields.

#### Add/Delete/Insert Element

Adds/Deletes an element to the array. By default new elements are added to the end. Insert Element allows it to be entered at any location

#### Get Element

Get the element as specified by its index

#### Find First/Next Element

Find the First/Next element which returns true when passed to the provided function.

#### Sort Array

Sort the array based on the comparison made by a provided function.

### **Searches**

These two routines hide a multitude of code:

#### Get First/Next Chit by Type,Name,ID

This is a Resource Manager compatibility. The user provides a Type and/or a Name and/or an ID, and the code finds the first/next chit which matches the specification.

#### Find First/Next Chit

Given an internal format of a query command, find the first/next chit which matches the specification. Whether this calls the content based retrieval engine or just the field/value look-up is yet to be determined

### **Changes**

These routine are provided to allow the user/program to be notified when chits change.

### Install/Remove Change Task

Installs the provided function as a Change Task to be called whenever appropriate chits change. By default, the task is called whenever any chit is changed, but the user can optionally filter the calls by chit type or by a list of chits it is interested in.

### Set Change Task Filter

Set the list of types the given Change Task filters on.

### Add/Remove Change Task Interest

Add Change Task Interest indicates that the Change task wishes to be notified about changes to a specific list of chits. Those chits are added to the Tasks "interest list". Remove Change Task Interest removed the chits from the list when the task is not longer interested in them.

---

---

## **Crash Safety and Recovery**

The crash safety and recovery scheme used by the Chit Manager is based on the concepts of "duplicate on write" and keeping around one level of backups.

First, whenever a domain is opened, a local copy of the master block is made. Actually, this will be done by lazy evaluation, duplicating pages of the Master Block whenever they are modified.

Then when the user/programs requests to dereference a chit with intent to write, a copy of the block is made. The local Master Block entry is changed to point at the duplicate and the the original is added to a local list of modified blocks. The user is then free to modify the copy, but the original remains intact.

Finally, when the user requests that the domain be updated with the changes, the following sequence of events takes place: first, all of the blocks on the Backup List are freed. Second, all of the originals from the current set of modifications are added to the Backup List. Third, the Master Block is updated to point at the modified blocks. And finally, other parties which have the domain open are notified of the changes. A similar scheme is used to track changes to the Master Block itself.

This scheme insured that a consistent version of the domain exists at all times, either in the database proper or by reverting to the blocks in the backup list. If we flush the disk buffers on each update, we are even protected in the case of power failure.

---

---

## **Data Security**

*The data security scheme to be used by the Chit Manager has not yet been developed. This section reflects some of the known issues which need to be addressed by this feature when it is designed.*

---

---

## **Types of Access**

The first question is: what kinds of access can be granted or denied to a particular user. Obviously we need to be able to control who can read the contents of a chit; but can you also control the visibility of the chit as a whole? Likewise we need to control who can change the values of existing fields in a chit; but is that a separate kind of access from being able to add new fields to it? What about the right to copy a chit or make a reference to it?

---

## Granularity

Since it is always possible to make individual fields in a chit be references to other chits, protection at the chit level is sufficient to guarantee security. The only questions are related to the user interface, and what support is needed at the Chit Manager level to provide that interface. For instance if we want to hide from the user that certain fields are actually implemented as separate chits, then the chit manager needs some mechanism to tightly bundle several chits together so that they are never seen separately. Or, we can provide real protection at the individual field level.

---

## Access Lists

Since the user's personal dataspace will already contain person chits and group chits, it will seem natural for the user to specify arbitrary access lists for each type of access. Of course there is a serious performance cost in providing this level of functionality, so how much control do we provide to the user?

We could allow the user to specify some small number of different access groups, say 6 (some of which could be specified by shared chits on a server). Together with access for the user and "world" (those user not in any of the 6 lists), this would allow the "who" part of an access specification to fit in a byte. Then (assuming we have less than 8 types of accesses) the ACL for a chit could be a list of pairs of bytes, one to specify the kinds of access and the other to who has those permissions. Then, if the user specifies a default access for the Domain as a whole, only those chits which have different access requirements would need to have any access specification at all.

---

---

## Standard Types

This section contains a list of datatype, fieldtypes and chittypes which will be supplied as a part of the users default dataspace.

---

## Standard Datatypes

These are the the datatypes which will be built-in to the system.

### Internal Use Datatypes

These datatypes are used internally to a chit database, and in general to not relate to anything a user wants to deal with directly

#### ChitName

The name of the chit (as opposed to the name of what the chit describes). This is essentially the "name" used to simulate the Resource Manager with Chits. This Chit Manager does not require Uniqueness, even within a Domain.

#### ChitID

An numerical ID for a Chit. This is essentially the "ID" used to simulate the Resource Manager with Chits. This Chit Manager does not require Uniqueness, even within a Domain.

#### Chit Reference

A reference to another Chit.

#### Domain Reference

A reference to another Domain.

#### File Reference

A reference to a file.

### **Number Types**

Datatypes for dealing with numerical values.

#### Integer

A 32 integer.

#### FixedPointNumber/FloatingPointNumber

*The format for these datatypes is yet to be determined.*

#### Currency

Currency values will include a Country Code, which will be defaulted by the system.

### **Text Types**

Datatypes for dealing with numerical values.

#### Str31/Str255

These are straight ASCII text, useful for names, etc.

#### RichText

Whatever our right text format is. This may be the same the the compound document format, or it may be a text-only subset.

### **Media Types**

Datatypes for dealing with media values. The three types here are used to store actual media images. There will need to be additional types here for constructing and/or combining media. *The Media team needs to help us out here.*

#### Sound

A digitized sound, ready to play.

#### Video

A sequence of video images.

#### Picture

PICT 3, JagPICT, whatever.

## **Chronology Types**

Datatypes for dealing with time values. All time values include a "mask" to indicate the precision of the value (date only, time only, date and time, etc.). Also all time values come in both single values and ranges (as noted below).

### Absolute Date/Time or Absolute Date/Time Range

An absolute location in time.

### Relative Date/Time or Relative Date/Time Range

A relative location in time (after 5 minutes, or 3 days from NOW).

### Recurring Date/Time or Recurring Date/Time Range

A recurring date (every Third Monday of the Month).

## **Utility Types**

Generally useful datatypes.

### Position

A 4-space position (x,y,z,time). Units for x,y,z,and t may also be specified.

### Relative Position

A relative position in 4-space (+10x, -20y, +0z, +5 minutes 12 seconds) Units for x,y,z,and t may also be specified.

---

## **Standard Chits**

Here is a list of the standard Chit types which will be built into the system.

### **Utility Chits**

These are some general use chit types which are referenced by other chits.s

#### Map

Used for specifying a physical location of something. This will include a picture and positioning information to relate the map to other maps. Maps may also be build into a hierarchy/network where each maps knows its location within a set of larger maps and vica versa.

#### Sequence

A list of (object,relative) position pairs.

#### Room

Fairly low level, but generally useful when dealing with things like meetings. Rooms will have schedules and a Physical Address.



## Filesystem Related Chits)

These types are described above in the section "Documents and Applications"

Application

Document Type

Document

Active Folder

## Identity Related Chits

These are things which can be communicated with

Person

A human being. A Person Chit includes a name, a list of (address, DateTime range) pairs, a schedule, and a talk list.

Company

An institution. Includes a list of addresses (of various types), and a Group of Person chits.

Machine

A computer which is electronically accessible. This includes servers, personal machines, and other such devices.

Group

A group is a list of other Identity Objects

## Communications Related Chits

These chits are used when communication with other Identities. The first few are essentially different access paths.

Physical Address (location)

A Map and a location within the Map. Used to answer questions like "where does Joe sit?" or "How do I get to Valley Green 5?"

Mail Address

Postal Address, includes a Country Code to determine format.

EMail Address

A user's electronic Path. This needs to be able to handle all of the different kinds of addressing used.

Phone Address

A Phone Number. Includes a country code to determine format.

### Message Header

Tells who sent a message, when it was sent, when it was received, who else it was sent to, etc.

### Voice Message

A Message Header, a Sound, and a list of attachments.

### Text Message

A Message Header, RichText, and a list of attachments.

## **Record Related Chits**

These are used to keep a record of information

### Conversation Notes

Contains a Group, a Time Range, and Rich Text

### Meeting Notes

Contains a reference to a Meeting Chit and Rich Text

## **Scheduling Related Chits**

These types are used to maintain a person's (or a room's) schedule.

### Task

Something which needs to be done **by** a specific date/time. Includes a duration, a deadline, a description, a requester and list of other tasks/events which it depends on or depend on it.

### Event

Something which needs to be done **at** a specific date/time (essentially a meeting). Includes a location, a data/time range (perhaps recurring), a group, and an agenda (a rich text description of what is to be done).

### Schedule

A sequence of Events. Schedule can either be exclusive (no events can overlap) or non-exclusive.

## **Request Related Chits**

These are special case communications. Which the software knows how to process.

### Request for Task

A Task which one person want to add to another persons list of things to do.

### Request for Meeting

An Event which one person want to add to another persons list of things to attend.

### Request for Cancellation

A cancellation of either a Request for Task or a Request for Event.

### **Command Related Chits**

These are used to perform actions

#### Script

A sequence of commands. *(The format has yet to be thought out)*

#### Query

A request for Chits. *(The format has yet to be thought out)*

#### Button

An object which when manipulated (click, double click, etc), performs a script.

#### Processor

An objects which performs a script on chits what are given (dragged, etc) to it.

---

---

## **The Bundler**

The Bundler is a tool which packages up a set of chits for transfer to another system/domain. There are two major issues which the Bundler needs to deal with. First, since chits tend to be highly interrelated through references to each other, sending an isolated chit to another system may not be very useful. On the other hand, to send every chit it relates to may mean sending user's entire database. The Bundler is the piece of code responsible for making the decision about what goes and what stays. It can operate automatically based on a set of heuristics and user preferences, or the user can intervene and edit the Bundlers selection.

Second, the Bundler is responsible for deciding which values need to be aliases in what way. If possible, it tries to determine what datatypes are available at the destination of the transfer, and makes the decision on that basis. If such information is not available, then it alias all non-standard datatypes possible. Once again the user is allowed in intervene and help the Bundler make these decisions.

*The User Interface and functionality of the Bundler is yet to be determined, and is dependant on the ongoing UI committee discussions.*

---

---

## **The Chit Application**

For the purposes of this document, the "Chit Application" is a tool which can be used to access and manipulate chits. Most, if not all, of the functionality represented here will be folded into the Digital Assistant. The reason it is listed here is a matter of focus. The Digital Assistant is focused on helping the user manipulate his environment. Given that focus, certain pieces of chit specific functionality may well fall through the cracks. By conceptually tracking a separate "chit application" here we make sure that all of the required power is provided to the user. When the design is complete, and all of the

relevant pieces of this application have been folded into the Digital Assistant, then an evaluation will be made based on what's left to see if an additional tool will be needed.

Current discussions by the UI Committee seems to indicate that the Viewer will provide all of the required functionality, and so a separate Chit application will not be needed

---

---

## Chit Daemon

The chit daemon is a background process which provides an added level of intelligence to the user's database. Each field type, chit type and datatype can declare an "On NewChit" and/or "On ChangedChit" message handlers in their scripts which should be run by the Chit Daemon. While these procedure can do anything the user wants, the purpose of this facility is to allow the database to "improve itself" over time. For instance, there could be a field type which knows how to extract its value from chits. All the user would need to do is define that a chit has a field of that particular field type and the chit daemon would fill in the value. What makes this more powerful than a traditional "calculated field" mechanism is that the chit daemon could even be "trained" to add the field to the chit when needed. One particular use of this is the ability for the daemon to build relationships between chits during the system's idle time.

Whenever a Domain is opened, the Chit Daemon first scans the domain for type chits which have scripts, and in particular ones which have appropriate message handlers. If any are found, then the Chit Daemon installs an Change Task for that database to receive notice whenever a chit is changed or added. When the Change Task is called, it then sends a message to each appropriate message handler. Some mechanism needs to be added to deal with changed over a network: only one system should send the message for each change.

---

---

## Prototyping Plan

The "proof of Concept" phase of development is a little different for the Chit Manager than it is for many of the other sections. The functionality of the chit manager not fundamentally a new idea, and as such there is little concern as to whether or not it can be done at all. Instead, the issue is whether or not the planned implementation will provide a performance and interface which will satisfy the user/programmer. This means that a quick-and-dirty prototype of the functionality will tell us nothing of interest. The prototype must contain "real code" for the critical sections of interest so that we can see what effect they have on the performance of the final product.

---

---

## Concepts to be tested

The following are the basic pieces of the code which need to be tested:

### File/Memory Management

The file and memory management scheme used both in the Chit Manager and in the user's process need to be implemented and tested. This is the primary bottleneck for all of the functionality, and its implementation is critical to testing the crash recovery and notification mechanisms.

### Update/Crash Safety

This is the closest the Chit Manager comes to an actual proof-of-concept. The planned transaction mechanism needs to be implemented and tested to make sure there are no hidden windows of

vulnerability which could leave a database in an inconsistent state. In addition, the transaction mechanism itself provides a significant bottleneck for through-put, and we need to collect data so that the system can be tuned.

### **Multiple User/Network**

We need to see if the architecture for chit sharing and networking will work. This may not actually be tested over a network, but rather through a local simulation of concurrent access.

### **Notification**

The planned implementation needs to be tested to see if all of the "users" of a particular chit set get notified of changed within an appropriate amount of time. In addition, there is a significant storage overhead associated with the planned notification scheme, and it is possible that it may be faster than required. If this is true we may need to "downgrade" the mechanism to take more time and less space.

### **Indexing**

The biggest bottleneck will of course be the indexing mechanism which will be used to process query requests, and much of the rest of the design ties into it. There are a lot of questions which need to be looked at in this area. For instance, the current File/Memory Management scheme makes removing a field from a chit a more costly action because of trade-offs which were made to optimize indexing. We need to see if it really makes a difference.

### **Interface**

Lastly, we need to determine if the interface to the chit manager which we plan to provide is acceptable. This will be accomplished by the Viewer/DA prototype trying to use the Chit Manager Prototype.

---

## **Schedule**

This is the schedule for developing the prototype. Current plan is to do all of the work on 68000 under Blue until the very end. This is justified by the fact that most of the work which is assigned to the Chit Manager is to be performed "in the background" and so it can not afford to take up a large percentage of the CPU time. Thus by getting it to work acceptably on a 68000, we are assured that it will not effect performance on the final hardware.

### **Phase 1 (May 28), Basic Prototype**

This is the quick-and-dirty implementation needed to keep the DA/Viewer work moving ahead. It has the real interface and file format, and a large part of the memory management scheme; but no Update, Network, or Indexing functionality (searched will be done by brute force). The time allotted here is a little generous to reflect the ongoing work of the UI committee.

### **Phase 2a (June 11), Memory Management Scheme**

The next stage is to complete the actual memory management scheme. At this point the basic access performance can be tested, as can the update mechanism.

### **Phase 2b (June 18) Crash Safety**

The next week in the schedule is to be spent testing and proving the crash safety scheme. A complete recovery suite will not be implemented, but a few critical pieces will be written and tested.

### **Phase 3 (July 2), Notification**

The next stage is to implement and test the change notification mechanism. This also includes some of the framework for network/concurrent access, but not enough to actually test it.

### **Phase 4 (July 23) Concurrent Access**

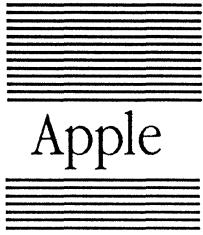
Next, we will complete the model for concurrent/network access. At this time, it is expected that this will actually use a local simulation of concurrent access, since the real network model is very dependant on real multiprocessing. If the state of OS software available is better than expected at this time, then we may change plans.

### **Phase 5 (August 27) Indexing**

The next step is to implement and test the indexing scheme. The large amount of time allocated here is because we are likely to test parts of more than one scheme before settling on the final design.

### **Phase 6 (September 10) Clean Up**

The final step is to implement a few of the smaller interesting pieces of the functionality such a field consolidation and multiple inheritance.



Apple

Jaguar Software

Integrated Media

Design Specification  
Special Projects

May 1, 1990

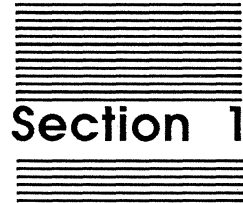
Return Comments to:  
Galyn Susman, Erik Neumann, Bill Aspromonte  
Phone: 4-4856, 4-0240, 4-9047  
AppleLink: SUSMAN1, ASPROMONT  
MS: 82D

Apple CONFIDENTIAL

---

---

## Introduction



Section 1

## Animation

---

---

### Introduction

This section is a description of the animation support provided by Jaguar software. *Underlying Services* describes some system level services necessary to support animation. *BOOP* covers the fundamental animation architecture including object structure, control and interaction. The *Animation Toolbox* is a collection of libraries that provide support for some of the most common and/or most useful animation algorithms. *User Interface and User Functions* covers I/O related issues. *Animation Editors* describes the basic editors necessary to create animations. *Issues* mentions the hot topics of real-time requirements, Pink, and integration.

More information on this topic is available in the HyperCard version of the software Design Specification.

'Computer animation' is a term that is used to describe any application in which a computer aids in the generation and manipulation of moving images. This broad category of activities includes motion control, computer-assisted cel animation, keyframe animation, color table animation, solid model animation, and 2D and 3D simulation. This wide range of activities is of interest to an equally wide range of markets. With improvements in computer speed, size, and cost, computer animation is now widely used in industry, science, manufacturing, entertainment, advertising and education. The following paragraphs cover the primary goals of animation on Jaguar.

1. Real-Time Performance. Although some people will use Jaguar for non-real-time animation applications, the vast majority of people will expect real-time interactive graphics (and sound) from a machine with the capabilities of Jaguar. Note that users will also want to display the results of non-real-time animations in real-time (either for display or recording to videotape), this is especially true since frame-by-frame videotape recorders are very expensive.
2. Ease of Use for Developers. Animation, sound, and video capabilities must be easy for application developers to use, so that they are encouraged to use these capabilities.
3. Ease of Use for Users. Some simple consistent paradigms for dealing with time-oriented data such as video, sound, and animation need to be put forth by Jaguar so that users feel comfortable in moving between applications of this type.



4. Data Interchangability. Users should be able to take clips of animation, sound, and video that are created in one application and use them in another. There should be clipboard support so that users can cut and paste animations. Animation & video should also be useable as "still graphics" when the user desires.
5. Extensibility. Jaguar's animation system must be extensible in terms of new hardware and software capabilities. Special purpose hardware should be able to improve the performance of segments of the animation system in an architecturally controlled way. Similarly, the software implementation of the animation system must be modular so that advanced software components can augment or replace existing modules in the system.
6. Hardware Independence. The animation system should insulate the application developer from addressing hardware directly, so that software will continue to work on future hardware enhancements to Jaguar. However, we must insulate without adding a lot of overhead, because the performance-critical nature of animation, video and sound means that developers will be very tempted to go around our elegant system to get the very best performance possible from the machine.

---

## Glossary of Terms

*Keyframes* - [to be completed]

*Descriptive/Assisted Animation* - Animation generation that uses a computer for camera control, inbetweening, and coloring. The input is a collection of images (pixmap), and the output generally is a sequence of these images, occasionally including simple frames that have been calculated from keyframes. Animation produced by VideoWorks belongs to this category of animation.

*Analytic/Full Synthesis Animation* - Animation generation that uses a computer to manipulate numerical data bases instead of images. All characters, objects and environments have a "geometric" description that specifies actions, coloring, sequencing and compositing.

*2.5D Animation* - Either descriptive or analytic animation where the animated objects are described in two dimensions with a back to front ordering.

*3D Animation* - [to be completed]

---

---

## Underlying Services

---

### Timing Services

The same timing services should be used for all dynamic media so that they are consistent throughout. For animation specifically, timing services must provide:

A way to "register" periodic tasks. Examples of periodic tasks are: updating the next frame of animation; or a process specific to an animation object (see *BOOP* below for more on animation objects).

Timing services will provide synchronization by having a "universal time" ruler (eg. a "LocalToGlobal" for time). This will be used to synchronize within an animation (eg. between animation objects), and between different media (eg. sound, video, animation).

It should be possible to use an "interval" method of dealing with time. This helps to support motion blur and collision detection. Time intervals are used instead of just looking at instants of time (this is to avoid the "jaggies" in the temporal domain).

---

### Object Storage

A database is needed to store animation objects and their component parts. For example, a 3D animation object might have a 3D model associated with it, as well as behavior and motion scripts. A pixmap object may have several pixmaps associated with it.

Note that in some cases component parts of an object could be shared between several animation objects. This would be to save memory (and disk space), or to have several identical objects. Issues of dealing with references & copies have important implications for how users work with these objects (for example: when changing a shared component, do we clone the component, or change the shared version?).

The chit manager is a likely candidate for being the database. Requirements on the database involve speed of loading, and capacity for having potentially large chits & large collections of chits.

---

### Display Services

Animation objects will use Albert graphics to render themselves as 2D, 3D, or other (eg. video) images. Wilson will be used in compositing to copy objects to off-screen buffers, and to copy off-screen buffers onto the screen.

---

### Compositing

Compositing is the process by which a frame of animation is assembled from its constituent objects. There are three methods of compositing that are necessary to support:

1. Simple Layered Compositing -- In this method, each object is asked to draw itself (see *BOOP* for details) into an off-screen buffer, and then the buffer is copied to the screen. The objects are drawn in back to front order (but only those objects that have changed since the last frame). This is fine for 2.5 D animation, and for some simpler types of 3D animation. This is the best method for high-speed real-time interactivity. Wilson will support this type of compositing, by blitting the graphics or

buffer at speeds high enough to avoid "tearing" of the image (beam-collision). Wilson will also provide some transparency effects through the use of the alpha channel.

2. Z-Buffer Compositing -- In this method, there is both an off-screen graphics buffer, and an associated "Z" buffer. The Z Buffer gives the Z location of each pixel that has been drawn. When new objects are drawn into the buffer, only those pixels of the object that are nearer in Z are actually put down (and their corresponding Z value put into the Z buffer). Objects do not need to be drawn in back to front order. This is a better method for 3D animation than simple layered compositing (it allows for concave, interpenetrating objects), but takes more CPU and memory. Currently we are looking into whether Wilson can accomodate this type of Z buffer updating.

3. 3D Database Rendering -- In this method, there is no off screen buffer, instead each object adds itself to a 3D "database" for the given frame. Any of several 3D scene renderers can then operate on the 3D database to produce a complete image of the frame (the entire frame is rendered at every frame, even if only a small part of the frame has changed). This gives the most realistic 3D imaging (cast shadows & so forth), but is the least real-time responsive approach. This method should be at least feasible, if we don't support it right away.

Compositing will happen in a "grafport", or some new equivalent for animation. This compositing grafport has a list of the objects in its view, and controls their imaging onto the screen. Note that there are issues of integration with the windowing system if animation is to occur on the desktop level (otherwise animation is constrained to within a window's animation grafport).

The "camera" associated with a grafport affects how the scene is constructed (information about camera position is passed to the objects when they draw). The camera should be able to "pan" across a 2D scene, and move more freely in a 3D scene. The same is true of lights (more on this in *BOOP*, below).

The compositing buffer will not be used for every instance of animation on Jaguar. For very simple animations (single-object, "flip-book" style), no compositing is necessary, and this makes for speed benefits. In a given animation, the designer might know that certain objects are never going to be composited with other objects (because they are off in a corner for example), and so can draw themselves directly to the screen. Video may also need to avoid compositing to acheive maximum video speed.

---

## Video Services

Video can be treated as a special type of animation object, which draws itself by displaying frames of video. In this way, an area of video can be added to and coexist with an animation. The area of video can even be moving around, or have other objects move on top of it, just like any other object in this environment. However, if the video is going through the compositing buffer, this might cause enough slow-down in speed to consider skipping the compositing and going direct to the screen. By skipping compositing, effects like animation objects moving over the video will not be available.

Video will also be used as a method of recording and distributing animations. For non-real time (or "near real-time") animations video will be used to achieve real-time performance by computing the frames and recording them frame-by-frame. Note that this usage of video is sometimes not tolerant of distortions from the digital video compression process (because animators want the animation to look exactly as it did before recording to video) so a less compressed, but less distorted video format will probably be required.

---

---

## BOOP

BOOP<sup>1</sup> (Behavioral Object Oriented Paradigm) is the fundamental framework of the animation environment. BOOP utilizes the underlying services defined above for synchronization, display, compositing and storage. What BOOP adds is the object structure, control and basic protocol.

BOOP will be implemented in C++ providing fully overridable and extensible classes. This is because BOOP is designed around the idea of animated objects. An animated object is an entity that minimally understands time and knows how to move and present itself (this will be discussed in detail in the next section.) Beyond these basics anything is possible. An object might have 2D or 3D appearance, or no appearance at all. It might scale or change color. It might know how to walk or run or jump or eat. An object may be a character, or it may be a camera or light that affects the resulting appearance of an animation. Of course, we will need to provide a core set of animating objects as examples of some of the kinds of objects that can be built. But we expect developers, artists, and users to come up with the bulk of the animation objects.

With this context set we will discuss in more detail the components of an animating object and describe the environment in which these objects will exist.

---

### The Basics of BOOP

The basics of BOOP contains all of the underlying object mechanisms necessary for general animation support. This includes a basic object structure and control, and object interaction protocol. This framework will satisfy our developer's and user's needs by providing the following:

- Encapsulated animation objects that can be cut and pasted between a variety of applications.
- An extensible object structure that encourages the proliferation of a wide variety of animated objects into a larger body of applications.
- Application and user generated control "languages" that allow simple but powerful animation specification.
- Beginning protocols so that objects can not only animate themselves, but also have effects on other objects' animations.

The basics of BOOP does not include the more sophisticated model of object interactions. This is described in the advanced features section. The implication being that the basics must be provided by ship time, while the advanced features could be provided at a later date (though it would be nice to do both, of course).

---

<sup>1</sup>BOOP (both in name and in architecture) originated in the ATG Graphics and Sound group in October of 1987. If a better historical perspective is needed, please refer to the document by this name and date.

## Animation Objects

An animation object is an object in the traditional sense of the word. It is composed of state which defines the status of an object at any given interval of time, and behavior which defines the object's capabilities.

Any entity that wants to animate, or have an effect on another animating entity, can be derived from an animation object. This includes characters, cameras, light sources, and even microphones. These objects are all derived from an animation object base class called TAnimObject. A TAnimObject has state that includes current orientation (i.e. location, rotation) and current time (TTime) which is relative to a clock (TClock). Specific objects have additional state. For example, a character might have additional shape and property information. A light might have directionality and color.

A TAnimObject also has two guaranteed behaviors: PrepareMeO and PresentMeO. PrepareMeO makes sure that all the state of the object is evaluated for a given time. PresentMeO determines the presentation of an object given this updated state. A character might have a PrepareMeO that executes methods to calculate its current position and orientation and a PresentMeO that generates an image given this data.

Animation objects are registered with a compositing grafport discussed above. At any given time this manager messages all of its associated objects to prepare themselves and then present themselves. Upon presentation, objects pass back images or other information about themselves (depending upon the compositor) to be compiled into a final image.

Let's begin to develop some sample objects that we can use in examples throughout the remainder of this document. We'll begin with Fred our trusty dog and the fire-hydrant (the story line may not be as obvious as you think.) We also have a camera that will affect the views we see of Fred and the hydrant.

Fred is a 2-1/2D object. His physical representation is described through sequences of pixmaps. His state includes not only the current time and position, but also the current pixmap to be displayed. Fred's behaviors are relatively trivial; all he knows how to do is walk. When Fred is walking his PrepareMeO method messages his WalkO method which selects the next pixmap in the walk cycle.

The hydrant is a 3D object. Its state includes a 3D geometry, a scale, and a color. At this point the hydrant does not modify its position or geometry, so PrepareMeO does nothing. However, it does know how to modify its appearance. The hydrant has two different methods for rendering, one for smooth shading and one for flat shading. By default, PresentMeO executes the smooth shading method. If, however, the object is asked to degrade itself (discussed in the issues section) PresentMeO will execute the flat shading method and return its results.

The camera is a basic 2D camera that can control panning (an effect similar to scrolling) over the field of the animation. Its state includes information like the field of view. It has a behavior that controls its motion called MoveMeO that gets initiated by the PrepareMeO method. Its PresentMeO method passes information back to the compositor that will effect the clipping of the final image. The camera's position and direction will also affect 3D objects in the field of view (if any).

We will provide basic object structure to cover a variety of animation objects including some basic characters, a camera, a light, and a microphone. Developers will want to extend these objects in two ways. First, developers may want to create new animation objects. For example, a visualization object may have an associated file containing a data set to render. Second, developers may want to

elaborate on the existing objects. For example, one might want to take Fred from the above example and extend it to run and jump as well as walk.

## Control of Objects

### Scripts

A script is a special purpose animation language<sup>2</sup> used to control objects in an animation. Traditionally, scripts have been the sole control structure of keyframe animation systems. They provide a concise way of specifying scene descriptions at key frames and interpolation methods used to calculate the inbetween frames. However, a typical script will contain information concerning both how the pieces of one object transform relative to each other, and how all objects move relative to each other. Scripts are traditionally sequential and predefined by nature, demanding that transformations be specified in a linear manner. Combined, these tendencies make scripts cumbersome to both read and create.

For Jaguar we will extend the power of scripts in three ways:

- Scripts can be encapsulated into an object as well as existing as an external controller of objects. Internal scripts will be used to orchestrate subpart movements of one character. External scripts will be used to control the movements of one or many objects. Used together, scripts will be able to direct a complete animation while still separating specific object behaviors from the overall choreography.
- The script language will include commands that are "evaluative" by nature such as loops, variables, and 'if' statements. This will allow scripts to both respond to user input and to react to the state of other objects.
- Users can create scripts directly or by using an application. Applications can aid users by hiding the scripting language with a more interactive interface. Applications may want to bypass writing the scripts 'long-hand' by generating 'compiled' scripts or C++ code. Compiled scripts will still be editable directly, but the commands will be specified by numeric codes instead of text to enhance performance. The implication is that all three types of control (ASCII scripts, numerically coded scripts, and C++ code) will be able to coexist within one object.

Below are some sample script commands:

Get <object type> <name>	Creates an instance of the named object, giving it its working name.
Move <name> to <position> <by, for> <time>	Moves named object to location
Interpolate <property> using <interpolant>	Use interpolant to tween for property (default linear)
Walk <name> to <position> <by, for> <time>	Moves named object to location using walk behavior

---

<sup>2</sup> The language may simply be an extension/specialization of the language used to communicate with the digital assistant.

Run <name> to <position> <by, for> <time>	Moves named object to location using run behavior
Color <name> <color value> <by, for> <time>	Change named object color

In this sample scripting language we have two different kinds of commands. The first three (get, move, interpolate) are commands that can be issues to any object. All objects have constructors (called by the get command), and a physical location (modified by the move command). Different interpolants (e.g. linear, splined) can be used for calculating inbetween frames for any property (e.g. location, color, size). The last three commands are object specific, addressing behaviors that exist in a particular object. When an object is created new behaviors may also be created. For these behaviors to be addressable by a script, the creator must add the new behavior to the scripting language. As a result, the scripting language must be easily extensible to custom commands.<sup>3</sup>

Let's add some scripts to Fred and the hydrant to control their motion. First, the walk behavior will contain a script that will select the next frame out of the walk cycle. Then an external script will *Get* a dog, and *Get* a hydrant. The hydrant will be *Moved* to the right and the dog will be moved to the left. With these initial positions set the dog will be told to *Walk* to the hydrant for 10 seconds.

\*\*\*insert picture of script on one side, executing animation on the other\*\*\*

Obviously we do not want our users to have to create these kinds of scripts all of the time. It will be our developer's task to come up with interesting interfaces to script generation as well as object creation<sup>4</sup>.

### Rules

Rules are a more complex method of controlling object interaction. They specify behavioral dependencies that are evaluated to determine an object's current state. Rules are considered to be a more advanced method of object control and will therefore be discussed in the section that covers advanced BOOP features. We only mention them here for completeness so that one may understand the full range of animation control possibilities.

## **Object Communication**

There are two kinds of interactions that objects can participate in: those with other objects, and those with the user. Object's can be designed to query and be queried. For example, an object might understand the message *WhereAreYou* and respond with its current location. In our example we will make the hydrant understand the *WhereAreYou* message. Then we can add a behavior to Fred to query the hydrant for its location and walk towards it. If we now script movement into the hydrant's location Fred will track it's location.

---

<sup>3</sup>Several issues still need to be addressed. How is a user presented the behaviors of an object so that he knows its capabilities and how to address them in a script? How do we use ASCII scripts to modify C++ objects?

<sup>4</sup>Note that the Vivarium group at Apple has been experimenting with very similar activities in their Playground project.

User's interactions can be programmed into objects in two ways. First, because scripts can have 'if' statements, conditions based on user events can be scripted. For example, a script could contain a statement like

```
if mousedown then walk Fred to (mousex, mousey) for 5 sec
```

which would execute Fred's walk cycle while moving him to the position of the mousedown.

Objects can also be internally programmed to respond to user events. In this case Fred would actually have a behavior called MouseDown that would handle Fred's response to user mouse events. In the first example, Fred's response to mouse events would not be a part of Fred's base class, and would therefore not be a behavior inherent to all dogs. In the second case all dogs would respond to mousedowns by walking towards the mousedown location unless the mousedown routine were overridden.

## Classes

The following are some (very preliminary) examples of the kinds of objects we would provide in the Jaguar animation system.

### TAnimObject

```
class TAnimObject {
    TTime          fCurrentTime;
    TClock         fReferenceClock;
    TGPoint(3D)   fPosition;
    Interpolant    fMoveInterpolant;
    BehaviorList   fCurrentBehaviors;

    TAnimObject();
    ~TAnimObject();
    PrepareMe();
    PresentMe();
    Move();
    Interpolation(property, interpolant);
}
```

This is the base class for all animation objects. Every object that wants to participate in animation will derive itself from this class, overriding the PrepareMe() and PresentMe() routines. Move() simply assigns an xyz value to fPosition based on an interpolation of where it is and where it is going. fReferenceClock can be a link to another clock or the object can actually have a clock of its own. This clock is run by addressing the clock start, stop, rewind, etc. messages. A behavior list is a list of current behaviors. If there is a move in process, Move() is added to the behavior list so that PrepareMe() can know what to call. Interpolation sets the interpolant to use for a given property calculation. In the base class the only property that can be interpolated is the position.



TCharacter

```

class TCharacter :TAnimObject, MGraphic {
    TSound                fCurrentSound;
    TImage                fCurrentImage;
    Interpolant           fRotateInterpolant;
    Interpolant           fScaleInterpolant;

    TCharacter();
    ~TCharacter();
    Rotate();
    Scale();
}

```

The TCharacter is one possible base class for a character object. It is derived from the TAnimObject so it already has basic time references and positional information. TCharacter adds a physical description by also referencing the MGraphic base class (Albert has made this a mixin. What will be Jaguar's standard for this sort of thing?) MGraphic includes information about the objects geometry, its properties (represented by a bundle) and its transforms.<sup>5</sup> Two possible additional behaviors might be to add scaling and rotating capabilities to the object. Obviously, controlling its sound, and adding attributes like walking, running, etc. would be the kinds of behaviors that we could model here.

TCamera

To be determined upon review of Albert specification.

TLight

To be determined upon review of Albert specification.

**Example**

Let us look at a more complicated example so that we can see some of the power of this system.

First we will extend dogs to include the behaviors of barking, and sitting, as well as walking. The class structure would look something like this:

```

Class TDog (TCharacter ) {
    Boolean        fIsWalking;

    TDog();
    ~TDog();
    Walk();
    Sit();
    Bark();
}

```

---

<sup>5</sup>Do we really want to use the MGraphic, or do we want to break out the primitives for our own control? To be discussed with the Albert folks.

}

[We will work this out when doing the HyperCard version of the document. Make sure to include examples of transitions between states and multiple active behaviors.]

---

## Advanced Features

The advanced features of BOOP begin to provide a more sophisticated model for object creation and manipulation. This section introduces the concept of object hierarchy so that objects can be composites of multiple sub-objects. It also presents a complete interaction model that allows objects to have a more substantial effect on each others' state.

### Object Hierarchy

The concept of object hierarchy is simply the idea of nested objects. It is mentioned here because of the potential complexity of nesting animation objects with relative spatial and temporal references.

Let us look at the classic example of a four wheeled car moving across the screen. The instance of the car 'contains' four instances of a wheel. The wheels each have their own relative time frame that cycles on three, indicating the three bitmaps that make up the rotating wheel of the car. Spatially, these wheels are offset at some constant from a point on the body of the car. However, their absolute screen location changes as the position of the car changes. The motion of the car is what sets the clocks on the wheel running.

Clearly, there is potentially a large amount of intra-object communication occurring even in a simple example like this. The mechanisms for specifying this communication must be made as simple as possible. To be determined by early prototyping.

### Extended Object Control and Communication

In the *Basics of BOOP* section above we described an object behavior as a method for performing an action. In our example, walking was a behavior attributed to Fred that described how Fred walked. Where Fred walked to was determined by the Move behavior that received its destination from either an external script, an internal script, or a response to a mouse down. Using a combination of behaviors and scripts, we built an animation that demonstrated interactions between Fred, the hydrant, and the user.

One can imagine that there are many possible behaviors that an object can possess. One can also imagine that it might be difficult to try to orchestrate multiple objects with multiple behaviors interacting in one environment. What we need to provide is a more comprehensive protocol for object interaction.

### Affect Volumes

Although an object can have very complex interactions with other objects, in general there is a limited range in which each interaction is relevant. For example, a mechanical collision interaction with another object can only occur within the range of the object's physical extent in space. For any given interaction, an object has a range of interactivity which is delineated as a volume in space. We call this volume a volume of affectation, or an **affect volume**.

An affect volume may be the exact volume of an object's spatial extent, as is the case with mechanical collisions. However, there are many interactions for which the affect volume is different than an object's physical extent. Consider the case of two magnets approaching each other with opposing poles: the 'collision' occurs before the objects actually touch, so the affect volumes of these objects must be larger than their physical extents.

### Affect Agents

At this point we can begin to think of interactions between objects as being an intersection of their affect volumes. When affect volumes intersect, the resulting interaction is handled by the **affect agents**. An affect agent is a component associated with an object that handles an interaction (of a certain type) with another object. An object may have many different affect agents, one for every type of interaction. For instance, Fred may have an affect agent for collision, one for attraction to fire hydrants, and one for sensitivity to dog whistles. When an interaction occurs between two or more objects (i.e. their affect volumes intersect), the affect agents share relevant information for the interaction.

Every affect agent has an associated affect volume which delineates the range of affectation. For example, Fred's collision agent would have an affect volume equivalent to his body volume, while his attraction to hydrants agent might have a volume representing his field of view. An agent also has information that it can share about itself (in the form of messages) with other agents, and information that it must receive (by message queries) from the other agent. For example, Fred's collision agent might want information like the velocity of the collision and the type of object it's colliding with (i.e. Fred may respond differently if he is colliding with a newspaper or colliding with a car).

There are two basic kinds of interactions that can occur between affect agents: symmetric and asymmetric. A symmetric interaction is one where both objects expect to respond to the interaction. Collisions are symmetric because both parties share information and potentially modify their behaviors as a result.

An asymmetric interaction is one where only one party expects to respond to the interaction. For example, Fred has a hydrant attraction agent that is interested in all hydrants within Fred's field of view. However, the hydrant may not be at all interested in dog's within the area (hard to believe.) In fact, it is quite likely that the hydrant does not even have an agent representing this kind of interaction. One way of handling this is for every object to have an 'I'm a' agent specifying its object type. Then whenever Fred's attraction agent intersects an 'I'm a' agent, it checks to see if the object is one that it is interested in (after all, Fred may be attracted to other dogs as well as to hydrants).

### Interaction Management

The intersection detection of affect volumes and the management of affect agent interaction is handled by the Interaction Manager. The Interaction Manager is an extension of the compositing grafport that moderates inter-object communication. It keeps track of all active affect agents and the current location of the volumes. When two or more affect volumes of compatible affect agents intersect, an interaction has occurred.<sup>6</sup> The Interaction Manager's function is to effect and monitor the exchange of information between affect agents.

---

<sup>6</sup>Actually, the intersection detection is more complicated than this. Intersection calculation should be evaluated across on interval of time as well as a continuous range of space. If we

How do we deal with interactions that span more than one interval of time? What if the interaction is occurring with a subpart of the object? Hopefully, to be determined by early prototyping.

## Examples

[Work this out when doing the HyperCard version of the document.]

Object Collision

Camera

Lights

Microphones and Speakers

---

consider interactions without regard to time (e.g. evaluate collisions at instants rather than intervals in time), then we run the risk of missing or misinterpreting interactions. For example, a rapidly moving object may collide with another object, but because the collision falls between instant of evaluation it is missed. A more general method of evaluating intersections is to extrude the affect volumes extent along its path of motion for that time interval. Thus, an object's affect volume may change over time.

---

---

## Animation Toolbox

The Animation Toolbox is a collection of libraries that will help developers create animations. They would typically be used within the PresentMe() or PrepareMe() procedures of animation object for drawing itself or changing itself across time.

The routines included here fall into two categories: the most commonly used or the biggest bang for the buck (i.e. great results for not too much more effort on our part). Each section will describe the types of routines that would be part of that library. The specific routines to be included will be determined at a later date.

---

## Interpolation

Interpolation is the most common requirement for animation. It is the mechanism by which inbetween frames are calculated. Basically, interpolation routines take two or more points (and possibly other information like bias, tension and continuity), and calculates intermediate points. There are a variety of ways to determine the number of points and their values (not to be outlined here). Jaguar will provide a general interpolation library that will satisfy the most common animation interpolation needs.

---

## Metamorphosis

Metamorphosis is the notion of taking one shape and transforming it into another shape. For example, one might want to take a circle and, over a period of time, turn it into a square. This is a very powerful tool for making the most of very simple graphics. If Jaguar were to provide the ability of metamorphosizing one primitive into another it would add enormous capabilities to applications that generally would not bother with animation.

---

## Image Processing

There is a collection of tools that fall under the heading of image processing. They are not really considered to be animation capabilities, but for lack of a video person or an image processing person we will include a rough sketch here.

### Transitions

The most basic of image processing tools are those that provide for transitions between two images. For example, the cross image effects like the wipes and dissolves seen in Director or HyperCard fall into this category. A collection of high quality, 24-bit image transitions should be provided.

### ADO Effects

The second category of tools are generally lumped under the heading of ADO effects. These effects include image rotation, squashing and stretching, distortion, perspective, and other effects that take 2D imagery and manipulate it in a 3D environment. These types of effects can be applied to both static images and live video. We should provide tools to accomplish those effects that our hardware can achieve interactively.

### Mapping onto 3D Patches

The final category of tools include mapping images onto arbitrary patches in three space. This differs from ADO effects in that the rectilinear aspects of the image are no longer preserved. Methods for achieving this kind of effect also differ greatly from standard ADO types of algorithms (which are generally optimized for rectilinear regions as well as for hardware). This is a powerful tool for adding flexibility to the 2-1/2D animation domain. Routines that accomplish these effects should be included if time permits.

---

### **Color Table Animation**

Color table animation is a time-honored way of providing fast and easy animation effects. Although Jaguar is a 24-bit machine, color tables might be used when expanding an 8-bit image into the 24-bit space. Doing this repeatedly with different color tables can provide a color table animation effect, on an object basis (a unique capability for any system).

---

---

## User Interface and User Functions

The following section discusses aspects of animation that almost all users will interact with, irrespective of whether they are creating original animation. Upon reading this section it may strike you that this functionality really generalizes to all media types, and may need to be moved to the media section of the document.

---

### Animation Player

What kind of media players are we going to ship with Jaguar? The underlying question here is, what will be the fundamental media data formats that we will define at ship date. Presuming that we are going to have an ADF-like extensible data interchange format, we will need to design the basic data types that our developers can then expand on. Whatever those data types may be, we need to provide players to display those data types.

Of course, developers will be able to expand on the basic data type, and developers will provide their own specialized players to display their advanced capabilities. However, as long as the underlying format and basic types are agreed upon, any player should be able to play something of any like file. It is important that we provide the players that will display the basic information of a file.

For animation this may mean saving basic BOOP objects in the interchange format and providing a player to display the basic BOOP types (whatever they may be). What are the storage implications here? Do we impose our basic data types on all developers?

---

### Cut and Paste/Clip Animation

There are several possible methods of cutting and pasting animation. We may need to provide for some or all of the methods discussed. Either way, it is imperative that we supply some basic cut and paste capabilities upon shipping. This can not wait.

The simplest method of grabbing animation is to take every rendered frame, much like video. This animation would only be editable at the bitmap level, losing all of its animation specific knowledge. Presumably, the same encoding and compression schemes that would be used to cut video would also be used here.

The next level of cutting animation is to separably cut keyframe and object information. The idea is that all objects in an animation and their associated paths of motion could be cut. This presumes that the receiving end has the ability to reevaluate the paths of motion, generating the inbetween frames. This method is optimal in terms of limiting storage requirements and allowing some level of editability at the other end.

Beyond this point we begin to enter the domain of more sophisticated clip animation. The first class of clip animation rests on the idea of having separable motions from character descriptions. Motions would be described as generic paths for 'hot-spots' on a restricted character set. For example, an animator might generate a generic walk for any four-legged creature. This walk would indicate how the hot-spots of a character move through space and time. Then a library of four-legged creatures with the specified hot-spots would be defined. A user could then pull down a library of motions and a library of characters and construct his own animation. This is clearly a very powerful tool for allowing users to specialize their animation without having to be artists or animators.

Research is still be conducted to determine the feasibility of this concept. We hope to be able to provide capabilities on this scale.

Finally, we must provide the ability to cut and paste complete BOOP objects with all of their state and behavior. This implies that users will be able to create nondeterministic animations simply by pasting a variety of objects into an environment and watching what happens. At the interface level, we must provide some standard mechanism for inspecting an object to see what its possible behaviors are and how the user can interact with the object. Of course, this is true for more than just animation objects, and will need to be addressed at a global scale.

---

### **"Standard File" of Time**

Standard File provided a way for applications to to give users lists of files without the application having to deal with the mechanics. In the same way, the color picker package provided a way to allow users to choose colors without the application having to understand color spaces.

A package-like entity for time seems to be a desirable thing. We might be able to provide a package that displays a time-line and allows users to sequence data presentation along this time-line, all without the application have to pay too much attention to its existence. This should provide a self-contained, easy way for applications to include pasting of temporal media.

---

### **"VCR Type" Controls**

This may be the same thing as the "standard file of time". This refers to the simple controls found on a video cassette recorder for play, stop, rewind, fast forward, pause, etc. Some form of control like this will be used by many applications that can import animation/video/audio.

---

### **Extended Input Devices**

Much of the 2-1/2D animation world focuses on material artwork represented to use as bitmaps. The mouse is not a very good device for generating this kind of artwork. Most artists prefer to use a stylus. In the same regard, the mouse is a poor tool for specifying 3D motion. Knobs or more 3D oriented devices are much more flexible for motion specification. The use of alternate input devices must be considered likely with this system and should be provided for in any way possible.<sup>7</sup>

Video people are requesting to have a rotary knob control, for moving easily back and forth within a video segment.

Sound people are requesting a hardware knob of some kind to provide for immediate volume adjustment (without climbing around to find the Control Panel desk accessory).

---

<sup>7</sup>An investigation into gestural input should be undertaken if we determine that a more flexible input device will be shipped with our machines. Gesture input seems to be a very promising input medium when specifying temporal information.



---

---

## Animation Editors

Just as there must be editors to generate sound and video data, there must also be editors to generate animations. It is not clear which (or how many) of the editors described below should be delivered at ship time or who should supply these editors (Jaguar or 3rd party developers). However, some animation editing and display capabilities need to be distributed with the machine.

---

## State Graphs

A state graph is a control structure that defines the allowable transitions from one object state to another object state. Each node on the graph represents an object state such as the walking state or the running state (otherwise know as a behavior), and each edge is a condition and path for transitioning between states.

A state is a small executable entity, usually a sequence of pixmaps, that can be cycled through on a repetitive basis. A transition is usually a user event such as a mouse down. By combining states and transitions one can generate an compiled, interactive, animation (this is a technique that video games use).

A graphical state graph editor would be an ideal way of combining sequences and transitions to compose interactive animation from previously generated sequences. For examples of this kind of application refer to Pinball Construction Set (don't remember the publisher) and FilmMaker by L.B.A.

---

## Key-Frame Animation

A key-frame animation system is one that allows a user to set key positions for objects in space, through time, and then has the computer interpolate the intermediate frames. Most 3D computer animation programs are a derivative of the key-frame animation idea. Key-frame animation is a powerful way of generating animation sequences. However, it is generally not the optimal method for a naive user. To produce high-quality animation from a key-frame system one must have a reasonable good understanding of splines and how they control not only positional information, but also the velocity and acceleration of an object.

We can extend the power of a key-frame animation systems to a wider base of users. First, our key-frame system must incorporate both 2-1/2D and 3D characters so that our users can generate the information in the medium which he is most comfortable. Second, we can add style tools that implement some of the fundamental principals of animation that users can easily apply to their objects, avoiding unnecessary understanding of splines. For example, one basic principal of animation is *ease-in and ease out*, the idea of objects slowing into a stop and accelerating into motion. We could provide tools where a user simple requests to ease in or out of a motion and let the computer handle the subsequent calculations.

---

## Basic Object Editor

The BOOP model of animation is based on the 'intelligent objects' that have state and behavior. We clearly need to have an object editor where this intelligence can be assigned.

Objects can have a variety of information associated with them including pixmap sequences, video sequences, interpolation paths, rules, hierarchy, sounds, 3D geometry and characteristics, affect volumes and affect agents. An object editor must allow the creation of some of these parameters and

the assignment of all of these parameters. For example, sounds, pixmap and video sequences and 3D geometry and characteristics are all aspects of an object that would be created in other applications but would be assigned to an objects in the object editor. The generation of affect volumes and agents and their controlling rules would probably occur within the object editor.

A basic object editor that at least allows the construction of objects from pregenerated information (sound, video, etc.) must be provided at ship time. Of course, the ability to do the higher level editing would be ideal.

---

---

## Issues

---

### Real Time

[Issues relating to keeping lots of activity going & synchronizing it, to be completed]

---

### Memory

It is important for animation to be able to use 8 bit and 1 bit pixmaps with the same speed as 24 bit pixmaps. This will allow more animation to be held in memory at any one time. As well, a given animation can be loaded from disk more quickly, if the images are 8 bit or 1 bit pixmaps.

The compositing off-screen buffer can take up a lot of memory, it should possibly be allocated "on the fly" so that it is not using up memory all the time.

---

### Disk Speed

Waiting for an animation to load destroys the interactive experience. Users are accustomed to the fast pacing of television, and delays longer than about 3 seconds cause them to rapidly lose interest. We need to ensure that enough animation/video/sound can be pulled from the disk to keep the user interested. One way to do this is to break the animation into smaller pieces and be loading the next piece while playing the current one. We need to work with the hardware folks to ensure these scenarios can be done

---

### Journaling

Journaling, or "recording QuickDraw calls" is another type of animation that doesn't quite fit into the BOOP scheme, but which might be useful to support. This is the method used by the Farallon ScreenRecorder product. When recording the animation, the QuickDraw calls are patched so that whenever a QuickDraw call is made, it is recorded in some special file. The calls can then be played back by reissuing the QuickDraw calls in the same sequence. This results in a record of a user's interaction with a given program, and is often used for software training. The resulting animation file can often be very compact (unless bitmaps are being displayed). The issue then is whether to support this type of animation in Jaguar.

---

### Degradation

Graceful degradation is a goal of the system, when system resources are being overloaded. Animation objects will be asked to lower their needs for processor or CPU, and the animation object can respond in a variety of ways: Objects can have varying levels of rendering (pixmaps vs. full 3D vs. geometric primitives vs. video) which have tradeoffs in speed, memory, and quality of rendering. Objects can change how much associated resources they keep resident in memory for fast animation. Objects may have other shortcuts in style of rendering, or amount of calculations (eg. collision detection). We will encourage animation object designers to build several levels of performance degradation into their objects by specifying message calls which affect this.

[Related Questions: How do we indicate that a character is in the middle of executing a set of behaviors at a given time. Do we have start and stop routines. How does our prepare routine adjust. What about initialization of state like position from an external script. Where is frames/sec information kept. What is the degradation model.]

---

## Relationship to Pink

### Compositing

Compositing as described above will need to be reviewed with the Pink folks to see how it can fit into their world neatly.

### Albert

[to be completed]

### Animation Toolbox

[to be completed]

### Window Management

[see comments under compositing above]

---

## Integration with Sound and Video

[see realtime above... to be completed]

---

## Interface to External Devices

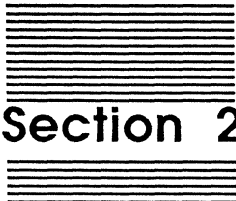
Should we try to provide a standard way to control external devices such as video tape recorders (for example in the frame-by-frame mode), video disks, audio tape players or CD's, slide-show projectors, etc.?

Should we go further and provide software for controlling this stuff, or leave that to third party developers?

---

## Standard File Format

The Standard File Format (described later in this document) can benefit greatly from the concept of animation objects, since an animation object can encapsulate many different types of media: animation, sound, video, and even non-moving text and graphics. The issues here relate to where the code that interprets an animation object resides -- is it in the object itself, or in an external module that activates the object. This also relates to the Player, extensibility, and to the ability to easily exchange documents. [to be completed]

A graphic consisting of a central text element 'Section 2' flanked by horizontal lines. There are ten lines above the text and five lines below it, all in a simple, thin black font.

**Section 2**

## **Video**

---

---

**Introduction**

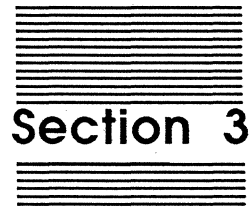
**Compression and Decompression**

**Special Effects**

**Real Time Issues**

**Classes**

**Video Editor**



## Section 3

# Sound

---

### Introduction

This document covers the four major areas of sound on Jaguar and the development system on which they will be implemented. The Audio section covers Audio Objects, Real-time Audio Services (how audio is processed through the system), the Sample Rate Conversion Manager, and Clip Sounds (actual sounds shipped with the machine). The Speech section includes both Text-to-Speech and Speech Recognition and how they are integrated into the system. The section on Timing is extremely important because it discusses the timing resources that will be provided for all media. Topics covered include Timing Services, Ports, Sequences, and Jaguar Sound Event Types. The fourth section covers editors for Sound, Audio Objects, Audio Sequences, Time Ports, and Text-to-Speech. The last section reviews the development system on which we will build the prototype software. The following paragraphs cover the primary goals of sound on Jaguar.

1. End-user focus. We need to give developers the tools they need to write sound applications that satisfy customer's needs or wants. Our expectations are that customers will want to use the following sound applications:

- voice annotation
- voice mail
- soundtracks for multimedia presentations and animations
- telephone control
- remote telephone access to electronic mail and other information
- voice control of computers (speech recognition)
- sound feedback in the human interface
- music synthesis and composition

2. Standard Application Programmer Interface (API). Developers need a standard API for audio, speech, and telephony so their applications will work across a wide variety of different sound hardware platforms. We need to provide for hardware independence. A phone answering application, for example, shouldn't have to concern itself with what type of phone system the user's

computer is connected to. The tools should allow the application to work with any phone system as long as the customer purchases the appropriate hardware and object libraries.

3. **Simplicity.** Sound is a new technology for many developers. The tools have to be easy for developers to use, or they won't use sound at all. To borrow an overused but apt maxim, "Simple things should be simple, difficult things possible."
4. **Generality.** Because sound in personal computing is so new, it is impossible for us as toolsmiths to predict exactly what the big applications for sound will be. Therefore design must emphasize generality and flexibility, so developers can create applications we didn't think of.
5. **Extensibility.** Developers will want to add new functions to our tools. Luckily, extensibility can be accomplished easily with C++ by subclassing objects.
6. **Scalability.** A given sound application should run across the widest possible variety of machines. This is difficult with sound, due to its real-time nature. Some algorithms can be degraded, however. For example, when mixing multiple sounds together, some sounds could be dropped. Where possible, we must provide graceful degradation.
7. **Synchronization.** Often sound playback is most useful when combined with animation or video. We must provide a means for synchronizing sound to other system functions and visa versa.
8. **Standard interface for sound.** We need to provide a standard user interface for recording, editing, and playing sounds. Such an interface would define the way that users edit sound, much like TextEdit did for text on the Macintosh. An editor can also be used by developers to create and edit sounds.
9. **Standard interchange formats.** Users will want to trade and swap sounds. Our tools must provide standard formats for storing audio data.
10. **Reliability.** Our tools must work. Because sound is a real-time process, it will exercise the system in different ways than most standard applications. Multitasking must be taken into account, as multiple processes will want to make sounds simultaneously. This puts a burden on the sound tools and Pink System designers to insure that sound performs properly, so that developers can rely on it.

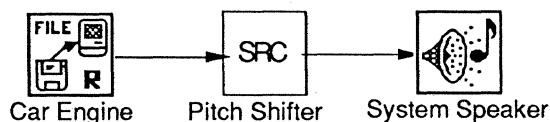
## Audio

### Audio Objects

#### Introduction

An audio object is a signal processing routine that, when connected to other audio objects, can create speech, music, multimedia, and telephony devices. The signal processing can be as simple as mixing two signals together or as complex as voice recognition. The input and output of an object is a buffer(s) of data that needs to be or have been processed. Audio objects can also have predefined parameters that define custom characteristics about each object (for example, the volume on an amplifier or coefficients in a filter). These parameters can also be modified in real-time by external sources.

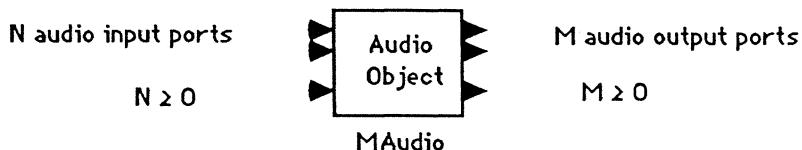
The following is an example of defining and launching a hierarchical audio object. If we were to simulate a car engine, the user would connect an engine sound file to a pitch shifter (i.e. sample rate converter) and finally to the system speaker. The pitch shifter's pitch parameter would be predefined to correspond to the initial speed of the engine. If the user accelerated the speed of the engine, a messages would be sent to the pitch shifter, via the sound server, to change its pitch parameter in order to audibly simulate the increased speed of the engine.



Audio Object Graphical Editor Example

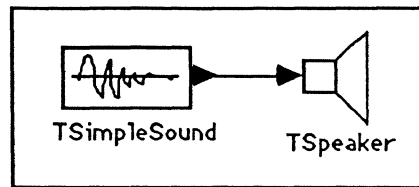
#### Architecture

Audio objects are the heart of the Pink sound tools. Audio objects generate, process, or consume audio data. All audio objects are descendants of the C++ class, MAudio. An audio object can have N audio input ports and M audio output ports.



Audio objects can be connected together by connecting their ports. This is analogous to using patchcords to connect audio components together in the real world. In the illustration below, an audio object, TSimpleSound, is connected to a speaker, TSpeaker.





Audio objects are controlled using C++ member functions. TSimpleSound, for example, has member functions for playing and recording audio from disk files. When TSimpleSound's member function Play() is called, audio data will be fed from TSimpleSound to TSpeaker, which will cause the audio to be heard on the computer's speaker. TSpeaker does not have a Play() function, because it just passively plays whatever audio data is pumped into it. TSpeaker does have a SetVolume() control, however, which sets the volume for whatever data plays through it.

Generally, audio objects are implemented completely in software. However, it is possible for an audio object to represent a physical piece of hardware, such as TSampleRateConverter represents the sample rate converter hardware located in Mazda. In this way, external audio devices, such as third party plug in cards or external boxes, can be represented as audio objects.

Normally, audio objects are run in real-time or, if the system doesn't have enough CPU time, they are dropped. Sometimes it is desirable to run objects in non-real-time. An example would be non-real-time signal processing, where complicated sounds are synthesized and stored on disk instead of played through a speaker. Member functions can be added to MAudio to set and clear a real time flag. When in non-real time mode, the object is executed at a low priority in the background. It is never dropped.

## Audio Players

An audio player is an audio object that represents a sound. Audio players have a Play() member function, so that the sound can be heard. Some examples of audio players are sound files, music synthesizers, speech synthesizers, and tone generators. Because this type of object is so common, there is a mixin class defined for it - MAudioPlayer. MAudioPlayer provides an abstract interface for playing, stopping, and navigating around in streams of audio.

Defining an abstract base class for audio players has the benefit of making it possible to play any object polymorphically, as long as it descends from MAudioPlayer. For example, it is possible play each sound in a list of pointers to MAudioPlayers without caring if the sound is a sampled sound, synthesized musical note, sound effect, or segment of synthesized speech.

The most important member functions of MAudioPlayer are Play(), which starts playback, and Stop(), which stops playback. GetPosition() returns a TTime containing the current play pointer into the sound. This is analogous to the value of the tape counter on a tape recorder. GetPosition() can be called after Stop() to determine where playback stopped. Another function, SetPlayRange(), takes two TTime objects as parameters. It restricts playback to the range in the sound between the two TTime objects. These four functions - Play(), Stop(), GetPosition(), and SetPlayRange() - can be used to implement the familiar tape recorder-style functions - play, stop, pause, resume, skip, fast forward, and rewind - as well as more advanced features needed in sound editors such as playing a highlighted segment.

Another member function is PlayPrepare(), which performs time consuming playback initialization, such as paging in the first few seconds of a sound off of the disk. Calling PlayPrepare() before Play() guarantees that when Play() is eventually called, playback will commence with minimal latency.

Wait() causes a process to block until playback is finished. GetClock() returns a TClock that is synchronized to the sound's playback. This clock's offset is zero when the sound starts playing. The clock advances at the same rate as sound playback. This clock can be used for synchronizing events to playback of the sound.

It is often necessary to indicate a position in a stream of audio. For example, GetPosition() has to return the value of the playback pointer. TTime objects are used to represent time in all parameters and return values in the Pink Sound Tools. TTime is subclassed into various time units, such as seconds and milliseconds. For programmers who would rather think in terms of sound samples instead of some other time unit, a subclass of TTime, TSoundSamples, is defined. TSoundSamples requires two parameters in its constructor, a sample offset and a sample rate. These two together can be used to convert a TSoundSamples object to any other TTime object or subclass.

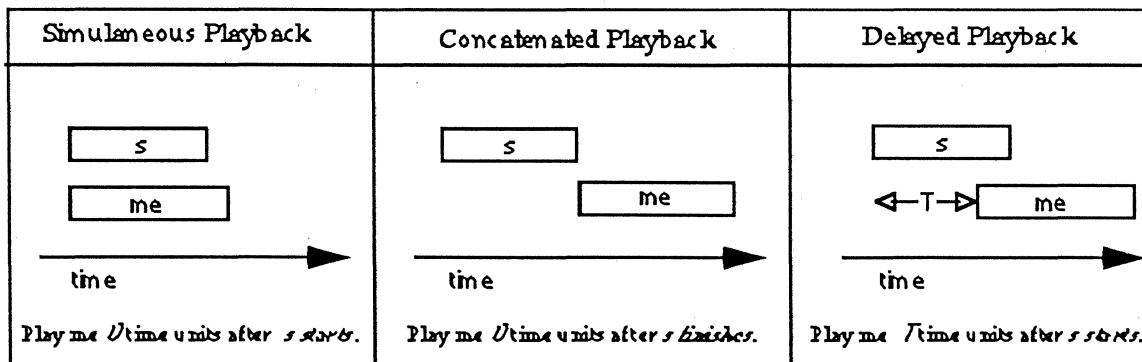
Often it is desirable to play two objects simultaneously. When playing the notes of a musical chord, for example, all notes should be started at once.

At other times one wants one sound to immediately follow another. Concatenating digitized phrases together for voice mail prompts is an example. When playing "You have five messages in your inbasket", one might play "You have" followed by "five" followed by "messages in your" followed by "inbasket".

Both of these cases can be handled by one general rule which can be applied to any audio player:

Play me <T> time units after audio player <S> <starts | finishes> playback.

Examples:



MAudioPlayer defines a member function, PlayWhen(), to handle this:

```
enum RelativeTo { kStart, kEnd };
```

```
PlayWhen(MAudioPlayer& s, TTime t, RelativeTo startOrFinish);
```

## Audio Player/Recorders

An audio player/recorder is an audio player which can also record audio. The C++ class MAudioPlayerRecorder represents an audio player/recorder. It descends from MAudioPlayer. TSimpleSound, which plays and records sound files, is an example of an MAudioPlayerRecorder. MAudioPlayerRecorder has a Record() method to initiate recording. Recording normally starts at the

beginning of the sound. `SetRecordRange` can be used to selectively record into a portion of the sound. `RecordPrepare()` performs time consuming preparation prior to recording, if any.

Recording normally causes audio to be inserted into the existing sound. `ReplaceWhenRecording()` can be called to cause audio data to be recorded over instead of inserted. Audio player/recorders that support multiple channels of sound will often use the replace feature, as sound needs to be recorded into one channel while maintaining synchronization with the other channels. `InsertWhenRecording()` resets recording back to inserting instead of replacing.

## System Objects

System objects are used to represent system resources such as the computer's speaker and microphone. `TSystemSpeaker` and `TSystemMicrophone` are objects that represent the computer's audio inputs and outputs. Each has a gain control. It is possible to connect processors, such as `TReverb`, into `TSystemSpeaker`. Doing this would cause all sounds coming from the computer to be reverberated. It is also possible to redirect audio within `TSystemSpeaker` and `TMicrophone` so that a telephone handset is used for input and output instead of the computer speaker and microphone.

Control-panel type applications can use these objects to implement system-wide volume controls.

## Processing Audio Objects

Processing objects are those which process the input in some way and include reverb, filtering, sample rate conversion, etc. These objects are more fully described in the media document.

---

## Real-time Audio Services

### Introduction

This section includes a description of all the real-time tasks that are performed to produce and consume audio data. There are three main components to the real-time audio services, Audio I/O Interrupt Service Routine (ISR), Audio Access Manager, and the Sound class.

### Audio I/O Interrupt Service Routine (ISR)

The Audio I/O ISR is initiated every 5ms by Mazda on the completion of Mazda DMAing four sample buffers. During the interrupt its only task is to tell the OS to wake up the Audio Access manager.

### Audio Access Manager

The audio access manager is that part of the O/S which manages audio tasks on Jaguar. Several schemes have been discussed and a decision will not be made until tests are done. There are issues surrounding where and/or how sample rate conversion will be done and how we handle graceful degradation.

---

## Sample Rate Conversion Manager

### Introduction

Sample rate conversion is the process of changing the sample rate of a set of data representing a given signal into another set of data representing the same signal at a different sample rate. In its general form, the

problem is to compute signal values at arbitrary times from a set of discrete samples. The problem is therefore one of interpolation between samples of the original data.

Conceptually, this process can be thought of in three stages. First, an oversampled signal is achieved by inserting samples of zero value at the oversampled sample rate. The new signal is then passed through a filter to give the extra samples real values. The filter's task is to prevent images of the input-sampling spectrum from appearing in the output spectrum. The filter necessary is a finite impulse response (FIR) low-pass filter with a response which cuts off at the Nyquist frequency of the input sample rate (up sampling) or output sample rate (down sampling). The third stage is to decimate the signal down to the desired output sample rate.

Sample rate conversion is required for several reasons:

- If the sound file has a rate lower from that of the output hardware (48 kHz), its sample rate must be converted before being sent to the DAC. Lower sample rates are very common due to the increased storage and processing efficiency. This also gains us compatibility with Macintosh (22 kHz) and other Hardware file formats.
- To shift the perceived pitch of a sound. For example, if a piano is sampled at 24 kHz and then played back at 48 kHz the perceived pitch would be an octave higher.
- To route digital signals between the high quality audio I/O system(48 kHz) and the phone, ISDN and Handset (8 kHz).
- When sampling a sound from a constant rate ADC (48 kHz), it is often desirable to lower the sample rate to compress the data and improve computational efficiency for DSP algorithms at the expense of decreasing the frequency bandwidth.

The Mazda implementation of the a sample rate converter can also be used as a:

- Up to a 16 tap FIR filter
- Decompressor
- Sub-band decoder
- Gain section

The sample rate conversion manager is the interface between clients (typically audio objects) and the hardware sample rate converter located in Mazda. Its primary tasks are to:

- receive requests of data to be processed from clients
- Create Channel Command Programs and pass them to Mazda
- notify clients upon completion of their request

## Features

- Variable order The SRC allows us to vary the order of the FIR filter from one to sixteen points by adjusting the numofLobes register. Variable order allows us to decrease the order of the FIR filter to trade-off quality of individual sounds for a higher quantity of sounds. This is not a bad trade-off considering that the decrease in quality will be hidden by the complexity of the final output, resulting from the quantity of many different sounds being played.

- RAM Coefficients Since the 64 word coefficient table is stored in RAM and loaded every frame, the filter characteristics can change every frame. This allows the user to change the low pass filter cutoff frequency in order to support variable order filters and multiple down sampling ratios.
- Linear interpolation on/off By modifying the lerp flag the user can turn linear interpolation of the coefficient table on or off. Linear interpolation places a zero at the output Nyquist frequency, which effectively improves the stop-band rejection of the FIR filter. Turning off linear interpolation approximately doubles the performance of the SRC at the expense of decreasing the quality of the final output.
- FIR Filter By setting ptsPerLobe equal to 1, turning lerp off and ratio equal to unity, the SRC can be turned into an nth order FIR filter (n equals the value stored in numofLobes). The coefficients for the FIR filter are stored in the first nth locations of the RAM coefficient table.
- CDXA Decompression By setting ptsPerLobe equal to 1, turning lerp off, ratio equal to unity, and numoflobes equal to 2 the SRC will perform CDXA decompression. The two filter coefficients are stored in the first 2 locations of the RAM coefficient table. The Input type is set to 1 for level A and 3 for level B & C compression. For every frame (28 Samples) a new parameter block must be sent to the SRC.
- Variable Input Type The input data type can be either 16-bit linear, 8-bit linear, 8-bit binary offset or 4-bit linear.
- Gain Section The gain section can adjust the output signal level from 0 to 8 times its current level with a step size of 1/4096.
- Speed The SRC will be able to convert a minimum of eight 48kHz voices in real-time, with linear interpolation on and 8 points of interpolation of the input signal.

---

---

## Speech

---

### Text to Speech

#### Introduction

There are three distinct phases in the process of converting arbitrary ASCII text into spoken words, phrases, and sentences. The first phase involves deciphering language-specific spelling conventions and generating an unambiguous representation of the speech sounds, or phonemes, required to pronounce a given stream of text. Conversion of text to phonemes is accomplished by means of a set of spelling rules and a built-in dictionary of exceptions to these rules. This can be augmented by the addition of one or more custom dictionaries embodying pronunciations preferred by an individual user or appropriate for a specific application.

The second phase in the process of text-to-speech conversion replaces the relatively abstract phonemic representation with an explicit description of how speech sounds corresponding to the original text are to be realized in context. The output of this phase of processing is a list of the positional variants of the required phonemes, or allophones, along with control parameters specifying the pitch and duration of each item.

The final step consists of accessing pre-stored digital data or synthesized digital data corresponding to each pair of allophones, modifying their pitch and duration in accordance with the control parameters included in the allophone stream, concatenating the resulting sound segments into a buffer of continuous audio data, and playing the buffer as a sampled sound.

Pink Sound Tools will support the use of multiple synthetic voices. These voices will differ in such features as speaking rate, speaking style, baseline pitch, and pitch range, as well as in the language each voice is designed to speak and the actual digital data used to play out allophones.

---

## Speech Recognition

#### Introduction

The main concept behind providing speech recognition on Jaguar is integration. It must be integrated fully into the system and not appear as though it were an afterthought. The plan is to provide speech recognition as input to the Digital Assistant in addition to text. Other requirements are that it be accurate and expandable. Under investigation is Sphinx which is the most accurate continuous speech, speaker independent ASR system in the world. The vocabulary is 1000 words.

The first stage takes the speech input and turns it into strings of phone labels by means of vector quantization. This step also compresses the data into a form that is manageable. The second stage takes these labels and converts them to a word string using Hidden Markov Models (HMM). The goal of training in Sphinx is to determine the transition probabilities in the HMM so that the network will pass only certain types of speech sounds.

It is estimated that this system requires about 60 MIPS for a 1000 word vocabulary. It has been suggested that we reduce this to 30 words with unlimited branching or 150 words with limited branching. A few of the problems to be confronted are microphone placement, limited phone bandwidth, noise, limited vocabulary, false activation, training, and changing the grammar.

---

## Event Sequencer

---



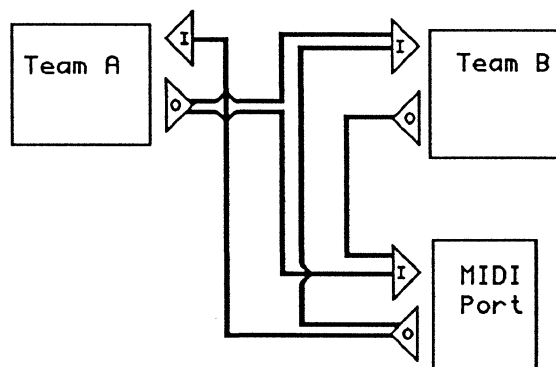
---

### Introduction

This section includes a description of the timing services to be provided by Pink, . There are three main components to the timing services required for sequence events, time (TTime), alarms (TAlarm, and TPeriodicAlarm), and clocks (TClock and TRootClock).

This section also includes a description of the time ports and sequences. Time Ports and Sequences are used to help control the flow of time related data. A good example of this is MIDI. MIDI data needs to have it's time preserved by a time stamp, so that the rhythm information is retained. The MIDI data also needs to be sent at the appropriate time. A user may want to have MIDI shared between multiple teams. Time ports can be connected to each other over a variety of different teams. There are two main kinds of time ports, input, and output. Output ports can only be connected to input ports. There can be multiple connections from and one output port going to many different input ports. There may also be many output ports that are all connected to one input port. Data flows from one port to another when an output port is written to. That data is then transmitted to any input port it is connected to. The teams that own those input ports can then read the data.

An example:



Both teams A and B receive any data coming in from the MIDI port. This data is time stamped for them by the clock used with their input ports. Team B is also receiving data from Team A's output port. This data will also be time stamped by Team B's clock. Team B is send data from it's output port to the MIDI Port's input port. This data is sent when Team B's clock is equal to the time stamp for the data to be sent. Team A is also sending data when it's output port clock says it is time to be sent. Team A's data is sent to both Team B and the MIDI Port's Input Ports.

---

## Timing Services

### Time

Intervals of time are represented in time objects (TTime). These intervals of time are used by the alarms and interval-timing services to represent various points in time. The time objects allow for the representation of time in many different units, that can be used across different time bases. Time units can also be converted to other time units. It is therefore possible to describe time in units of seconds, milliseconds, microseconds, days, SMPTE frames, samples and more.

### Alarms

Alarm objects provide notification to a task that a certain time has passed. Periodic alarms provide notification repeatedly at certain intervals. Alarms are set on the time base of a clock, and therefore need to specify a clock to use. The default clock is the hardware clock provided by the pink operating system. This clock is based on the free running clock included in Mazda, which is set at system start-up time.

### Clocks

Clocks are the heart of the synchronization and timing services. Conceptually, a clock is just a counter. Clocks are used for creating independent time bases, which can then be used for setting alarms and getting the time. They also provide means for defining a relationship between different time bases.

Clocks can be controlled by sources other than the hardware timer. A clock can be controlled by an external source such as a MIDI clock coming from an Apple MIDI interface, SMPTE code coming from a videotape recorder, or even from an audio object such as a speaker (TSpeaker).

A clock measures the passage of time in fixed units (TTime). Clocks can run at uneven intervals, can speed up and slow down, and can even run backwards (e.g., all of these things will happen when a root clock is synchronized to a videotape unit that is shuttling forwards and backwards).

There are two types of clocks: a clock (TClock) and a root clock (TRootClock). A clock is used to represent a local time base which can be used for setting alarms, and getting the time. Root clocks are used to synchronize to a source; such as the hardware timer, SMPTE, or someone's private software counter. One example, is the sound clock which is synchronized to a speaker (TSpeaker). The speaker can set a root clock's time every time samples are played. When a root clock is being set or controlled by a given source, it is said to be synchronized to the source.



TRootClock



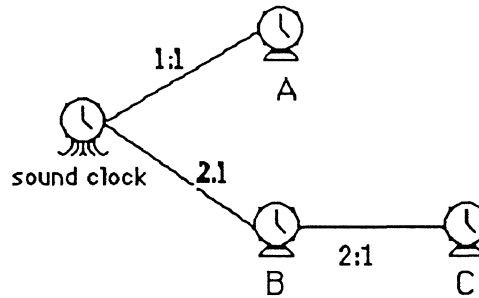
TClock

A clock can be connected to any other clock through a linear function. One of the clocks is the master, the other is the subordinate clock. The master clock can itself be controlled by another clock. In this manner, a chain of clocks can be connected with a defined relationship. A clock can only have its value changed by other clocks. Only a root clock may have its value controlled directly. This is how a new time base can be created and implemented. A clock must be connected (directly or



indirectly) to a root clock in order to provide its functionality. Pink will provide some built-in root clocks such as: MIDI clock, sound clock, and the hardware clock.

It is possible then to define a relationship where alarms are set on different clocks all based on the same root clock. For example, three video clips that are played repeatedly, while synchronized to a sound and where the first video clip is to cycle once per sound; the second clip twice per sound; and the third clip four times per sound. The following clocks would be needed, and would need to be connected by the following functions: clock A in a 1:1 relationship with the root clock, clock B in a 2:1 relationship with the root clock, and clock C in a 2:1 relationship with clock B.



Since the clocks are connected in this manner, it would be simple to change the rate of the second and third video clips to be played three and six times per sound, instead. To do this the only change needed would be to change clock B's function to 3:1. Clock C will still run at half the speed of clock B. All of the alarms set on the clocks would still be set to go off at the same local time, but the local time for clocks B and C would be changed at a slower rate.

The abstract relationship between time, alarms, and clocks allows for users to define timing in their own units regardless of their time base. For example, it is possible to write an animation sequence where different characters move at different rates. This may be controlled by setting Alarms in units of milliseconds, and SMPTE time code. This application could run on any pink machine, and could be controlled by any clock, be it a SMPTE clock, hardware clock, or sound clock. This allows for the user to control the synchronization of the animation with other applications.

TClock has methods to get and set its value (the current time), stop the clock, start the clock, and to connect to other clocks. A single TClock may have to be accessible to many teams. A TRootClock has methods to update its value; this in turn causes the values of all of its subordinate clocks to be updated. TRootClocks can also be made accessible to many teams.

---

## Time Data

Time data (TTimeData) is a class that consists of two important objects. A time stamp, and a chunk of data. The timestamp is a TTime object. The data is a TMemory object. TTimeData is the object for time data. It has methods to create, modify and compare time data. Time data can be used to hold MIDI commands, other non-MIDI representations of musical data, points of interest in an animation, or any other time related data. The data part of a time data object must be understandable on its own. It must be intelligible even if interleaved with other data of the same type. For example, you can not use the data to represent a MIDI command with free running status.

---

## Time Data Ports

Time data ports (TTimeDataPort) are used to send and receive time data. They are similar in concept to IPC, in that events can be sent from one task to another. Events are written to output time ports (TOutputTimePort) and can be read from a connected input time port (TInputTimePort). TInputTimePort and TOutputTimePort are descendants of TTimePort. Time data ports can use a clock (TClock) as the timing source for the port. Output ports can be connected to input ports. Time data ports may also use a clock for their timing needs.

---

## Sequences

Time events can be collected into an ordered linked list called a sequence. Each time event in a sequence can be mapped to any output port for that team. When a sequence is played, the events are sent out through appropriate output time data ports at their scheduled time. A Sequence can also use multiple input time data ports. The data from these ports can be recorded directly into a sequence. Possible uses of Sequences would be for implementing a MIDI sequencer, or an animation sequencer that runs MIDI, animation, and sound effects.

---

## Jaguar Sound Event Types

### MIDI

Pink provides a MIDI Driver that works with the Apple MIDI Interface and compatible Interfaces. This driver has two data ports, one for input one for output. Time data objects that contain a MIDI command for the data can be written to the input port, and will be sent to the MIDI Interface. MIDI information coming in from the MIDI interface will be packaged into time data objects as whole MIDI commands, and will be time stamped using the MIDI clock provided with Pink. Multiple output ports may be connected to the MIDI input port, and the MIDI output port may be connected to multiple input ports.

### SMPTE

SMPTE stands for the Society of Motion Picture and Television Engineers. There is a timing standard that measures time in hours, minutes, seconds and frames. The number of frames per second depends on the standard being used. The MIDI Time Code Standard provides a way for MIDI devices to support SMPTE. Given that we are integrating both video and animation we must support SMPTE.

### Internal Audio

Our internal audio streams will be 16 bit data streams, however we will support other data types. These include 8 bit and floating point types. Converters will be provided for compatibility.

---

## Audio Tools

---

### Introduction

The Sound Tools will provide editors for editing sounds, graphically connecting audio objects, editing sequenced events in time, graphically connecting clocks to ports and ports to clients and changing the voice characteristic of the text to speech synthesizer. The ability to playback existing sounds/sequences and record new ones will also be available. All of these editors are built from objects that any application can create and use directly. The editors serve two purposes, 1) to give developers tools to create sounds and connections of audio objects for use in their programs, and 2) to provide a standard interface for end users for doing the same.

---

### Sound Tool

#### Introduction

This tool will allow the user to edit the actual waveform of the sound. In addition to the expected cut and paste functions the ability to set start, end and loop points will be provided. There may also be some processing options such as crossfade, loop tuning, etc.; options that are standard on most commercially available digital samplers.

This player/editor has two levels of use. At the initial level the user will see buttons similar to those on a tape recorder - play, stop, record, erase, forward, and reverse. Each button will have associated with it a menu of behaviours allowing the user some level of customization. For example, stop might give the option of resuming from the place you stop or going back to the beginning. When scanning through a sound file the nature of increments could be chosen to be samples, milliseconds, seconds, or user defined phrases. At this level people could also record messages, sounds, or music. The idea is to provide as simple an interface as possible that is easy to use.

This tool will allow the more sophisticated user to edit the actual waveform of the sound. There will be a small button that will allow the user to get to a display of the waveform and provide some DSP functions. In addition to the expected cut and paste functions the ability to set start, end and loop points will be provided. There may also be some processing options such as crossfade, loop tuning, reverb, flange, FFT, etc.

See the media document for an example of the first level of use.

---

### Audio Object Tool

#### Introduction

Connecting audio objects together is inherently a visual process. In the section on audio objects, we outlined the C++ syntax for doing this, which is inherently non-visual. The audio object tool would allow developers to create connections of audio objects using a visual editor. Each object would have an icon associated with it. These icons could be chosen from a palette and connected together by dragging and rubber-banding lines between two icons. Potentially, the audio icons themselves could be edited by double clicking on them. This would display the source code for the audio

object. It is possible that we might in the future use a visual digital signal processing language to create the audio icons.

---

## **Audio Sequencer Tool**

### **Introduction**

The audio sequence tool will provide the user with the ability to manipulate 'media blocks' in a time sequence. Media blocks encompass all media but for sound they can be a MIDI sequence, sampled sounds, SMPTE data or internal audio. It functions much like a commercial MIDI sequencer but is more generalized.

---

## **Time Port Tool**

### **Introduction**

In MIDI applications, it is desirable to visually connect time ports together using a graphical editor. Such a tool is supplied with the MIDI Management Tools currently provided by Apple for the Macintosh. This editor, called PatchBay™, graphically depicts applications and their MIDI time ports. End users can patch the time ports together in any manner they please. Such an editor is an intuitive and powerful tool for musicians, who are used to patching physical MIDI cables together. There will be some similar way of doing this for all time ports in Pink.

---

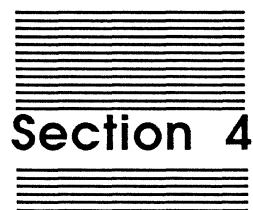
---

## Development Systems

---

### Introduction

This section covers the process by which we hope to simulate the Jaguar sound functionality. Our proposal is to use one Astro card for the I/O, another Astro card for the sample rate converter, a Cub card for Pink sound and the Mac for a clock source. With this hardware we intend to implement many of capabilities described previously in the document. Further discussed are the schedule and hardware requirements.



## Section 4

### Jaguar Data Format

---

---

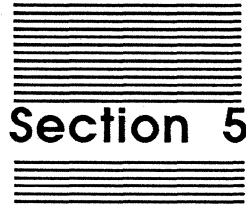
#### Introduction

Jaguar Data Format an extensible format for integrated media documents on disk and in memory for interapplication communication. Its closest analog in the Mac world is AIFF, but JDF will address much more: rich text and the layout in time and space of all Jaguar's media types. The difficulty of transferring files between applications on the Mac is a major aggravation to users. Given Jaguar's richer set of media types and key goal of increased integration of those types, this problem must be addressed.

A central feature of the early Mac was the ability to move text and pictures between applications. This ability to move data between applications was (and is) an important part of the integrated feeling of the machine. But as the document formats manipulated by applications became more complex and additional data types were added, applications became increasingly isolated from one another. Now, most documents can only be read and edited by a single program and the 3-D drawings and richly formatted text cannot be cut and pasted between applications. The integration of numerous file conversion utilities into applications is a cumbersome and inadequate solution — the available utilities in one application always lag the latest file formats of other applications. Jaguar must support easy transfer of integrated media types among applications to provide an integrated feel. Another problem with the PICT format is that unlike AIFF, it does not offer random access and cannot be edited in place on disk. This is an unacceptable omission for a media file format, since media files can be huge, and the data one wishes to access or modify may be only a small portion of the file.

JDF is extensible, so that applications or software modules with special needs can add information whose content is not predefined by Apple; this information will be ignored when the file is read by software which doesn't understand it, but the rest of the information is still accessible. This avoids the problem of continually losing information as a file is transferred from application to application.

JDF's extensibility is supported by a set of classes which provide convenient developer access to JDF data.

The graphic consists of a central text label 'Section 5' flanked by two sets of horizontal lines. The top set of lines is above the text, and the bottom set is below it. Each set consists of approximately 10 parallel horizontal lines of uniform thickness and length.

**Section 5**

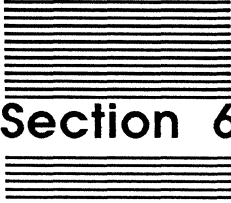
## **Media Sequence Player**

---

### **Introduction**

The Media Sequence Player displays anything in Jaguar Data Format — it is Jaguar's TeachText. Just as it is imperative to ship a simple application on the Mac which can display text documentation, it will be imperative to permit the Jaguar user to display documentation containing integrated media. Since we expect applications which manipulate any of the Jaguar's standard media types to be able to write JDF files, the Media Sequence Player can be used to view the documents created by most applications. Because the Mac does not have a rich JDF and an application for displaying it, users cannot read most documents created by applications they don't own. This is frustrating to users and encourages software piracy since the perceived value of buying an expensive word processor or animation editor just to view files is very poor.

The Media Sequence Player has a programmatic interface so that any application can use it to display media types to the user in a standard way, media types the application itself may not otherwise manipulate. This user interface can be customized to add additional controls (like Standard File on the Mac) while retaining the basic familiar functionality. The Player's functionality can also be extended by adding modules "at the bottom" through Pink's shared library mechanism to display additional media types or enhance the display of existing types. In this way, a user who buys an application that produces documents which require extensions to JDF can view the extended portions of a document in other applications, not just the application which created it.

The graphic consists of a central text label 'Section 6' in a bold, sans-serif font. Above and below the text are two sets of horizontal lines, each set containing ten lines of varying lengths, creating a rectangular frame around the text.

**Section 6**

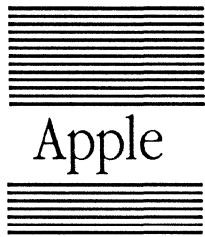
## **Media Sequence Editor**

---

---

### **Introduction**

The Media Sequence Editor coordinates and connects the various specific media editors. The Media Sequence Editor handles the layout in space and time of media elements whose contents are edited by the individual media editors. The user would edit sound, animation, and video with separate editors, but would use the Media Sequence Editor to specify the synchronization of sound and animation, and the precise overlaying of animation and video. The interfaces between the media editors and the Media Sequence Editor are specified so that the user can replace the Media Sequence Editor or individual media editors. Like the individual media editors, the Media Sequence Editor is not intended to be the ultimate sequence editor, but the MacWrite of sequence editors.



Apple

Jaguar Software

Communications

Design Specification  
Special Projects

May 1, 1990

Return Comments to:  
Jim Nichols, Jeff Soesbe  
Phone: x4-0672  
ApplieLink: NICHOLS1, YEFF  
MS: 82D

Apple CONFIDENTIAL



## JaNeT: Network/Telecom

---

### Introduction

Computer communications was a technical novelty in the 70's marketplace. Apple harnessed the complexity surrounding this technology in the 80's with tremendous success. We have the momentum and expertise to shape and direct the evolution of telecommunications in the 90's. Jaguar will introduce the concept of truly *universal connectivity* to personal computing experience.

The only certainty is that the personal computer landscape will change dramatically in the coming decade. Jaguar is designed to anticipate those changes, and to give developers the preferred environment for evolutionary applications development.

The following trends are expected to unfold in the 90's:

- Increased mobility: An increasing amount of work will be done away from the office. This will place increased focus on "plain old" dialup phone service, whether it be from home office, work site, or cellular telephone.
- FAX usage will increase tremendously. Dialup FAX info/shopping/banking services will intensify this trend.
- Global connectivity: Computers will be able to communicate regardless of distance or country. But this won't happen until ease of use approaches of a phone call. Converging networking technologies (eg., X.400 E-MAIL, OSI, TCP/IP) and regulatory environments (eg EEC 1992 convergence, COS, CCITT) will help make this a reality.
- LAN clusters: LAN networks will continue to grow in sophistication, and will form the basis of many corporate workplaces. High speed backbone connections (T1, fiber, etc) will interconnect the LAN clusters. Routers will be play an increasingly important rôle in the interconnection of different LAN and WAN technologies. Dialup LAN access will become commonplace as a result of telecommuting and worker mobility.
- ISDN deployment: ISDN Basic-Rate connections (2B+D) will find growing use, especially in small and medium sized businesses.
- Telecom Agents: High-performance CPUs will cause voice input/output technology to mature in the 90's. Machines will be judged not by raw speed but by "personality": that is, friendliness and the potential for useful work away from the office by mobile information workers.
- PBX technology will compete with LAN and ISDN services. Many larger corporate customers have large financial commitments to their in-house systems; long-term migration will be away from the PBX.

Considering these trends, the Jaguar Net/Telecom subsystem 's (JaNeT) goal is to allow our customers to freely exchange ideas and information, regardless of location, distance, means of communication, or *proximity to machine*.

---

## Goals

JaNeT is designed to be a tightly integrated, versatile, and extensible communications resource that anticipates the future applications outlined in the previous section. In particular, JaNeT comprises general purpose telephony, ISDN basic rate, LAN, and serial datalink services.

JaNet intends to add substantial embedded value to Jaguar by providing maximum communications capability at minimal cost. The two focus areas are telecom and voice I/O. The rationale behind this thinking is as follows:

- The wide-scale connectivity of the telephone network has not been successfully exploited to date. Jaguar will put this connectivity at the service of Apple Mail, FAX, and other wide-area comm applications.
- Practical voice-machine communication will become practical in this decade. The tremendous power of XJS-class cpu(s) should make that a certainty. JaNeT anticipates this development with versatile data/voice interfaces to the phone network - both POTS<sup>1</sup> and ISDN. This should motivate developers to create exciting new applications to take advantage of these capabilities.

Despite Jaguar's tremendous communications potential, JaNeT allows Apple system software developers to exploit analog phone, PBX, and evolving ISDN technologies as a global idea routing tool. Hopefully application developers will be motivated to view Jaguar as a datacom "power amp" that brings the benefits of a intuitive, globally connected phone network to our customers.

---

## JaNeT Architecture

JaNeT is the system software that controls Jaguar's built-in communications hardware. The major components of JaNeT are:

- RALPH: the Real-time AnaLog PHone. RALPH is the analog telephone interface which comprises an analog interface system (ADC, DAC, and AGC), telephone network interface circuitry (the Data Access Arrangement or DAA), and modem emulation software. r could be viewed as a
- BRIAN: the ISDN basic rate interface, which provides Jaguar with ISDN TE1 (Terminal Equipment 1) services on the ISDN S/T (System/Terminal) interface.
- SPAM - the Signal Processing Access Manager: a low-level DMA buffer management task that controls the flow of real-time digital sampling systems inside Jaguar. *It's really a simple buffer manager but I like the acronym.*
- Apple "FriendlyNet" LAN interface which provides an Apple AUI to support the Ethernet variants 10 Mbps thick cable, 10 Mbps thin cable, and 10 Mbps twisted pair.
- LocalTalk interface. This is provided using the ubiquitous SCC silicon.

---

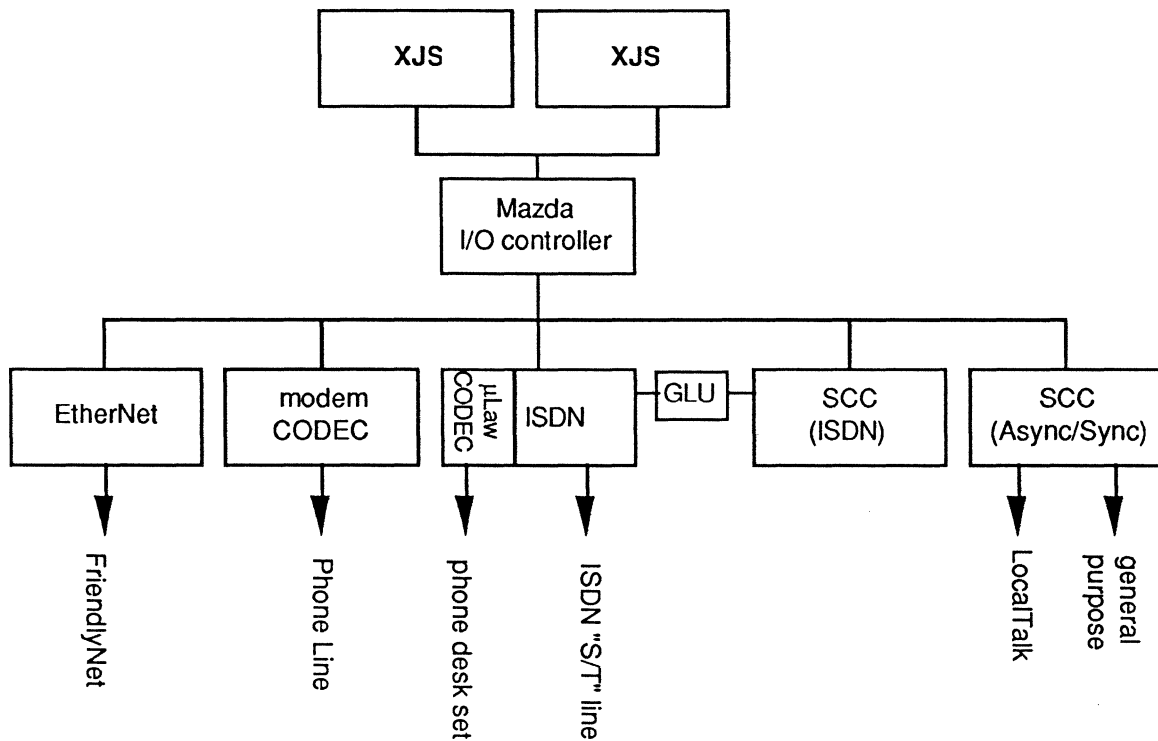
<sup>1</sup>Plain Old Telephone Service: ie., conventional analog copper circuits.

- Asynchronous and limited synchronous communications. Again, the SCC is used for these functions. An I/O controller and DMA circuitry ensure maximum performance with low system overhead.

Card-based (BLT) communications controllers - such as FDDI, token ring, or X.25 - will be based on ongoing N&C technology that's currently designed for nubus. It is anticipated that the A/ROSE - MCP environment will be adapted to the Jaguar pink/BLT environment.

Some JaNeT functions are implemented on current Mac hardware using an MCP<sup>1</sup> card. Jaguar moves some of these functions - such as serial (LocalTalk) and LAN controllers - into highly integrated new silicon. Other functions - such as ISDN and modem emulation - are available in integrated form on Jaguar for the first time.

The JaNeT hardware domain is as follows:



One or more XJS processors perform the protocol processing previously allocated to "smart card" I/O processors. The XJS processing power makes this feasible for the first time. The protocol state machines running on XJS communicate with VLSI controller hardware via the Mazda I/O processor which executes Channel Control Programs prepared by the JaNeT control software.

The Mazda I/O processor is designed to relieve the XJS of time-critical tasks, offload tedious polling. Further, the I/O control program executed by Mazda's several Wankel I/O control programs raises the "atomic unit" of information transfer as seen by the XJS cpu. For example, the lowest level

<sup>1</sup>Macintosh Communications Platform - A 68K-based nubus card; eg, Livonia, John Galt, Token Ring cards

of data transfer on a LAN circuit would be a LAP frame. This frees the central processeors from timing and polling operations, thus minimizing critical real-time response constraints.

The ISDN and codec functional blocks are based on third party VLSI controllers. The Ethernet, SCCs, and GLU components are being consolidated into a single custom package.

## Software

The RALPH subsystem interface provides standard driver level open/close/read/write functionality with additional control calls so that client software can configure and monitor phone connections if required. The presumed client would be the Integrated Voice-Data manager or Connection Toolbox.

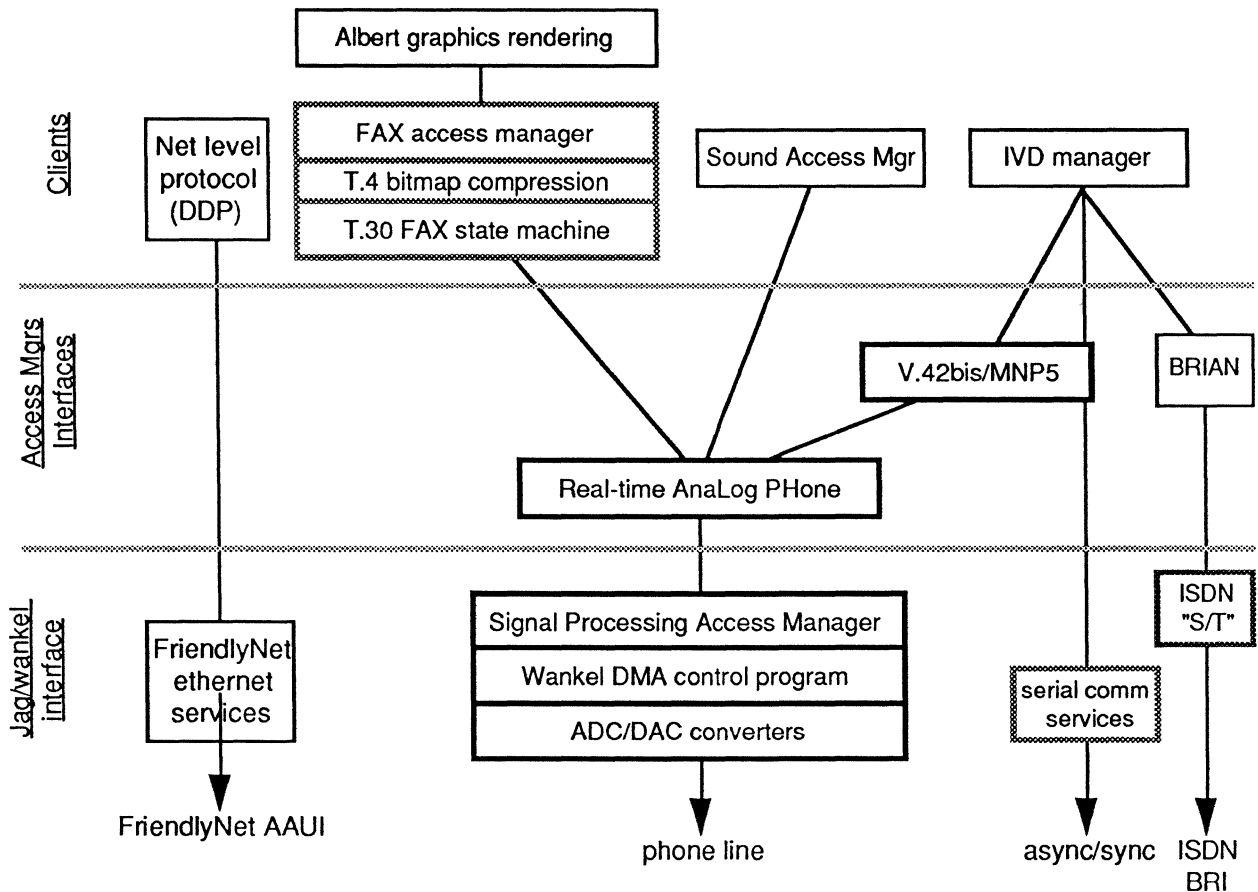
The BRIAN ISDN basic-rate ISDN software appears as an access manager that provides OSI level 2/3 call management. Again, the normal client is the IVD, but sophisticated developers may wish to access the stack directly.

The async/sync ports provide the normal Mac LocalTalk and serial functions, but offload the cpu of these compute intensive tasks. The classic Mac serial driver functionality is provided, along with a general-purpose control mechanism for "bare metal" serial port programming. The Datagram Delivery Protocol (DDP) would typically be the client of the JaNeT ATLAP data link protocol handler, while serial comm services would usually be accessed through the Integrated Voice-Data manager or Connection Toolbox.

The 10 MB/sec FriendlyNet port provides a standard Apple AUI (Attachment Unit Interface) which allows customers to access thick, thin, or twisted-pair ethernet networks using self-terminating connections. The FriendlyNet client is normally DDP; we must provide compatible driver level calls.

An analogy might be made to the PBxxx (Mac parameter block) low-level I/O calls used in Mac. However, Jaguar's I/O system is substantially more powerful than those found on our previous systems; as a result, JaNeT can provide applications with a higher "atomic unit of transfer" (AUT) than previously possible without a "smart" I/O controller. Serial data transfers are performed without host intervention.

The diagram below is a synopsis of the JaNet facilities and typical clients.



**Connectivity summary**

Traditional analog (POTS) services

Connection to the analog phone system (POTS) can be made by parallel connection to the PTT network interface. This allows calls to be placed and received by either a traditional 2500 desk set or by Jaguar. Although the Jaguar user can perform voice I/O via Jaguar control (using the analog I/O facilities described in the sound system section), Jaguar's primary function here is to *enhance* the POTS telephone service for the customer by providing intelligent remote connectivity: the intent is not to replace the desk set.

In this configuration, Jaguar will be able to detect a ringing signal and answer the phone, even when the machine is "off".

ISDN service

The Jaguar can co-exist with a standard ISDN telephone set on the "S/T" interface bus. The ISDN phone is always available for communications, independent of Jaguar. Jaguar voice calls are possible using the Jaguar voice I/O facilities. Communication between the local ISDN phone and the Jaguar, however, will entail establishing a call over the "B" channel.

PBX services

The PBX station connection may use digital or analog signalling. The signalling protocol is often highly proprietary<sup>1</sup>. Both types of signalling can be accomodated. In the analog case, a "T" box provided by the PBX vendor breaks out the voice and data channels, presenting them on standard RJ-11 and RS-232C interfaces respectively. Jaguar can route signalling information and data over the serial channel, while the voice channel now appears to be a conventional POTS line. A digital interface can be developed in conjunction with the PBX vendor to provide a pure PCM voice/data stream directly into the PBX/sync modem interface port.

---

<sup>1</sup>The PBX manufacturer policy being "give away the PBX and make money on the phone stations"

---

---

## Implementation

The process of implementing Jaguar involves many milestones. These milestones are given here along with GIO, Net & Telecomm's plan for implementing JaNeT, the Jaguar Network and Telecommunication subsystem..

---

### Implementation Plan (JIO)

The JIO (JaNeT Implementation Outline) is a five-step implementation plan, moving from early software prototype to early software running on preliminary (emulated) hardware to final software running on final hardware.

The five stages of JIO are :

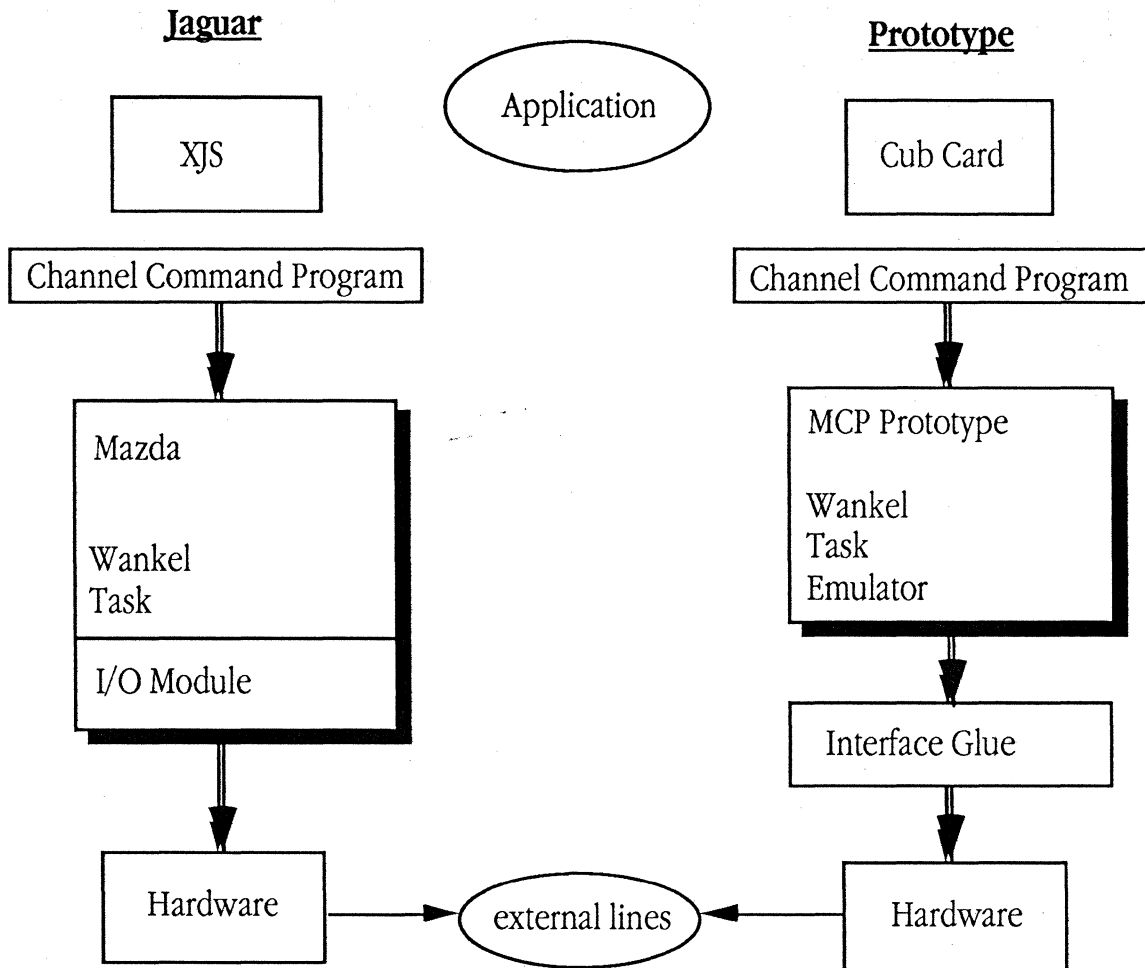
- Initial Prototyping,
- First Pass Software,
- Environment Reenhancement,
- Final Software, and
- Hardware Port.

This section places these activities within the overall Jaguar Software Project framework. Since all JaNeT subsystems will be developed sharing a common strategy, that is outlined below.

---

### Prototyping Diagram

The following is a diagram comparing the software environment constructed by JIO with the final environment as it will exist on Jaguar.



The XJS CPUS functionality will be prototyped using the two 88100 microprocessors on the Cub card. The wankel I/O processor will be simulated using the 68K found on the MCP card. An MCP card will be developed to design, prototype, and test the target Jaguar silicon for ralph and brian. The gussy serial subsytem will be emulated using a standard Livonia serial card. The FriendlyNet system will probably be developed directly on the prototype XJS motherboard, since the Ethernet transceiver silicon will not be available until summer 1991.

**Proof-of-Concept**

The "Proof-of-Concept" is the first milestone in the Jaguar Software timeline. It is designed to be a first evolution of the Design Document. This milestone is expected to identify the risky and least understood components of Jaguar software as given in the Design Document.

The current date for the Proof-of-Concept Milestone is October 15, 1990.

Two stages of JIO occur within the Proof-of-Concept Milestone : Initial Prototyping and First Pass Software.

**JIO Stage 1 : Initial Prototyping**

The Initial Prototyping stage will bring together the hardware necessary to provide a prototype for Jaguar hardware.



For the Jaguar CPU, this consists of the Cub Card (a NuBus card with 2 *standard* 88K processors) running a simplistic operating system which generates A/ROSE messages. For the I/O subsystems, this will be an MCP card with the controller chips and hardware interface glue used in the specific Jaguar I/O subsystem.

Support hardware, such as external communication lines and a test/development lab, will be obtained and organized concurrently.

## **JIO Stage 2 : First Pass Software Development**

The JIO First Pass Software stage is a first attempt at software which approximates the Jaguar software functionality as described in the design document.

This software will run on the hardware prototype platform assembled in the Initial Prototyping stage of JIO.

Specifically, a Wankel Task Simulator (WATUSI) for the specific task will be written, in C. This task will take channel command programs and simulate the actions of these CCPs (as executed by the final Wankel task - I/O Module combination on Jaguar) on the prototype hardware.

Software will need to be written which will serve to generate channel command programs for the WATUSI. This software will run on the 88K Cub Card platform.

A preliminary format for the channel command words dealing with the specific I/O task will be determined in this stage.

---

## **Software ERS**

The Proof-of-Concept milestone of the Jaguar Software timeline is complete when decisions have been made about the software functionality which will be provided and (possibly) the specific software components and interfaces which will define Jaguar Software.

At this point, the Software ERS can be written. The Software ERS is a specific description of the Jaguar Software components, including functionality, classes and interfaces. This ERS will be distributed to others both inside and outside Jaguar in order to get feedback on the provided interfaces and software components.

Work on prototyping and design decisions can continue throughout the writing of the Software ERS, but there must be a (temporarily) frozen version of the Software which correlates to the description given in the ERS.

The Software ERS milestone is currently set at December 15, 1990. (Happy Holidays).

---

## **Prototype #2**

Once the Software ERS is complete, feedback has been (gracefully) accepted and evaluated, and the holidays are over (and everyone is 10 pounds heavier), work can begin on the Prototype #2 Milestone of the Jag Software Timeline.

Prototype #2 Milestone is designed to get the Software as close to the final hardware stage as possible. By the end of Prototype #2, we want to have as much "true" software up and running as possible in an emulated environment.

Heavy use will be made of emulators, simulators, assemblers, stimulators, and perambulators.

Prototype #2 Milestone is currently set for April 15, 1991.

Two stages of JIO fall within Prototype #2 : Environment Reenhancement and Final Software Development

### **JIO Stage 3 : Environment Reenhancement**

The Environment Reenhancement stage is designed to create an environment which mimics the true Jaguar environment as closely as possible.

This begins the first stage of the port to the target hardware software. A Wankel emulator (WANKEM), hopefully available at or near this point (or at the local emulator store), will be running on the MCP prototype (from JIO stage 1: Initial Prototyping). I/O Module simulators (or actual hardware) will also be created for each specific task and placed between the Wankel emulator and the specific hardware. This creates a working environment for the next stage of development.

### **JIO Stage 4 : Final Software Development**

Once the Jaguar Hardware environment has been faithfully emulated to a high degree, work can begin on development of the final software package which will be the Jaguar software.

A Wankel task (YEFF - Yoke to Emulate the Final Frontier, sub whatever the specific task is) will be written to run on the Wankel emulator on the MCP card with the proper I/O Module. This setup will then be exercised to iron out all bugs.

---

### **Hardware Port**

The current Jaguar Hardware delivery date is around spring 1991.

Once hardware is delivered, work can begin on porting the Jaguar Software (as developed on the emulators) to the production hardware.

The Hardware Port milestone occurs when all Jaguar software has been ported from the emulated environment to the Jaguar Hardware.

The deadline for the Hardware Port Milestone is August 15, 1991.

The Hardware Port Milestone comprises the final JIO Stage : Port to Final Hardware.

### **JIO Stage 5 : Port To Final Hardware**

When the XJS/Mazda hardware and full Wankel software is available, the Wankel tasks and hardware will be ported to the final platform. (This should take, oh, 30 minutes?)

If there is preliminary hardware available earlier, then test ports can be done of the Wankel task. This is why a Wankel emulator should be one of the earliest things developed. It allows us to write working Wankel code ASAP and also provides a means of helping debug early versions of the hardware.

---

## Alpha Milestone

Alpha Milestone is the date at which all functionality promised in the Software ERS (with subsequent revisions and Tech Notes) exists on the Jaguar hardware platform.

The current Date for Alpha Milestone is November 15, 1991.

---

## Beta Milestone

Beta Milestone is the date at which the software functionality and interface has been completely frozen. Most major bugs have been ironed out and the software runs almost completely.

The current date for Beta Milestone is February 15, 1992.

---

## Final Milestone

Final Milestone is the date at which all bugs have been ironed out of the software and the software works completely.

The current date for Final Milestone is sometime before May 15, 1992.

---

## Golden Master Milestone

Golden Master Milestone is when the beast is frozen for manufacturing and subsequent shipping to customers.

The current date for Golden Master Milestone is May 15, 1992.

---

---

## RALPH: The Real-time Analog PHONE

This section describes ralph rationale, features, and implementation. First we discuss the hardware and software components of the system architecture. The events behind a FAX transmission are traced as a tutorial. Next the RALPH data transmission (modem) facilities are described. This is followed by an introduction to the RALPH call management features. Finally we touch upon the real-time OS aspects of this subsystem implementation.

---

## RALPH System Architecture

RALPH provides a high performance, integrated connection to the world's analog phone system. RALPH is an intelligent, integrated replacement for the usual assortment of datacom "belts and braces" required to communicate over analog phone channels: those (hard to find and easier to break) serial cables, wall bricks, and modems. RALPH consists of the datalink and physical level protocols that make communications *happen*. That is to say: This interface supports the following functions:

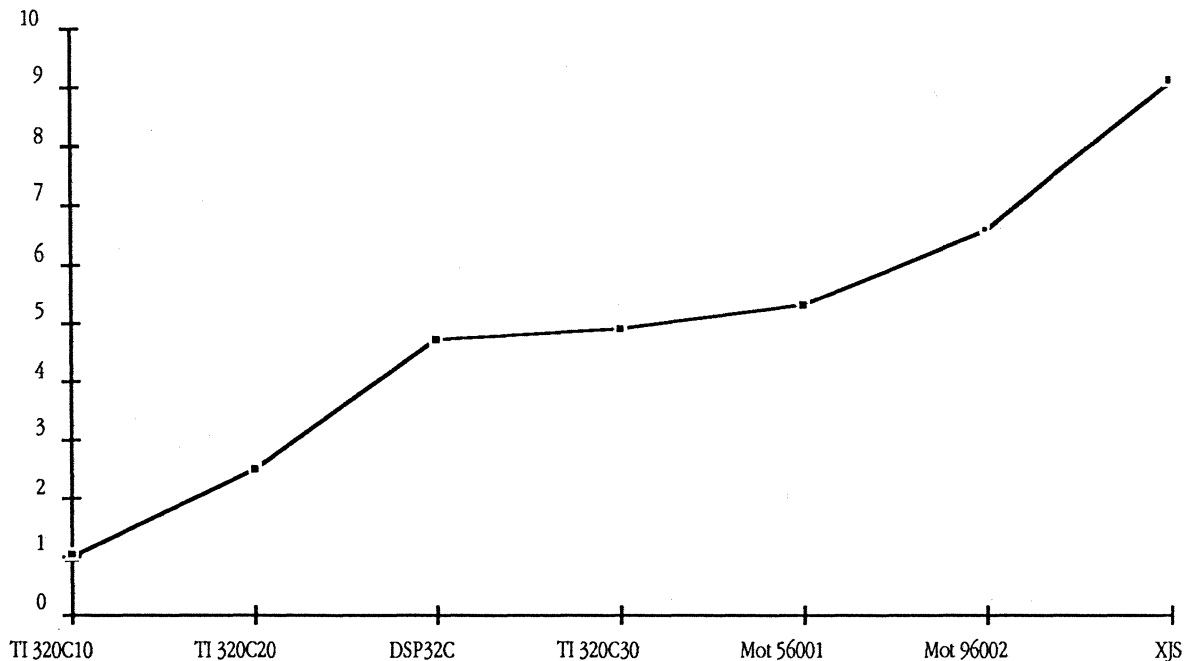
- Telephone-quality (300-3.4KHz voice grade) speech input/output using standard desk phone or hi-fi I/O channels.
- Modem emulation in software, initially including 300/1200/2400, v.27ter (2400/4800 bps), v.29 FAX (7200/9600 bps), and later, the v.32 standards.
- Facsimile transmission and reception using CCITT recommendations T.30 and T.4 (group 3).

- Call progress monitoring, including DTMF encode/decode support, incoming call and busy signal detection,

RALPH consists of the following components:

- the analog phone access manager, providing data transmission and call-progress monitoring for clients such as the FAX manager, serial drivers;
- user-selectable data compression and error-correction filters to ensure a reliable datalink level service. These are inserted into the data stream to provide enhanced data transmission capability and compatability with existing modems and software packages.
- modem data pump modules. These software modules emulate the functions traditionally associated with a standalone modem or proprietary silicon.
- call progress monitoring, which are those functions required to establish and maintain a connection on a dialup line. This includes generation and detection of DTMF (TouchTone™) signals and detection of off-hook, busy signal, incoming call events.
- a WANKEL control program that routes RALPH sample buffers between the RALPH modem and call progress tasks and the analog interface subsystem.

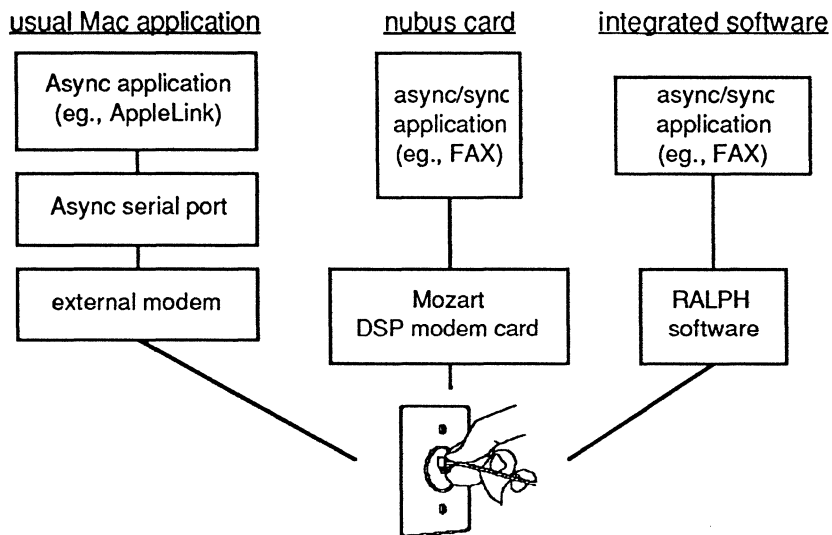
RALPH takes advantage of Jaguar's exceptional processing power, which permits traditional DSP algorithms to be implemented in real-time without a dedicated DSP processor. The chart below shows the relative power of a single 50 MHz XJS processor, relative to specialized DSP hardware<sup>1</sup>. Note that no premium is placed on floating point capability, which is not important for comm work. Also, the XJS figure does not include operating system overhead.



<sup>1</sup>EDN DSP benchmarks,, 29 September 1988.

Several advantages result from using the central processor(s) for signal processing instead of a dedicated DSP chip. The primary advantage is that common operating system, languages, and development tools are used on an integrated architecture; no special caching, memory busses, RAM, or languages are required. Second, processing power is extensible. Additional processors will not just boost the capabilities of the DSP subsystem; since the DSP is integrated, all applications will benefit from the additional processing power. The result of this integration is easier software development, flexibility, and an economical design.

From the application's point of view, RALPH should be invisible. Traditional phone communications occur with an external modem, or plug-in card. RALPH offers performance including datalink compression and error correction - that meets or exceeds "premium" modems. The diagram below shows three possible alternatives to connect an application with its remote peer over a phone line. With RALPH it's a snap.

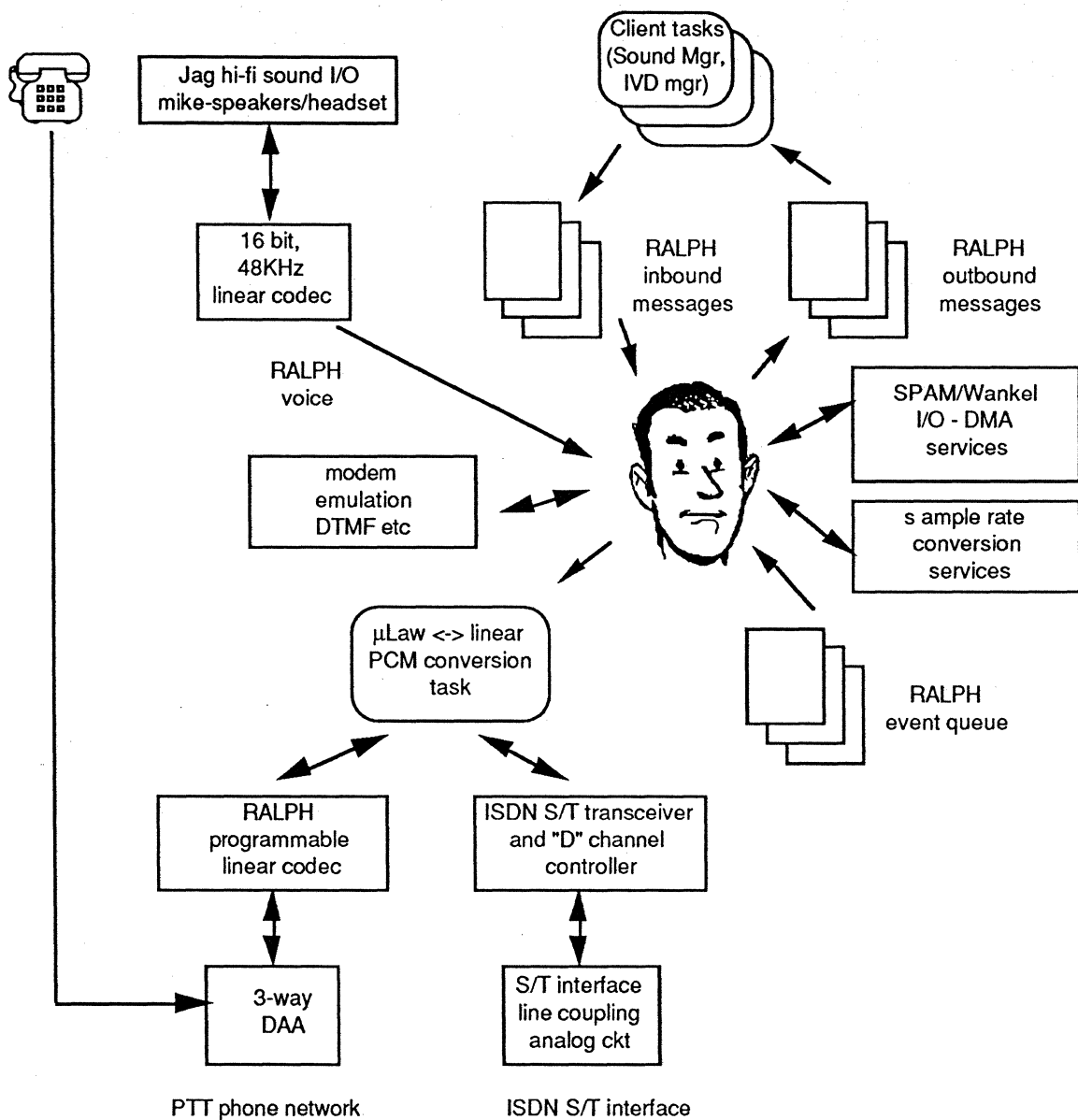


Let's now take a look at the software and hardware components of the RALPH subsystem.

## Software

### RALPH Access Manager

The diagram below shows the relationship of RALPH with Jaguar hardware and system software.



RALPH is a telecom-oriented, low-level application team that provides voice and data facilities to clients via its message-oriented interface. Higher level services - such as the Communications Toolbox or C++ objects - will be built on top of this foundation for direct exploitation by application programmers. Real-time scheduling requirements are severe, as buffer underruns occurring while acting as a modem pseudo-device can cause loss of synchronization. RALPH uses SPAM - the signal processing access manager's - to route signal buffers between the Jaguar CPU and physical interface.

RALPH maintains a queue of telecom events and passes these on to its client. These events include:

- modem and telephone call progress signals, such as carrier detect/drop or phone hook status
- Ring Detect

- DMTF sequence received
- full duplex modem data stream data

RALPH is responsible for the following tasks:

- routing digitized voice traffic to and from the human interface and ISDN or analog phone line.
- software modem emulation. RALPH encodes the input bit stream into modulated signal and decodes received modulation into a bit stream. This stream could be a redirected the serial driver I/O stream or a directly callable user interface.
- PCM encoded data conversion between linear and  $\mu$ -law or A-law companding (the latter being used outside the US and Japan)

Typical client applications would include:

- FAX bitmap encode/decode.
- file transfer protocols
- data compression, error correction, or encryption protocols

These services will be provided by higher level FAX manager, and connectivity tools that will be developed for JaNeT. JaNeT will enforce an exclusive access policy on RALPH resources when appropriate.

RALPH clients - notably the Integrated Voice Data Manager and the FAX manager - access RALPH services using conventions established by the pink I/O Name Server and Access Manager. The IVD "hides" implementation-specific details of the communications path from the application software. JaNeT in turn hides the physical implementation details from its clients. For example, it would be transparent to an application if a "real" external modem were connected to the serial port and substituted for a RALPH pseudo-device modem.

The Communications ToolBox and Integrated Voice Data Manager represent a procedural interface "shell" that attempts to hide implementation details from the application programmer. These shells will be JaNeT's primary clients. Pink has introduced the idea of an object-oriented Sound Manager and Sound Server; these too would be primary RALPH clients. Among the objects currently proposed by the Sound Manager are a generic telephone line and modem. The Sound Manager services such as voicemail recording/playback will be built on JaNeT services.

#### Example: FAX transmission

Let's take a specific example of how RALPH would be used in a practical situation: sending a Jaguar document as a standard FAX document to a remote station. The three components that comprise this task are the operating system, the FAX manager, and RALPH.

Refer to the diagram on the following page. The transmission command might have been initiated by a verbal request made to the Digital Assistant, an icon dragged to some iconic destination, or execution of a command chit. In any case, this would result in the following events:

- the O.S. determines via the Retriever that the target destination corresponds to a "remote volume": in particular, a Group 3 FAX receiver at a certain telephone number. This information is passed to the FAX manager, part of the Communications Toolbox.

- The FAX manager, using RALPH dialing/call management services, places a call to the remote station. The answering station sends RALPH an analog modulated handshake sequence identifying the FAX session parameters. RALPH passes the demodulated, digital data stream up to the FAX protocol engine, which proceeds to establish the FAX session with the remote site. Session parameters such as image resolution and transmission speed are exchanged.
- Once the session parameters are exchanged, the imaging system is directed by the FAX manager to generate the document's bitmap. The bitmap resolution might be 100v x 200h (group 3 coarse) to 200v x 200h (group 3 fine) to 400v x 400h (ISDN group 4)<sup>1</sup>.
- The resulting bitmap is compressed by the FAX manager using its T.4 compression task. The precise algorithm used depends upon the current FAX session parameters.
- The FAX manager then proceeds to transmit the compressed bitmap using the T.30<sup>2</sup> protocol engine task. First, the protocol machine passes the compressed image data to the HDLC framing task for encoding to ensure reliable transmission. The resulting synchronous bit stream is passed to ralph for transmission over the phone line.
- RALPH converts the synchronous bit stream into a digital sample buffer representing the analog V.29 modem signal. RALPH passes this sample buffer to the WANKEL I/O controller, which transmits it on the phone line via the Jaguar analog subsystem. The sample buffers are multi-buffered to minimize latency problems. WANKEL generates a signal to awaken RALPH when the buffer has been transmitted.

---

<sup>1</sup>Currently available Mac FAX products image the bitmap through a chooser-installed "print driver". Surely we can be more flexible using a dynamic imaging scheme.

<sup>2</sup>CCITT standards that define image transmission over group-3 fax terminals.



Operating System

Digital Assistant

Verbal command, dragging icon, or other action generates "send FAX" command

Imaging system (Albert)

The O.S. imaging subsystem stands by to prepare a FAX-compatible bitmap. The resolution and size is a function of FAX standard used; typically 200 dpi. ISDN FAX (group 4) can go to 400 dpi.

FAX manager

T.4 compression task

The T.4 compression task translates the imaged bitmap into a compressed bitmap using parameters specified by the T.30 state machine for this session.

T.30 FAX protocol state machine

T.30 protocol engine determines the imaging parameters that are given to the O.S. imaging system, which builds a T.4-format document bitmap.

HDLC framing task

The HDLC framing task encodes/decodes FAX message unit as checksummed HDLC frames. The T.30 protocol can thus maintain an error-free link. This task is performed by the SCC chip looping through the ISDN controller.

Real-time AnaLog PHone



RALPH modem task

RALPH Modulates and DEModulates the analog phone signal according to CCITT specs (V.29, V.27ter and V.21 for FAX)

Wankel DMA engine

The Wankel I/O controller exchanges RALPH sample buffers with the RALPH modem task on a synchronous interrupt basis

A/D - D/A subsystem

The A/D-D/A subsystem includes the hardware necessary to couple the phone line to Jaguar. This hardware is configured and monitored by RALPH.

Jaguar FAX transmission components

The hierarchical structure of JaNeT ensures that low-level protocol implementation differences are hidden from the upper layers. For example, the procedure described above would be certainly be identical from the user's perspective whether the destination was ISDN or POTS: the only difference being the potentially higher image resolution and near-instantaneous transmission speed provided by the ISDN connection. The operating system accesses both "remote volumes" through the Communications Toolkit; the imaging software works the same, only the bitmap parameters are different; the FAX manager hides link establishment details from its client; and RALPH emulates the modem standard negotiated for the session.

Now suppose that the destination mapped to an ISDN basic rate address. The operation would be similar to the outline above, except that the Communications Toolbox would invoke the ISDN fax manager to establish an ISDN call and transmit the FAX document using BRIAN, the basic rate access manager.

### Wankel RALPH task

The WANKEL RALPH task routes samples between the codec and DMA buffers. Sample buffers are synchronously generated by RALPH. The buffer size is constrained by processor scheduling efficiency on the small side and user latency on the other. Ideally, the buffers would be very large (>1000 samples), but this would result in a latency of some 1/7th of a second - almost 1/3rd second round trip - for interactive RALPH clients such as remote terminal services.

Buffer size is also determined by the sampling rate. The sample rate used by ralph modem applications will certainly be 7,200 samples/sec. This is dictated by the integral number of samples per modem symbol (typically 600, 1800, or 2400 symbols/sec). However, the additional bandwidth obtained by using a higher sample rate - 9,600 samples/sec is an option with some chips - could prove helpful in improving deskset voice recognition accuracy. The WANKEL configuration control word selects the sample rate (7,200/9,600) and buffer size.

## Hardware

An important part of ralph's job is to control Jaguar's telecom hardware. This subsystem places critical real time scheduling demands on the scheduler. Thus, although implementation details are rightfully hidden from the client, a brief overview of the controlled hardware may be of interest.

The ralph telephone interface consists of three parts: an analog/digital conversion stage; an automatic gain control system; and an electrical telephone network interface.

### Analog Interface System

The Analog Interface System (AIS) comprises analog-to-digital and digital-to-analog convertors. The AIS is designed to provide high quality signal translation for voice and modem applications, while minimizing the processing requirements for the Jaguar DSP software.

The Jaguar AIS will use two sampling rates: 7.2 KHz for modem emulation, and 8 KHz for PCM voice. The 7.2 KHz rate is optimized for signal processing, as it is the lowest rate that both satisfies the Nyquist

sampling requirement *and* is integral multiple of the modem symbol rates. This minimum sample rate reduces processing overhead while simplifying algorithms. The 48 KHz hi-fi sample rate is precisely six times the 8 KHz rate, resulting in higher quality sample rate conversion with less processing as well.

A sampling rate of 9.6 KHz would provide additional bandwidth (compared to 8 KHz) that could be useful for voice recognition work. This rate would have to be downsampled to 8 KHz for ISDN. More analysis is needed to determine the potential advantages of 9.6 KHz over the 8 KHz rate.

### Automatic Gain Control

The analog signal's amplitude at the A/D converter must be properly scaled for optimal A/D conversion. The input signal from the remote transmitter has a possible dynamic range of 0 dBm to -46 dBm. However, in full-duplex operation the transmit signal is looped back to the receive input through the hybrid with perhaps 10 dB attenuation, resulting in a minimum lower ceiling of around -19 dBm. If the analog signal level is too small, dynamic range is lost; too high, and serious distortion will result. The signal is scaled by the automatic gain control (AGC) circuit, which consists of a programmable hardware attenuator and RALPH controlling software. RALPH AGC control software monitors the RMS power within the spectrum of interest by scanning the post-conversion DMA buffers. After calculating the signal power, RALPH adjusts the programmable analog gain stage. This process is iterated during the handshaking phase (with faster tracking to achieve a suitable scaling quickly) and then periodically during steady-state operation.

The required dynamic range is a function of the codec sensitivity and resolution. Since Jaguar provides floating-point processing, post-conversion normalization is not required except at the symbol demodulation operation.

### Data Access Arrangement (DAA)

The Data Access Arrangement (DAA) module provides a standardized interface to the various international PTT phone network interconnection standards. The DAA performs the following hardware functions:

- Couples the separate transmit and receive analog data paths between the codec to the composite (two-wire) subscriber telephone loop<sup>1</sup>
- Signals "ring indication"
- Allows Jaguar to take the phone line "off-hook"
- Provides current to drive the desk set
- Provides electrical isolation and transient protection.

The DAA features a versatile interconnect scheme that provides separate connections to a standard 2500 desk set and to the telephone line. This allows the desk set to be used for both direct voice I/O in addition its conventional function. The following configurations are software-selectable:

- desk set directly connected to phone network; Jaguar out of loop

---

<sup>1</sup>the hybrid four-wire to two-wire will be based on the motherboard.

- desk set directly connected to Jaguar; phone network out of loop
- desk set and Jaguar connected to phone network in parallel

When the Jaguar power is off, the default configuration connects the desk set to the phone network: the Jaguar is entirely removed from the circuit. Loop current is supplied by the phone company as usual.

When powered up, Jaguar selects one of the three configurations shown above via the RALPH interface. The desk set audio paths are routed to a standard  $\mu$ -law 8 KHz codec, the telephone line (modem) codec, and the high fidelity 48 KHz codec. System software can choose between these sources depending upon application requirements. For example, voice annotation would benefit from the reduced data rate produced by the 8 KHz codec, while best-quality speech recognition would be achieved with the higher sampling rate coming from the high fidelity codec.

Dialling may be performed by Jaguar or the phone handset. A separate ring detect circuit allows incoming calls to be handled while the desk set is being used for voice I/O.

## Buffer management

Buffer management will be provided by SPAM. N-way buffering will be implemented to increase scheduler elasticity, thus minimizing the probability of incorrect operation from scheduling problems.

## Real-time processing issues

RALPH modem emulation routines are critically dependent upon obtaining sufficient processing time on a synchronous basis. Processing epochs will typically be around 10 ms (72 samples at 7200 samples/second). This represents 96 bits of information at 9600 bps (FAX/v.32). The lower bound is predicated by OS task switching overhead; the upper bound is primarily determined by maximum latency that's visible to the user and to a lesser extent by the AGC update granularity.

The Jaguar receiver sample buffer will overrun, and the transmit sample buffer will underrun, if insufficient processing cycles are received in an frame. This will result in garbage being transmitted on the line; the result could range from simple (and probably, recoverable) bit error(s) to loss of synchronization on the line which could result in communications collapse. The Jaguar receiver would attempt to demodulate that old sample buffer, resulting in a stream of garbage being passed to the RALPH client.

According to the Jaguar real-time scheduling module<sup>1</sup>, the RALPH modem emulation tasks fall into the category of "super-critical tasks": those which cannot tolerate degradation<sup>2</sup>. Fortunately, the CPU overhead can be computed with high accuracy either *a priori* or by using dynamic "frame" timing. The CPU load is almost constant and has a very small standard deviation.

---

<sup>1</sup>R Williams, "Results of real-time scheduling simulations v2.0", 3 March 1990

<sup>2</sup>Actually, some modems have a "fallback" (slower) speed, but the CPU savings are slight due to fixed overhead in the DSP algorithms.

---

## RALPH modem (data pump) services

V.32, the 9600 bps, full-duplex standard using trellis encoded<sup>1</sup> QAM, may be implemented on first release, subject to vendor negotiations. This standard is extremely compute intensive: an estimated 50% of the XJS processor would be required to emulate this mode.

Synchronous data streams using HDLC framed data are supported by routing the HDLC data in a loopback mode between the SCC and Jaguar's ISDN controller. See the ISDN section for more detail.

Higher level protocol processing, such as MNP compression/error correction will be performed as a normal XJS-based task. The V.42/V.42 bis data compression standards will probably replace MNP in the 90's, so this standard will be supported as well.

In short, using intrinsic RALPH data pump, all Jaguars will have the capacity to communicate worldwide with any 300-9600 bps modem, with integral support of group 3 (digital compression) FAX.

## data transmission facilities

### 300/1200/2400 bps modem emulation

The vast majority of current personal computer applications fall in this category. Jaguar will be able to communicate with the large international installed base of low and moderate speed modems that use the following standards:

<u>type</u>	<u>speed</u>	<u>full/half duplex</u>	<u>modulation</u>	<u>where used</u>
103	300	full	frequency	USA
v.21	300	full	frequency	international
v.23	1200/75	half <sup>2</sup>	frequency	international
212	1200	full	phase	USA
v.22	1200	full	phase	international
v.22bis	2400	full	phase-amplitude	universal

### V.32 modem emulation

V.32 is a full-duplex, 9600 bps modem standard. It is primarily used as a backup for leased line applications, but has found growing interest as a basis for high speed PC communications. V.32 is by

---

<sup>1</sup>a sophisticated encoding technique that improves modem performance on marginal lines.

<sup>2</sup>v.23 has a low speed "back channel" designed for keyboard input found in videotext applications, eg Minitel

far the most complex standard to implement, and computationally intensive as well: an estimated 50% of an XJS processor would be consumed.

Although V.32's full-duplex performance potential is very good, there are three points that should be noted:

- V.32 requires very good quality lines to achieve the 9,600 bps rate reliably. A V.32 modem will often "fall back" to 4,800 or 2,400 bps on marginal lines - particularly in Europe. Further, the widely-used 32 Kbps ADPCM encoding standard was designed to accomodate only 4800 bps modem traffic; only three ADPCM conversions can be supported by V.32 (according to ATT's spec).
- Full-duplex operation requires computationally intensive echo cancellation software. The estimated processor loading (50% of an XJS) is quite high relative to the transmission rate.
- The V.32 installed base is relatively low compared to FAX and 300-2400 modems.

Work will proceed on V.32 following successful implementation of the V.29 FAX protocol.

### Data compression

The RALPH system will have data compression facilities as a user-selected feature. The data compression algorithm can be dynamically installed. The algorithms provided by Apple will be MNP 5, and V.42bis. This software has been purchased from a developer.

The MNP 5 (Microcom network protocol) compression is public domain yet not as effective as Lempel-Ziv. V.42bis is a new standard that's based on LZW; developed by Hayes and British Telecom, it's essentially a low-cost implementation of the LZW algorithm.

Data compression is an especially valuable function from a computational viewpoint. Software running on the Mac II emulates a v.22bis (2400 bps) receiver at 18 bytes/sec, while LZW compression runs at 17K bytes/sec. Assuming that the compression factor is 2:1, the test case showed that 58 ms seconds of compression processing saved 27 seconds of v.22bis DSP processing. This is equivalent to a "power multiplier" of 465:1!

### Error correction

An error correcting protocol filter can be optionally placed in the RALPH data stream. The de facto standard is currently MNP 5. V.42bis may also be supported.

### **image transmission facilities (fax unit)**

V.29 is a 9600 bps, half-duplex standard using QAM modulation. Fallback rates at 7200 and 4800 bps<sup>1</sup> are also specified for use on impaired lines. This standard is universally followed for FAX transmission. v.27ter, the fallback speed used on marginal quality lines, is a 2400/4800 bps, half-

---

<sup>1</sup>4800 bps V.29 uses the V.27ter standard

duplex standard using phase modulation. This is the low-speed standard for FAX transmission, and is used in Europe for dialup X.25 packet network service as well.

<u>type</u>	<u>speed</u>	<u>full/half duplex</u>	<u>modulation</u>	<u>where used</u>
v.27ter	2400/4800	half	phase	universal
v.29	9600	half	phase-amplitude	universal

FAX transmission requires more than the data pump; image compression and call-establishment handshaking software is required as well<sup>1</sup>.

Let's take the example of an application that wants to transmit a fax image. This entails the following sequence of events:

- Application calls Integrated Voice-Data Manager to establish connection with remote station. Note that the IVD hides the details of the remote station's characteristics: it could be an ISDN terminal, external modem connected on the async port, or V.29 group 3 FAX
- The IVD configures RALPH by issuing the DoCommand() procedure. This translates into a message containing an ASCII-encoded string of commands that could include going offhook, DTMF tone generation, selection of modem pseudo-device to emulate.
- RALPH asynchronously returns a message to the IVD containing status reflecting the outcome of the associated request. In this case it could be "connected", "not connected", "busy", or "modem pseudo-device in use".
- application can now proceed to transmit encoded image over the modem pseudo-device.

#### T.4 image compression service

This is the CCITT standard for image compression on the universal "group 3" FAX machines. This package has been purchased from an east coast firm by Apple Paris R&D.

#### T.30 FAX handshaking protocol

The T.30 handshaking protocol defines the procedures followed to initiate and maintain an image transfer session. This involves machine identification, imaging parameter selection, transmission speed negotiation, and so on. It too has been purchased by Apple Paris R&D from an east coast firm.

#### v.29 - v.27ter data pump

V.27ter a 2400/4800 bps, half-duplex standard using PSK modulation. This is the low-speed standard for FAX transmission, and is used in Europe for dialup X.25 packet network service as well.

---

<sup>1</sup>This software will be adapted from ongoing development work in the Paris R&D labs

V.29 a 9600 bps, half-duplex standard using QAM modulation. Fallback rates at 7200 and 4800 bps<sup>1</sup> are also specified for use on impaired lines. This standard is universally followed for FAX transmission.

V.32, the 9600 bps, full-duplex standard using trellis encoded<sup>2</sup> QAM, may be implemented on first release, subject to vendor negotiations. This standard is extremely compute intensive: an estimated 50% of the XJS processor would be required to emulate this mode.

Synchronous data streams using HDLC framed data are supported by routing the HDLC data in a loopback mode between the SCC and Jaguar's ISDN controller. See the ISDN section for more detail.

Higher level protocol processing, such as MNP compression/error correction will be performed as a normal XJS-based task. The V.42/V.42 bis data compression standards will probably replace MNP in the 90's, so this standard will be supported as well.

In short, using intrinsic RALPH data pump, all Jaguars will have the capacity to communicate worldwide with any 300-9600 bps modem, with integral support of group 3 (digital compression) FAX.

#### interaction with HDLC framing task

Certain portions of the fax image transmission protocol require HDLC framed data. This will be obtained by using the brian HDLC framing facilities. wankel passes the data to be framed through the ISDN-dedicated SCC, which performs the framing and routes the data back to the processor memory space via the ISDN controller chip.

---

## **Call management facilities**

Besides the data transmission facilities, RALPH will provide additional functionality to support data and voice messaging over the dialup telephone network.

### **call progress monitoring**

RALPH will signal via events the presence of ringing signals, busy signals, DTMF tones, and hook status. The RALPH line snoop task will be launched when the line is taken off-hook, or an incoming ring has been detected. The snoop task "listens" to the telephone line and detects the presence of DTMF tones, busy/ring signals, or incoming modem signals. Note that detection of the incoming ring is performed by the I/O hardware.

### **dialling support**

RALPH can generate both DTMF and pulse dialling. Pulse dialling is achieved by rapidly switching the DAA's off-hook relay. The make/break ratio which varies between countries - is defined in the RALPH configuration record.

This service will be used by autodiallers, voice messaging systems, etc.

---

<sup>1</sup>4800 bps V.29 uses the V.27ter standard

<sup>2</sup>a sophisticated encoding technique that improves modem performance on marginal lines.



## person/machine answer sequence

RALPH must be able to detect whether the other party is a human or computer, and act accordingly. It may not be known *a priori* whether the caller is voice or modem. There are two alternatives in this case:

- RALPH is instructed to answer the phone assuming a specific mode of operation. This would mean using a predefined modem protocol (eg. FAX) or silence in the case of anticipated voice calls).
- RALPH uses a pre-defined sequence of modem initiation sequences to try and establish communications with the (presumed) originating modem. Note that modem standards provide for "graceful" determination of calling modem type.

Neither of these two alternatives permit intelligent switching between the domain of voice and data. RALPH would lock Jaguar into a specific communications session (voice or data) without the possibility of dynamic switching. It would be much better if RALPH could exercise some intelligence in "answering the phone".

One method providing more flexibility would work as follows. On answering the phone, RALPH would play a prerecorded message that's stored on disk. RALPH would then "listen" to the line for a short period of time (say three seconds), expecting a response. The response could be voice energy (the caller's voice) or a DTMF sequence: DTMF signalling would be simplest and least ambiguous method of routing the call. At its simplest, any DTMF tone response could be interpreted as "caller wants to leave a voice message". More sophisticated users could create a more elaborate parser using the by-now familiar "tree walk" paradigm: "press 1 to ring user, press 2 to leave voicemail, press 3 to consult mr pink...". The eventual (and I believe achievable) goal is to permit relatively structured, yet extremely powerful voice I/O interaction with the machine via Jaguar's Digital Assistant.

If Jaguar's message is received with silence, Jaguar knows that the caller is must be a modem in originate mode (or crank call!). In either case Jaguar will respond with a modem training sequence according to the pre-defined sequence mentioned above. This technique will work because the Jaguar voice message playback should not be "audible" to a calling modem; it's expecting the answering modem's audio handshake sequence which is a very structured signal. Further, the originating modem always remains silent until the answering modem's sequence is recognized, so even simple measure of audio energy in the voice channel would suffice for differentiating a human from a modem.

Note that the CCITT FAX procedures specify a "CNG" (calling) tone that alerts the answering party that the incoming call is a FAX machine. RALPH will recognize this signal and route the incoming data to the FAX manager<sup>1</sup> for the user's convenience.

The Jaguar voice daemon will provide truly useful remote access capabilities. Imagine, one's machine can be given a recognizable "personality" through acquired recognition skills, vocabulary, speaking (output) accent, behavior...

## voice messaging

The PCM data stream coming from RALPH can be optionally compressed and stored to disk for playback, editing, and archival purposes. This will be handled by the Sound Manager.

---

<sup>1</sup>A FAX tone detector currently sells for about \$150 alone. Another example of built-in value.

## interaction with linear/mu-law conversion

The telephone signal is sampled using a linear 16-bit codec. RALPH can optionally convert this data to 8-bit  $\mu$ -law companded data to reduce storage; this is required for ISDN voice calls.

## voice compression

Voice compression services using subband or CELP coding will be provided to minimize storage and transmission expense.

---

## DSP toolbox

The DSP toolbox is a collection of signal processing algorithms that are highly optimized for maximum efficiency on the XJS architecture. They will be heavily used by the RALPH subsystem. Each routine maintains a context for the caller that preserves the state of the continuous-time variables that are modified by the routines.

### Routines

IIR(): performs direct-form implementatino of infinite impulse response (recursive) filters. Input data, coefficients, and filter length are passed to the routine.

FIR () performs highly optimized symmetrical filtering using finite-impulse response, linear phase filters. Input data, coefficients, and filter length are passed to the routine.

FastFIR(): performs filtering using the efficient fast convolution approach, which reduces to multiplication in the frequency domain. This routine provides substantial computational efficiencies when large amounts of data need to be filtered.

FFT(): discrete Fourier transform and its inverse. Data set size, forward/inverse flag, and real/complex data flag are the parameters.

Power(): calculates the rms power in the passed sample buffer.

LMS\_EQO: performs least-mean squared error equalization based on error signal, coefficients, and input data.

DTMF\_receive(): evaluates passed data for presence of DTMF energy.

DTMF\_generate(): fills buffer wil DTMF signal.

---

## RALPH standby power operation

RALPH must be able to monitor the phone line while the machine is in the standby state. The system power is always applied to the I/O circuitry when the machine is plugged in, allowing the system to detect a ring indication signal from the DAA via the mazda chip. A message must be passed to the system on bootup (analogous to finder info on a Mac program launch?), directing the system to auto-launch RALPH with the receptionist software running.

## power down/up issues

How long does it take for the Jaguar to boot up? The normal ringing cadence in the U.S. is six seconds per ring; assuming a "normal" wait of four rings before hangup, we have about 25 seconds to bring the machine up from the quiescent state.

---

## RALPH System Interfaces

### RALPH configuration profile structure

RALPH maintains a structure that defines the current operational mode and status. This structure can be dynamically accessed and set through corresponding procedure calls. Note that several of the operational parameters directly correspond to "AT" command set registers.

```

struct {
    UCHAR  emulationMode;      /* what type of modem */
    UCHAR  originateMode;     /* 0 = answer, 1 = originate */
    INT    bps;                /* transmission rate */
    BOOL   onHook;            /* phone DAA is on/off hook? */
    BOOL   busy;              /* called party is busy */
    BOOL   RTS;               /* caller's ready to send */
    BOOL   CTS;               /* ralph data pump ready to send */
    INT    operatingStatus;    /* state of ralph's data pump */
    INT    voiceSrc;           /* source of voice data */
    INT    voiceDst;          /* destination of voice data */
    INT    breakRatio;        /* make/break ratio */
    INT    dtmfTonesHigh[16]; /* dtmf high tone pair array */
    INT    dtmfTonesLow[16];  /* dtmf low tone pair array */

    union {

    struct {
        UCHAR  ringsBeforeAnswer; /* 0 */
        UCHAR  countRings;        /* 0 */
        UCHAR  escapeCodeChar;    /* '+' */
        UCHAR  crChar;            /* 0x13 */
        UCHAR  lfChar;            /* 0x10 */
        UCHAR  bsChar;            /* 0x08 */
        UCHAR  blindWaitTime;     /* 2 (secs) */
        UCHAR  dialToneWaitTime;  /* 30 (secs) */
        UCHAR  pauseTime;         /* 2 (secs) for comma */
        UCHAR  cdResponseTime;    /* 6 (1/10th sec) carrier detect wait*/
        UCHAR  hangupDelay;       /* 14 (1/10th sec) no carrier to hangup */
        UCHAR  dtmfDuration;      /* 95 (ms) DTMF tones duration/spacing */
    } ATregs;

    struct {
        UCHAR  s0, s1, s2, s3, s4, s5, s6, s7, s8;
        UCHAR  s9, s10, s11, s12, s26;
    } Sregs;

    } regs;
}

```

## Events

RALPH generates events as a result of normal operation. The client is expected to take appropriate actions on each of these events, which will be used to drive the software modem state machine.

### Hook status change

This event indicates that the desk-set handset has been lifted or put down. This event would typically signal activation of the voice I/O circuits.

### Ring indication

This event is sent up to the client queue when a ring signal is detected on the telecom interface. This event would be generated at a nominal rate of once every four seconds in the U.S.

### DTMF tone detected

The ralph tone-man DTMF detection task generates this event when a valid DTMF signal is detected. The corresponding keypad digit and duration are passed with the event.

### Transmit buffer empty

The transmit buffer event signals the user that more data is expected. Due to the multiply-buffered I/O, the caller will have at least one buffer-time worth of elasticity; this translates to roughly 800 ms at 9600 bps using a 1Kbyte buffer.

### Receive buffer empty

The caller must remove received data buffers within a period defined by the buffer latency to avoid overruns. This event will be generated with the same frequency as transmit buffer empty (in the full-duplex case); however, the two events will be asynchronous.

### Modem status change

This event signals a change of the modem pseudo-device status. Possible event types are:

carrier detect	the answering modem has responded
carrier drop	the distant modem has ceased operation
connect	the channel is established and ready for commuications
line quality alert	line quality is insufficient for sustained communications

## Functions

RALPH uses the virtual circuit concept to uniquely identify the resources associated with a communications session. The parameters defining the virtual circuit operation are found in a shared data structure reflects the current status of the virtual circuit. Procedure calls issued by the client manipulate the values in this structure.

An open/close, command, read/write interface is planned. Binary data is passed asynchronously to ralph, which translates it into an analog signal that is coupled to the phone line. A driver-level command interface controls RALPH operation. Character data may be redirected to the RALPH subsystem as well. For compatibility with existing applications, a Hayes-compatible "AT" command set parser may be included to translate "AT" command strings into direct RALPH commands.

The procedural interface will have functions similar to the following:

SetProfile: initialize the RALPH data pump to the operating mode specified by the ralphProfile structure. RALPH will update its operating profile to reflect the passed configuration data and begin operation as specified by the new operating profile.

GetProfile: return the current RALPH operational profile.

DoCommand: directs RALPH to execute the "AT" compatible ASCII command string. RALPH is in the command state. The following "AT" commands are supported:

SendData: transfer data over the analog phone line.

ReadData: read data over the analog phone line. Data is internally buffered until the ReadData is issued.

---

## Implementation Plan

See "JaNeT Implementation" document (30 April 90 - Nichols/Soesbe) for implementation plans and schedule.

---

---

## SPAM: The Signal Processing Access Manager

---

### Overview

An n-way buffered sample handler. May not in fact be an access manager but I like the acronym.

---

### System Architecture

#### SPAM buffer management

---

### Software Interfaces

#### SPAM channel control program

---

### Sample routing services

#### mu-law/linear code conversion

Required for telephone handset to ISDN PCM voice data routing.

## **sample rate conversion**

The telephone codec operates at 7200/9600 samples/sec. We will use 9600 for voice I/O and 7200 for modem emulation. ISDN requires 8000 samples/sec and the hi-fi interface requires 48,000.

Wankel Control Program : if get a CCWord "Convert", have to control the sample rate conversion.

---

## **Implementation Plan**

See "JaNeT Implementation" document (30 April 90 - Nichols/Soesbe) for implementation plans and schedule.

---

---

**BRIAN: ISDN Basic Rate Interface subsystem**

---

**Overview**

An Integrated Services Digital Network (ISDN) provides integrated access to a wide variety of communication services, including voice, data, text and video.

BRIAN gives Jaguar access to ISDN services. It provides a platform for the development of a wide variety of integrated voice and data applications, including :

- screen-based telephony,
- voice enrichment to data communications,
- collaborative computing, and
- remote access to information.

Specifically, BRIAN provides Jaguar with integral voice and data communication facilities as an ISDN "Terminal Equipment 1" operating on the ISDN S/T interface. The Basic Rate Interface (BRI) allows connection either directly to the "Network Termination 1" - in which case Jaguar acts as a TE1 on the "Terminal" (T) interface - or as a multidropped TE1 operating on the "System"(S) interface.

The service provided with a Basic Rate Interface consists of two full-duplex 64-kbps B channels and a full-duplex 16-kbps D channel (2B + D). A B-channel, or "bearer" channel, transmits user information at 64 Kilobits per second (Kbps). The D-channel carries signaling and control information at 16 Kbps, and can also be used to carry packet data. Since the D-channel transfers call control signaling, the full bandwidth of both B-channels is available for voice and data transmission.

This service allows simultaneous use of voice and data applications and meets the needs of most individual users, including residential subscribers and offices.

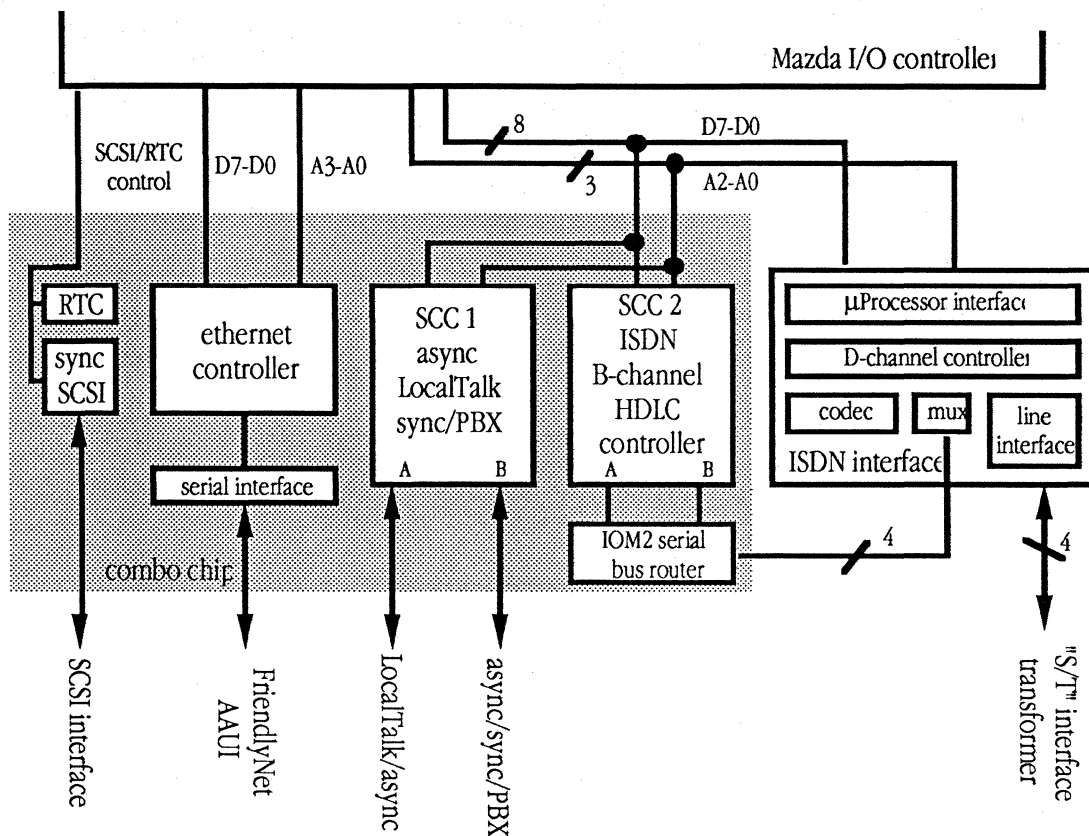
Features possible with a BRI interface include:

- Px64 videoconferencing,
- 128 Kbytes/sec video with 16 Kbit/second audio,
- 7 KHz speech (G.722),
- 32 Kbits/sec compressed speech ADPCM, and
- 16 KBits/sec CELP.

---

**BRIAN Hardware Overview**

The hardware layout is summarized in the below diagram. The following text gives basic descriptions of the hardware components.



The

Hardware descriptions given here are only for the purpose of familiarization with the basic components of the BRIAN system. For hardware details, see "Jaguar Hardware I/O ERS." (Don't worry about it. It's not that different).

### ISDN S/T interface transceiver

The transceiver connects the BRI subsystem to the four-wire, 192 KBit/sec, full duplex S/T interface. This transceiver performs complex analog signal processing and collision detection functions.

### channel controllers

#### D-channel protocol controller.

This on-chip controller provides LAP-D (OSI level 2) access to the BRI by performing bit-stuffing, flag detection, framing, and address recognition.

#### B-Channel protocol controllers (2)

These two controllers provide for transmission of digital data over the two B channels. Since standard X.25 can be used on the B channels, the essential requirement is that the controller handle HDLC frames (such as the SCC).

### ISDN serial bus router

This router is combined into the combo chip to provide the data and external clock signals to the SCCs, since the ISDN BRI silicon uses serial, time-multiplexed data highways that are not directly compatible with the SCC.



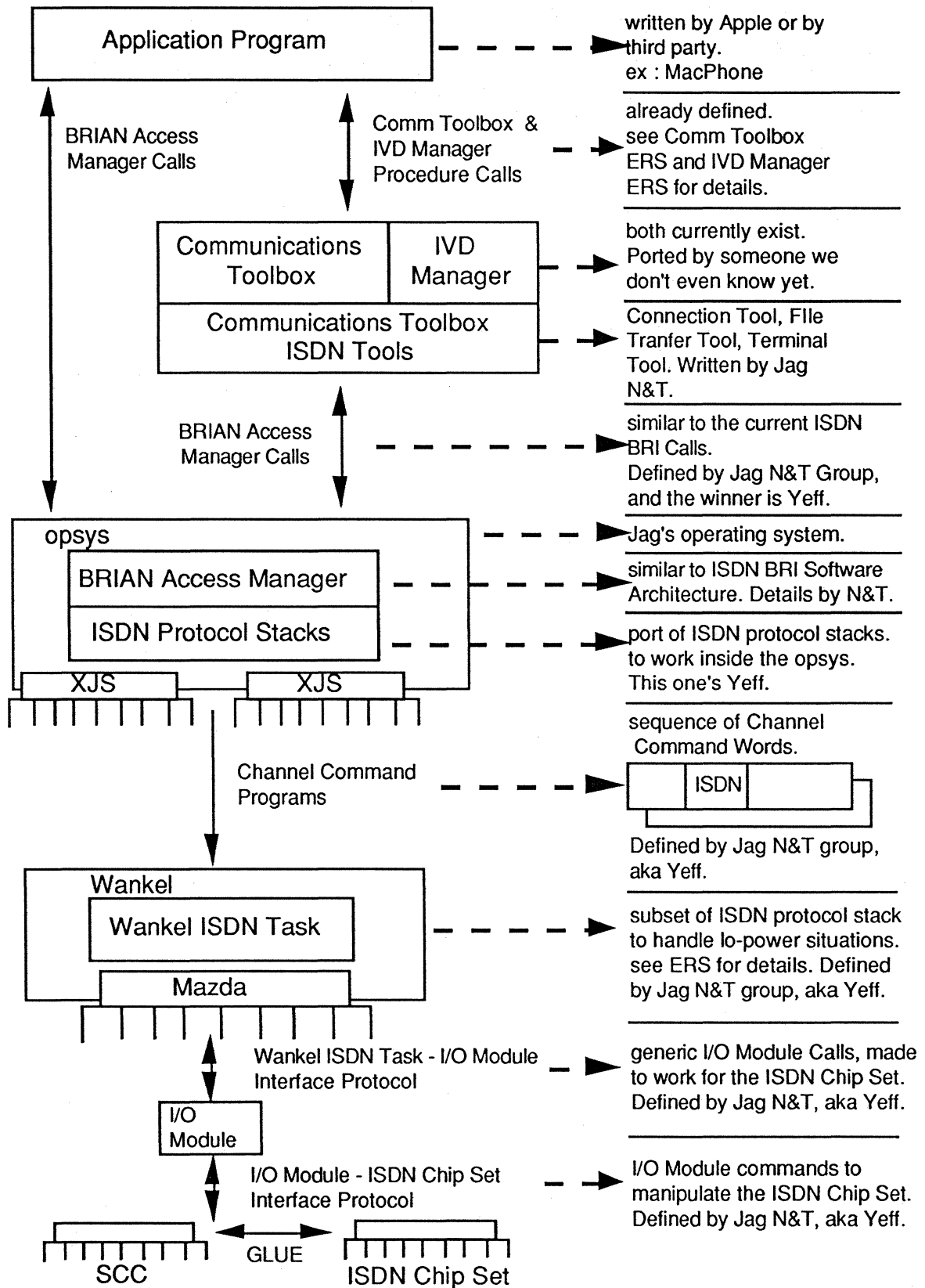
---

## **BRIAN Software Components and Interfaces**

BRIAN's software components are designed to provide full ISDN functionality from level 2 upwards. They are based on the currently available ISDN software from N&C, with specific modifications for Jaguar.

In particular, the Mazda I/O processor (with its Wankel engine) relieves the Jaguar XJS CPU of some of the low-level actions related to I/O organization and low-power situations.

The accompanying diagram gives an overview of the components and their relationship in the system. Also, the responsible party for producing for the software component is listed :



The following is a description of the components of the BRIAN software system and the interfaces to these components.

## **I/O Module**

The I/O Module allows the WANKEL ISDN task to manipulate the ISDN Chip and ISDN-associated SCC.

There are two interfaces associated with the I/O Module, as follows:

### I/O Module - ISDN chip set Interface Protocol

These operations are specific to the ISDN chip set and are designed to allow the I/O Module to manipulate the chip set according to the I/O Module's internal state, which is composed of the direct-access registers.

The operations given here effectively allow the Wankel task to manipulate the ISDN Chip Set, since the Wankel task uses the Wankel - I/O Module Interface Protocol (given below) to set the contents of the I/O Module direct-access registers.

These operations divide into groupings as follows. For each grouping, the necessary internal register state and operations to produce the given chip operation is given.

*These groupings are in no way definite and can change (and probably will) before the appearance of the software ERS.*

#### *ISDN chip operations*

#### *SCC chip operations*

#### *ISDN internal codec operations*

#### *mux operations*

#### *dma operations*

#### *diagnostic operations*

### Wankel - I/O Module Interface Protocol

These operations are extensions of the general I/O Module operations given in "Jaguar Hardware ERS" and are designed to work with the ISDN chip set operations.

*The operations fall into basic categories as follows :*

#### register operations

These operations are the generic I/O module operations which manipulate the various direct-access registers in the I/O module.

##### *read*

These operations read one of the six direct-access registers contained in the I/O Module associated with the ISDN chip set.

The state contained in these registers corresponds to a state of the ISDN chip set, as follows: (not yet determined)

*write*

These operations read one of the six direct-access registers contained in the I/O Module associated with the ISDN chip set.

Changing the state contained in these registers corresponds to a change in the state of the ISDN chip set, as follows: (not yet determined)

signalling a line

These operations are the generic I/O module operations which manipulate the various signal lines between Mazda and the I/O Module.

*assert*

These operations assert one of the signal lines between the I/O Module and Mazda

The signal lines and the operations corresponding to their asserting are as follows: (not yet determined)

*negate*

These operations negate one of the signal lines between the I/O Module and Mazda

The signal lines and the operations corresponding to their negating are as follows: (not yet determined)

address/data line operations

These operations work on the address and data lines between Mazda and the I/O module.

*read*

These operations read the information on the internal Mazda data bus.

*write*

These operations write information onto the internal Mazda data bus.

## **WANKEL ISDN task**

The WANKEL ISDN Task takes Channel Command Programs provided by the ISDN Access Manager and executes them to perform ISDN-related actions and basic functionality as follows:

Directs I/O operations

The WANKEL ISDN task controls the movement of data between Jag Memory, the ISDN Chip Set, the ISDN SCC, the "B" channel controller and internal Mazda buffers.

Performs some Level 2 operations.

The WANKEL ISDN Task also controls some operations related to level 2 of the ISDN protocol stack. This ensures that the data link to the ISDN network remains active while Jag is in a low-power state and that the Wankel ISDN task can power up the Jag XJS CPU when Jag power is needed.

see "BRIAN Standby Power Operation" for details.

WANKEL Task Interface

The operating system uses the WANKEL ISDN task through the passing of Channel Command Programs (consisting of Channel Command Words) into the Mazda memory, where the WANKEL ISDN task reads and executes them.

These commands are solely meant to be a first pass at functional groupings. The final functional grouping of Channel Command Words and their structure is to be determined.

The functionality of the Channel Command Words breaks down into the following areas:

*connection*

These commands are concerned with the creation, maintenance and destruction of connections from Jag to a remote ISDN-capable computer.

commands : open, close, status

*data transfer*

These commands are concerned with the transfer of data between 2 ISDN entities (one of them Jag).

commands : read, write (these commands will also be able to specify such variables as framing, V.120 rate adaptation, synch/asynch, which channel, etc ...)

*task management*

These commands are concerned with the management of tasks currently executing on Mazda

commands : abort, resume, halt

*Mazda memory operations*

These commands are concerned with the movement of data between Mazda memory and external sources (such as the Jag memory).

commands : read, write, move

*operations on channel command program*

These commands are concerned with the channel command words making up a channel command program.

commands : write, read, start, halt, modify

### Example Channel Command Programs

This section will be a list of example Channel Command Programs and the action they perform. This will perhaps eventually expand into a list of "WANKEL Libraries" for use by tasks with common needs.

The function of these "libraries" is still far off in the future. During the first prototyping stage, a more definite decision will be made.

### **BRIAN Access Manager**

The BRIAN Access Manager provides the functionality necessary for an application to use BRIAN's services. It is composed of two parts : the ISDN protocol stack and the BRIAN Access Manager Interface.

#### ISDN protocol stack and call management stack

The ISDN protocol stack contains the complete ISDN protocol stack code and provides for data transferral between two ISDN entities (one of them Jag).

This code will be a port of the ISDN code stack from the N&C ISDN Application to Jaguar.

The ISDN call management stack provides functions related to the creation, continuation and destruction of calls made through the ISDN system.

#### *part of BRIAN Access Manager*

The BRIAN Access Manager contained within the operating system runs the complete call management stack and provides for call associated operations.

#### *level 3 : call setup*

Call setup consists of creation, maintenance and destruction of calls made over the ISDN network.

*different for different switches, but should be transparent*

Since there are different switches within any ISDN system, the call setup protocols (used to signal the switches) are different.

However, which protocol is used should be transparent to the software making the call setup.

#### *level 2 : D-channel*

The D-channel management facilities establish and maintain connections for the reliable transferral of data from one entity to another across the ISDN network using the B-channels.

Data may also be transferred over the D-channel itself.

#### BRIAN Access Manager interface

The BRIAN Access Manager Interface consists of defined calls for applications to use BRIAN's services.

These procedure calls affect BRIAN's internally maintained data structures and map onto calls to the protocol stack and Channel Command Programs to be used by the WANKEL ISDN Task.

The procedure calls fall into groups of fuctionality as follows :

### *call connection and maintenance*

These calls are concerned with call creation, maintenance, status and destruction

calls : open, close

### *data transfer*

These calls are concerned with the transfer of data to the remote ISDN entity .

calls : send, receive

### *call configuration*

These calls are concerned with the configuration of the channel and call parameters.

calls : readConfig, setConfig

### *switch signalling*

These calls are concerned with information sent to the local switch which the Jag ISDN is connected to.

calls : status, ?

### *protocol establishment*

These calls are concerned with the establishment of the protocol to be used for the data transfer between Jag and the remote entity.

calls : setProtocol, removeProtocol

### *accessory control*

These calls are concerned with controlling any accessories connected to the ISDN interface (such as recording/playback, etc.)

### *This might actually belong in the Communication Toolbox Interface*

calls : record, playback, save, read

## **Communications Toolbox interface**

The Communications Toolbox provides a generic interface to services in communication entities within the operating system.

Some calls of the Communications Toolbox map onto calls to the BRIAN Access Manager.

Some features of the Communications Toolbox are also features of the Integrated Voice/Data Manager. *At this point, the relationship between the Communications Toolbox and the IVD Manager remains uncertain.*

The Toolbox interface will also contain an "ISDN Tool" to provide the necessary connection, terminal, and file transfer tools required by the Communications Toolbox

### Communication Toolbox Procedure Calls

This section will be a list of the Comm Toolbox calls related to the ISDN Software and the actions they perform.

*This will probably be a recounting of the various interfaces provided by the Communication Toolbox and how they relate to BRIAN Access Manager calls.*

### IVD Manager Calls

This section will be a list of the IVD manager calls related to the ISDN software and the actions they perform.

*This will probably be a recounting of the various interfaces provided by the IVD Manager and how they related to calls to the BRIAN Access Manager and other Access Managers inside JaNeT.*

### Communications Toolbox ISDN Tool

This will be a description of the ISDN Tool used by the Communications Toolbox to interface to the BRIAN Access Manager.

*The division of the ISDN tool into the three parts required by the Communications Toolbox (Connection, File Transfer, and Terminal) is yet to be determined.*

---

## **HDLC framing services(HUFFY)**

HUFFY (HDLC Universal Fast Framing Yoke, a part of BRIAN) provides services to frame information into HDLC frames for transferral and to deframe HDLC information read from the external lines.

HUFFY also provides for bit-stuffing of information to be transmitted and for de-stuffing of received information.

HUFFY is provided both for internal use by BRIAN and use by other external components of JaNeT, such as RALPH. (See the below example)

### **HDLC bit-stuffing and framing**

HDLC is a bit protocol where the frame delimiter is "01111110". This pattern must not appear within the data section of the HDLC frame. To prevent this, a 0 is inserted after every run of 5 1's in the data being transmitted.

However, the length of this resulting HDLC "bit-stuffed" stream is not necessarily an integral number of bytes, and the huffy task reads information by bytes.

Therefore, the task reading the bit-stuffed information stream (byte by byte) will keep the previously read byte in memory. When the information is complete, the task will work backwards from the end of the information to find the end of the HDLC frame (marked by the frame delimiter). The frame end will be at most 7 bits from the end of the stream

### **interaction example**

The following example shows HUFFY's actions when a RALPH v.29 task requests a framing service for data to be transmitted.



[this will be an animation]

the RALPH v.29 task provides data to be framed

write unframed data to the SCC chip

HUFFY frames the data in the SCC chip and passes it through the interface glue to the ISDN chip.

the information is ready to be read from the ISDN chip by the RALPH v.29 task

---

### **PCM voice routing**

In order to handle voice traffic over the ISDN channel, BRIAN will need to be able to do PCM coding and decoding on voice traffic and route it to the external speakers or internal storage.

A PCM codec is built into the ISDN chip set, which provides for coding and decoding of voice information.

The PCM will be controlled by a task running on either the Wankel or Jaguar CPU.

*This will probably be a list of procedure functionality and various modifiers which can be used to affect the PCM coding and decoding.*

---

### **BRIAN standby power operation**

This is a subset of the ISDN Level 2 state machine which runs to handle packets events not requiring Jaguar power and to start Jaguar from a low-power mode when Jag is needed.

### **UA frame actions**

The central office regularly sends out UA frames to poll for subscribers and check the subscriber's lines. If the subscriber does not respond, the line can be dropped by the central office.

The WANKEL ISDN task replies to these packets, since Jag power is not required.

### **RR frame actions**

The WANKEL ISDN task replies to Receive Ready packets and powers up the Jag to do the higher level protocol actions which follow a RR packet.

### **other frame actions**

*This will eventually be a list of actions the WANKEL ISDN task takes upon other not yet specified frames.*

---

### **Implementation Specifics**

See "JaNeT Implementation" document (30 April 90 - Nichols/Soesbe) for implementation plans and schedule.

---

## Network

---

### Overview

JaNeT's network interface provides Jaguar with connections to Ethernet, LocalTalk and serial hardware. These interfaces allow Jaguar to communicate with most network solutions currently marketed.

The port of the AppleTalk protocol stacks to the Jaguar platform provide Jag with software compatability with all AppleTalk networks.

EtherTalk protocols and LocalTalk protocols differ only in the Data Link level (layer 2), where EtherTalk uses ELAP (EtherTalk Link Access Protocol) and LocalTalk uses LLAP (LocalTalk Link Access Protocol). From layer 3 above, the common AppleTalk protocols (DDP, NBP, ATP, etc ...) are used on both hardware connections.

The rest of the AppleTalk protocol stack will be running as the AppleTalk Access Manager on the Jaguar XJS CPU.

The standard network interface is found on every Jaguar machine and consists of the three components listed below.

### **Issues**

are we going to provide a LAP Manager?

what about TLAP?

what about straight Ethernet? The Hardware ERS implies that this thing can hook right into TCP/IP, does it?

implications of AppleTalk phase 2?

### **FriendlyNet interface**

The FriendlyNet LAN adaptor provides an Apple AUI to support the Ethernet variants 10 Mbps thick cable, 10 Mbps thin cable, and "10 Base T" or 10 Mbps twisted pair. FriendlyNet brings "plug 'n' play" simplicity to Ethernet LAN installations.

The FriendlyNet will be accessible through a common LLC (Logical Link Control) interface.

#### Software interface

The software interface for the FriendlyNet protocol stack will be the same as the current software interface for calls to AppleTalk.

#### ELAP interface

The ELAP interface exists between the ELAP protocols running as a WANKEL/Mazda task and the rest of the AppleTalk protocol stack running on Jaguar.

The ELAP interface consists of calls to configure the Ethernet connection and to send and receive data. They are divided into functionality as follows :

configure (AttachPH, DetachPH, getInfo)

data transfer (read, write, readCancel)

special cases (setGeneral?)

## **LocalTalk interface**

A LocalTalk interface (230.4 Kbps) will be standard on Jaguar, providing it the ability to connect with current LocalTalk networks.

LocalTalk LAP packets are read and written by a WANKEL task without intervention from the user.

### Software interface

The software interface for the FriendlyNet protocol stack will be the same as the current software interface for calls to AppleTalk.

### LLAP interface

The LLAP interface exists between LLAP running as a WANKEL/Mazda task and the rest of the AppleTalk protocol stack running on Jaguar.

The LLAP interface consists of calls to configure the LocalTalk connection and to send and receive data. They are divided into functionality as follows:

configure (openProtocol, closeProtocol)

data transfer (write, read, readCancel)

## **Generic Simple Serial Interface (GUSSY)**

«expansion work on details is needed here»

GUSSY is a generic serial interface which can be used for communication in asynchronous or synchronous mode, or can be used as a special software-dependent interface.

Each of these is controlled by a "generic" WANKEL SCC I/O program which allows existing serial drivers to access the SCC hardware registers indirectly (through a WANKEL Command Control Program).

### Asynchronous interface

This interface provides the standard asynchronous and LocalTalk LAP services found on the Macintosh family of machines. The Jaguar implementation closely follows the Macintosh II variant for compatibility with existing hardware products. These interfaces are controlled by the ubiquitous SCC chip.

The functionality of the interface is composed of the following categories :

data transfer (send, receive)

register operations (read, write)

### PBX / sync modem interface

SCC port A can be configured to accept an external receive/transmit clock. This feature could be used by third parties to provide high performance PBX interconnect boxes using synchronous protocols at bit rates sufficient for concurrent PCM voice and data traffic. This channel interfaces to WANKEL under hardware DMA control.

### Programmatic Interface

Normally, a WANKEL control program will be provided for each application as part of the system software release. For example, the LocalTalk WANKEL control task would be loaded as part of the system startup tasks onto a port configured for LocalTalk.

Since the SCC is shielded from the programmer's address space by MAZDA, there must be a mechanism for "dumb" serial drivers to interface with a virtual SCC.

GUSSY provides a simple, admittedly inefficient access manager interface to a virtual SCC that would be used with the "generic" WANKEL SCC control task. Of course, developers could develop their own WANKEL control programs for special applications.

### *Programmable Features*

The functionality of the programmable features of GUSSY divides up into the following areas:

*configuration (reset, setBuffers, setHandshake, getInfo, setParityChar, setDTR )*

*Serial Data Transfer (send, receive)*

*SCC Register Operations (read, write)*

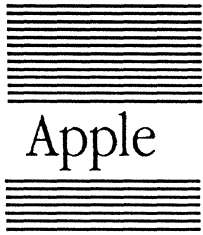
*SCC Manipulation (assert/negate DTR, clearBreak, Xon/Xoff, clear)*

(Currently, for Blue, the programmer directly manipulates the registers of the serial chip in order to change the functions).

---

## **Implementation Specifics**

The implementation plan for the JaNeT's network components is similar in style to the implementation plan for other components. See "JaNeT Implementation" document (30 April 90 - Nichols/Soesbe) for implementation plans and schedule.



Apple

Jaguar Software

Low Level Software

Design Specification  
Special Projects

May 1, 1990

Return Comments to:  
Al Kossow  
Phone: x4-5136  
AppleLink: Kossow2  
MS: 82D

Apple CONFIDENTIAL

---

## Introduction

Jaguar low-level software consists of all code executed by an XJS processor prior to loading the operating system kernel, and the hardware-specific support code needed as an interface between the kernel and the Jaguar hardware. The low level software also provides configuration **hints** to the kernel as machine-dependent configuration records stored in EEPROM or PROM, containing operating system independent information such as CPU serial number, hardware revision level, motherboard I/O configuration, preferences for BLT configurations, network IDs, user names, system name, and bootstrap preferences.

When power is first applied to the machine, enough code is stored in the motherboard non-volatile memory to perform a minimal self test of the possible boot devices, selects an appropriate boot device, and loads the second stage power-on self test and system bootstrap. Possible bootstrap devices include any mass storage device supported through the motherboard or BLT, such as disk or tape, as well as bootstrapping across appropriate network interconnects.

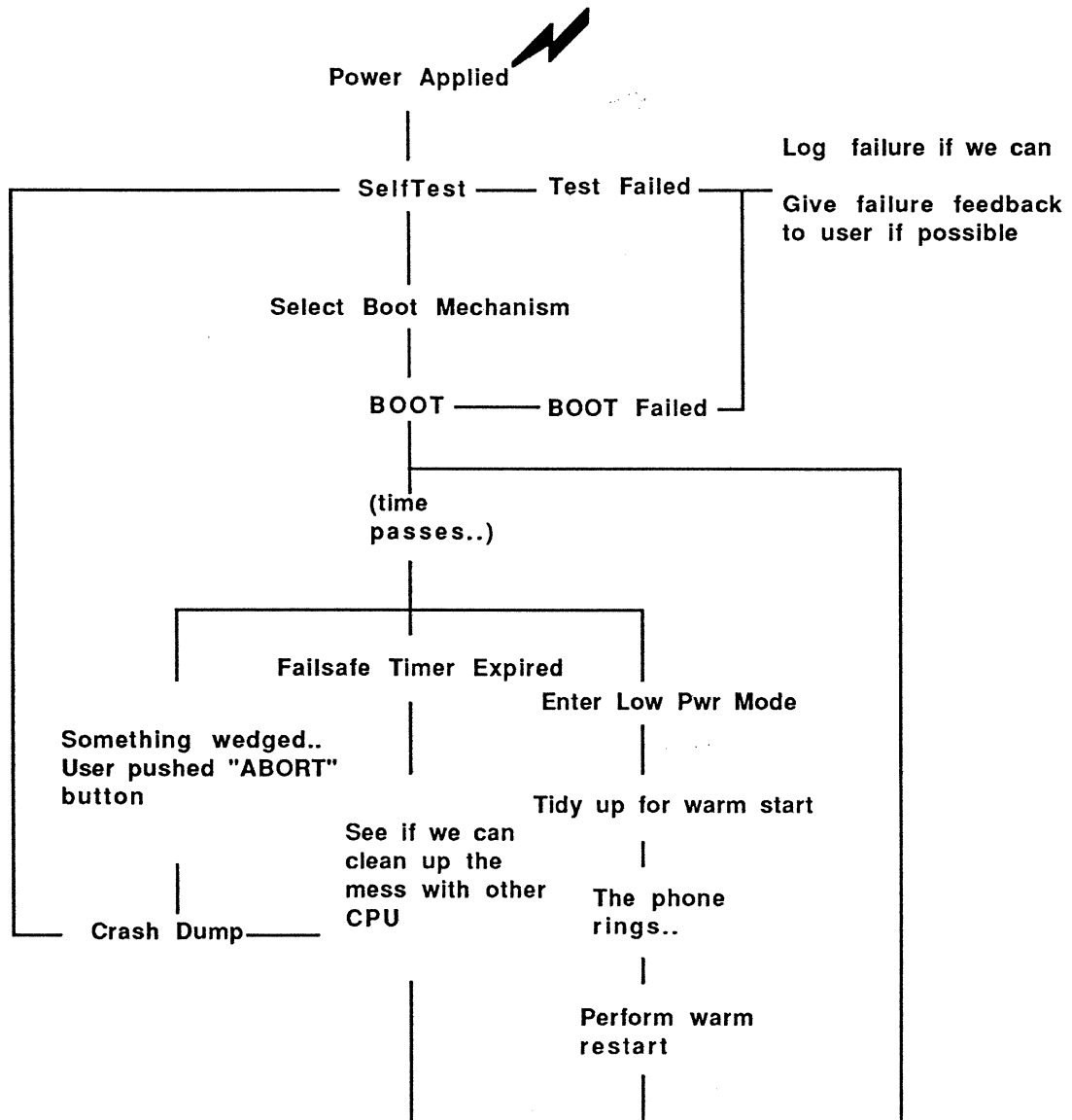
Tools for development and checkout of the system are also being developed to aid the Opus porting team as well as the Jaguar hardware team. These tools include a kernel-level debugger capable of full manipulation and control of the CPU, display, and I/O system of the Jaguar, an integrated kernel profiler for measurement and tuning of the real-time aspects of the machine, and a crash-dump analyzer for post-mortem debugging. The debugging tools will support execution from a remote machine across a backplane or network interconnect.

The low-level software implications of Jaguar always being powered on include providing fail-safe mechanisms for a CPU getting stuck in an infinite loop, low-level backup and recovery from 'power down' mode, and putting the machine into hard power off state or reboot if the user **really** wants to move the machine, or perform a cold restart.

## Major System States

From a low-level software standpoint, there are six major software states of the Jaguar:

- Power-On Self Test (POST)
- Bootstrapping
- Normal System Execution
- Low-power Standby Mode
- Failsafe or Other Hardware Exception Recovery
- User Abort/Powerdown



---

---

## Software Installation/System Autoconfiguration

*(This section may make more sense in the catch-all section)*

Two different system configurations require different thinking for software installation. The first is the traditional Apple stand-alone machine, the second a configuration with insufficient local storage for all the required system software. The latter will require software installation on a shared server machine, which may be supplied by another vendor if the customer lives in a heterogeneous computing environment. On diskful machines, it should be possible to install software using the local floppy, or an external removable media device such as DAT tape, Magneto-Optical or other removable media disks. Even if we ship initial system software with local disks, it should still be possible to replace all of the system software down to the second-stage bootstrap and diagnostics.

### System Configurations

#### Stand-alone Machine

Stand-alone machines have enough local disk storage for a complete system. They would be useful in situations where a user is not connected to a local-area network, for example, a user working in a hotel room. All production Macintosh systems today are stand-alone machines since they all require a local boot device with a complete copy of the system software to run.

#### Client-Server

Client-Server systems trade off the economy of larger, more expensive central storage services for disk transfer speed through a local-area network. Except at the very low end of a product range, most networked computers have some local disk storage for paging and temporary storage and share the common executables and configuration files from a larger "file server".

#### Out-of-Box Setup

What is the customer's *Out of Box Experience* in market-speak? I dunno... It depends on the level of technical experience we will expect of people buying Jaguars. Setup will also depend on our interest in supporting diskless machines. At the minimum, a diskless machine requires some mechanism for selecting the preferred server to boot and get system files from.

#### Installing a Diskful Jaguar

##### Installation Media

Floppy, MO, Tape

#### Installing a Diskless Jaguar

##### Server Configuration

##### Diskless Booting out of the Box



---

---

## **System Development Tools and Debugging**

Jaguar is the most complicated computer system ever developed at Apple. It is very important to have the best software tools available from the start for system development and debugging. Having the best tools, though, does not imply having unproven or high-risk components. The tools must be easy to use, reliable, and capable of extension. Because of the long lead time for HOOPS, initial software development for Jaguar must occur in another development environment. The tools strategy presented here assumes developing using a mixture of cross-development tools under MPW and some native programs running on co-processor cards in a Macintosh until a Jaguar prototype is available.

### **Blue Development Tools**

#### **C Compiler**

##### **Diab C Cross-compiler**

Diab supplies the C compiler which ships with the Tek 88000 NuBus card. It generates good code, and appears to be the preferred compiler source of Motorola for the 88110.

##### **DSG C++ Compiler**

Apple DSG is currently writing a native C++ compiler targeted for the 68k and 88k. The compiler is being developed under MPW for the 68k, but will migrate to HOOPS by the time they are ready to ship. An 88k MPW version may be available in '91.

#### **Assembler**

The 88k assembler is an MPW port of the Diab unix assembler. Andy Heninger has suggested modifying the assembler output to be MPW object compatible instead of generating COFF objects.

#### **Linker**

The linker is an MPW port of the Diab COFF linker. If the assembler modifications are made, it would be possible to use the MPW linker.

## KDB - The Kernel Debugger

*KDB* is a tool for system development. It is a *kernel level* debugger, and permits direct manipulation of CPU and I/O system state, as opposed to an applications-level debugger which manipulates the execution of applications programs. The debugger is split between a target resident nub and a host resident target manipulation program. Code in the nub is kept to a minimum, but enough debugging state is maintained in the nub to cope with transfer of control to another debugging host. This debugger derives its user interface design and code from Michael Tibbott's Hobbit debugger. The split nub/display design and nub communications model is derived from the NuBug debugger, written by Steve Williams et al.

### Target Audience

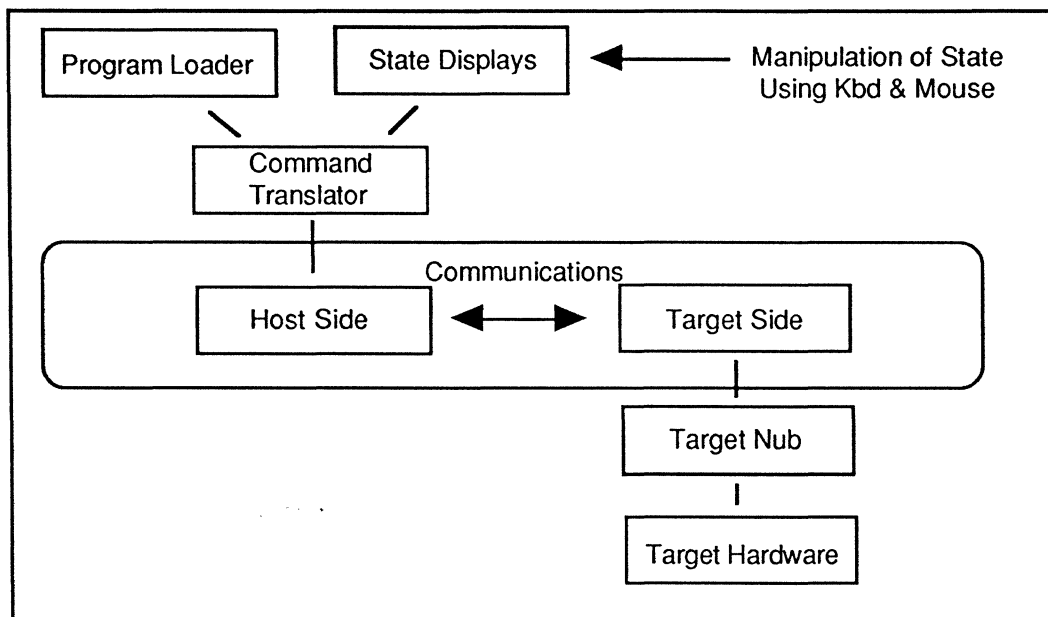
- Opus porting team
- I/O and Network device driver writers
- Hardware bootstrap team

### Features

- Manipulates and displays all accessible hardware state information
- Knowledge of kernel, WDMA, and I/O channel data structures
- Display server for system performance profiling
- Capable of over-the-wire teledebugging

### Target Hardware

- Initial implementation using Cub, Mazda, and Wilson simulators



KDB Block Diagram

## KDB Windows

**KDB** uses different windows for manipulation of different parts of the system. A default set of windows is brought up when the debugger is first started showing the state of CPU 0 and the main function of the initialization program.

### Things we can do

- Load COFF format 88000 executable with symbols
- Manipulate CPU/ CMMU State
- Manipulate Page Tables
- Manipulate Memory
- Display Functions in Assembly Language by Name
- Set / Clear Breakpoints
- Putc / Getc to host window
- Save and Restore complete core image of target for off-line analysis

### CPU General Register Window

CPU 0		
r1	00000000	r26 00000000
		r27 00000000
r2	00000000	r14 00000000 r28 00000000
r3	03400ab0	r15 00000000 r29 00000000
r4	00000042	r16 00000000
r5	00000000	r17 00000000 r30 00000000
r6	00000000	r18 00000000 sp 00000000
r7	00000000	r19 00000000
r8	00000000	r20 00000000
r9	00000000	r21 00000000
	r22 00000000	
r10	00000000	r23 00000000
r11	00000000	r24 00000000
r12	00000000	r25 00000000
r13	00000000	

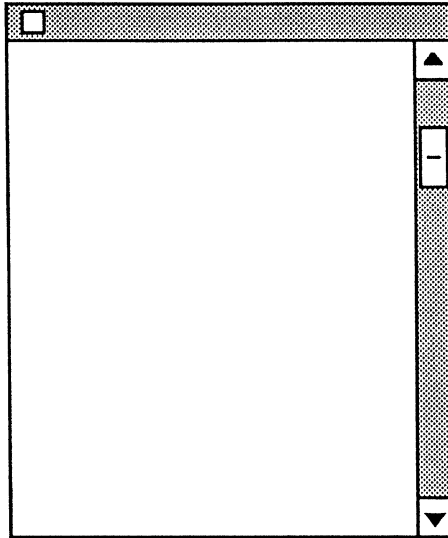
CPU Register Window

### CPU Control Register Window

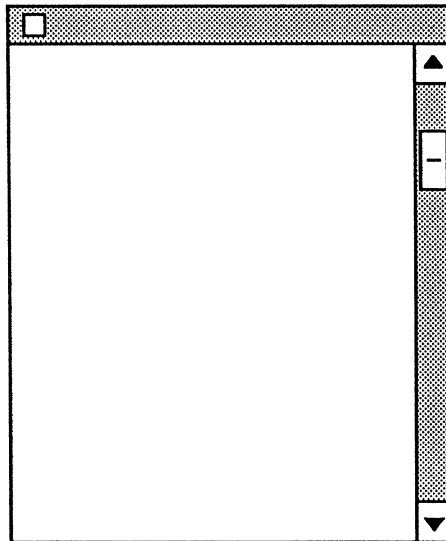
CPU 0	

CPU Control Register Window

### Page Table Display Window

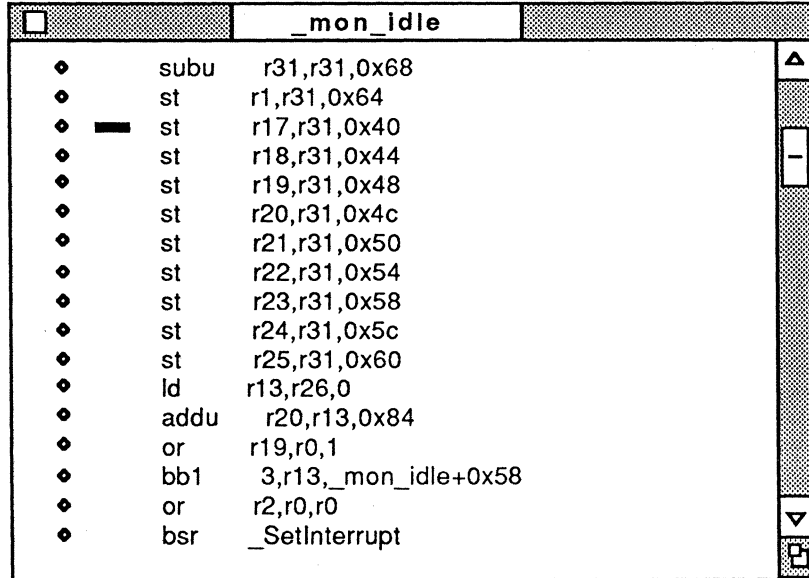


### Exception Table Window



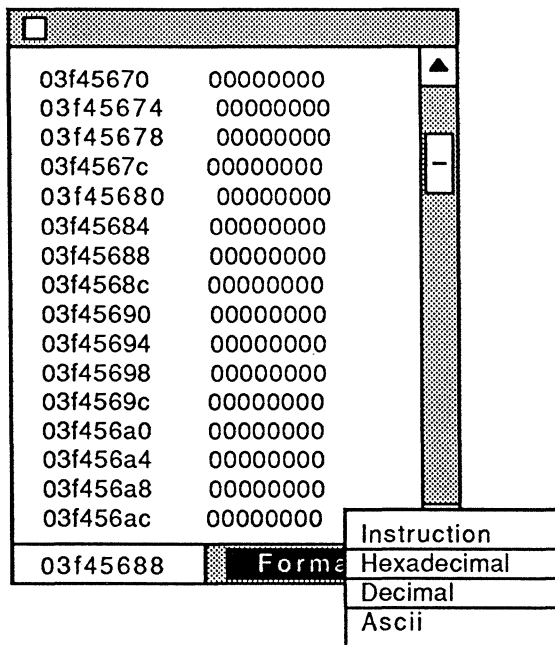
## Function Display Window

The function display window shows a single disassembled function. Breakpoints are set by mousing the little diamonds in front of each disassembled instruction. Shadow branches cannot have breakpoints set, so they have no breakpoint diamonds next to them.



## Memory Display Window

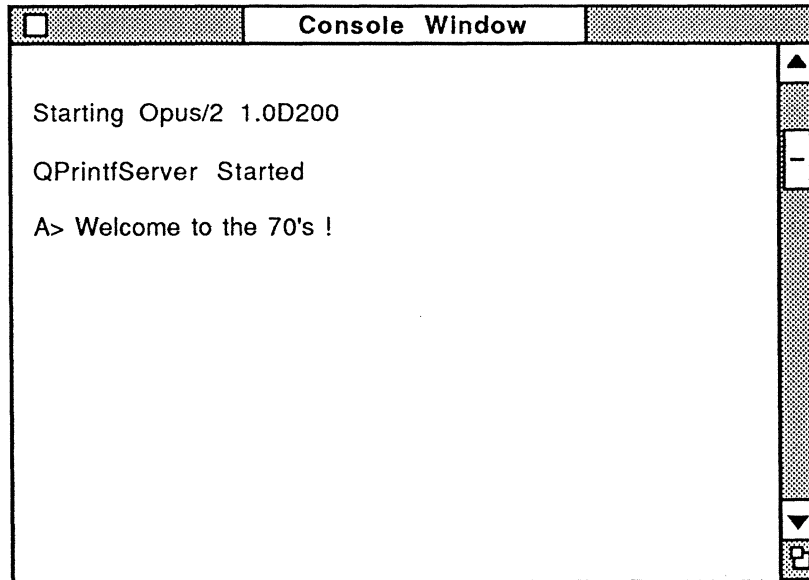
The memory display window shows the contents of target memory in the selected format; hexadecimal, decimal, ascii or as 88xxx instructions. The base address to display is selected by specifying a hexadecimal address in the lower left corner of the window.



Memory Display Window

## Console I/O Window

The console I/O window permits simple "getc" and "putc" operations between the host and the target. It appears as an 80 by 24 ascii "glass teletype" window with scrolling.

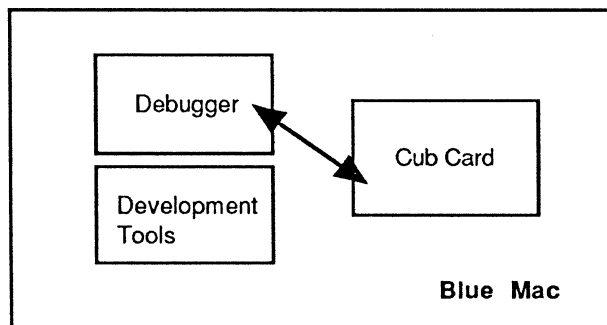


Console I/O Window



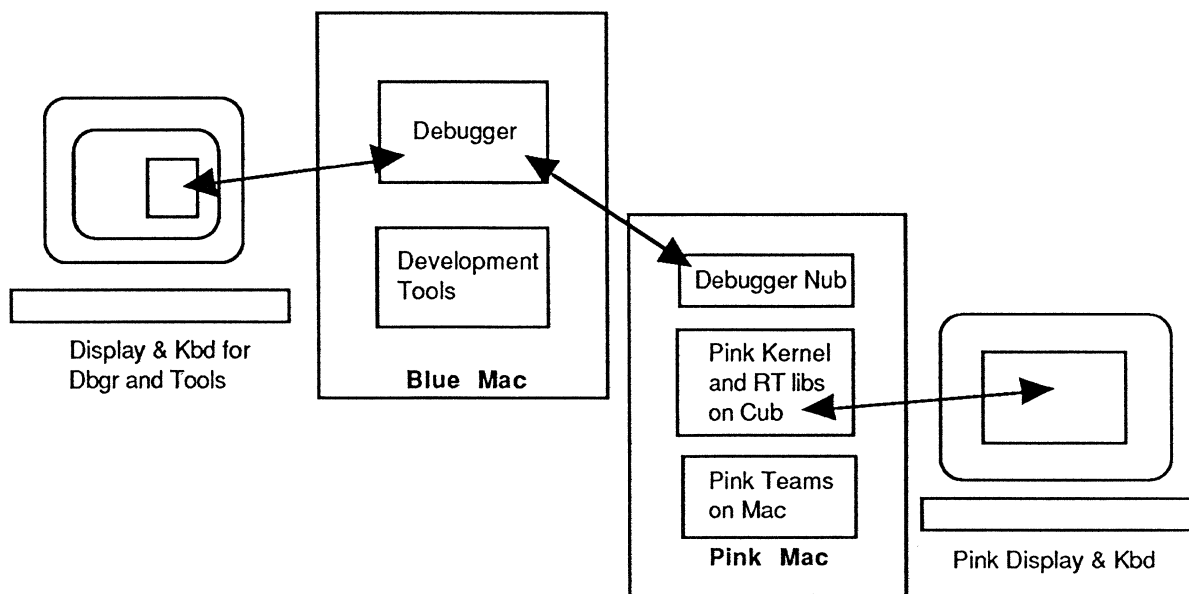
## Implementation Plan

Phase one of the implementation runs on a single machine, with a Tek or Cub card. The communications mechanism between the nub and the host portions of the debugger is implemented using shared memory, but without interrupts, to allow debugging of 88100 interrupt handlers.



**KDB Implementation- Phase One**

Phase two leaves the host side of the debugger on a blue machine, to permit quick code turn-around using the MPW based cross-development tools, and moves the 88000 card to a Pink Macintosh to permit the Opus porting team to migrate portions of the system onto the 88000 card running Opus a piece at a time. This step requires more sophisticated communications. The preferred communications link would be a reasonably high speed channel to allow quick downloads and crash dumps.



**KDB Implementation- Phase Two**

---

## **Boot ROM**

### **Two-stage Self-test and Boot**

The non-volatile storage on the Jaguar motherboard is assumed to contain just enough of the power-on self test (POST) and device-specific boot code to copy the full diagnostic and boot code either from the local disk device or from a network bootstrap server. Support for bootstrapping machines with no local disk, or only a small paging disk, is included. Two different booting protocols will be implemented. The first is an extension to the AppleTalk protocol suite, the second is Bootp, built on top of the DARPA UDP/IP internet protocol suite. Hooks into the bootstrap mechanism will be made available to third-party card vendors for BLT to allow booting from BLT network and mass storage devices.

### **Power On Self Test (POST)**

#### **Part One - POST in ROM**

#### **Part Two - System Tests at Boot Time**

### **System Bootstrap**

#### **Local Booting**

##### **SCSI Devices**

##### **Spock Devices**

##### **BLT Mass-storage Devices**

#### **Remote Booting**

##### **AppleTalk Diskless Booting**

##### **IP Diskless Booting**

##### **Bootp (RFCs 951, 1084)**

##### **BLT Network Booting**

### **Uses of the Local Paging Device at Boot Time**

#### **Boot Blocks**

#### **Crash Dump Storage**

---

---

## **Non-volatile Storage**

Non-volatile storage maintains the permanent configuration information of the machine even if main power is removed. The state of the information in non-volatile storage is static when power is removed, with the exception of the clock/calendar, which continues to advance. Some of the information, such as machine serial number and hardware revision level, may be made non-alterable by the hardware. From the standpoint of low-level software, non-volatile storage is useful for storing all of the information necessary prior to loading the second stage bootstrap, including the Mazda interpreter code, bootstrap related device drivers, and bootstrap preferences. It also maintains preferences for the system as a collection of configuration records, for BLT and network devices. It is also useful for other small bits of information that would want to stay with the hardware, such as machine name, that a user would not want to move if, for instance, the disk connected to the machine were moved.

### **EEPROM (Information that Stays with the Hardware)**

#### **EEPROM Data Format**

#### **EEPROM Contents**

##### **Mazda Boot Code**

##### **XJS First-stage POST and Bootstrap**

##### **Permanent System Information**

###### **CPU ID**

###### **Hardware Revision Level**

##### **System Preferences (Stuff Needed at Boot Time)**

###### **Local I/O Configuration**

###### **Slot Configuration**

###### **Network ID's**

###### **IP Number**

###### **User Name**

###### **Machine Name**

###### **Bootstrap Preferences**

###### **System Type to Boot**

###### **Preferred Boot Device**

###### **Preferred Server ID**

---

## **Motherboard and BLT Drivers**

### **Motherboard I/O at Boot Time**

**Motherboard Power On Diagnostics**

**Motherboard Device Configuration**

**EEPROM Config Preferences**

### **Motherboard I/O Exceptions**

**Mazda Memory Errors**

### **BLT at Boot Time**

**BLT Device Power On Diagnostics**

**BLT Device Configuration**

**EEPROM BLT Preferences**

### **BLT I/O Exceptions**

### **Low-level Implications of Hot Board Insertion/Removal**

---

---

## **Power Down and Exception Processing**

Events which cause the machine to leave "normal" processing include hardware exceptions, user generated exceptions, and requests to enter "low power" mode. An orderly mechanism for controlled power down needs to exist even on a machine that will be on all the time for operations like machine relocation. More detail on power control is available in the document "Jaguar Power Control and Reset Strategy" by Jeff Sewall.

### **Exception States**

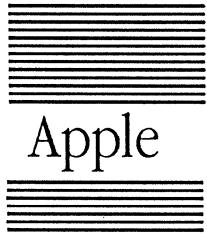
**Low power mode**

**User Abort Button**

**CPU Exceptions**

### **System Hard Power Down**

There are times when the machine will really be powered off. The controlled case is when, for example, the machine must be unplugged to be moved. The uncontrolled case is when there is a power failure on the power mains. The strategy is to provide a mechanism similar to the "Shut Down" menu item on the Mac Plus. The user selects "Shut Down", the system performs power-down operations, and informs the user when it is safe to throw the main power switch.



Apple

Jaguar Software

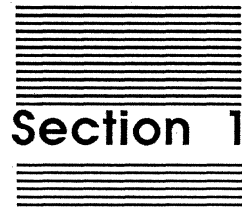
Miscellaneous Issues

Design Specification  
Special Projects

May 1, 1990

Return Comments to:  
Russell Williams  
Phone: 4-7662  
AppleLink: WILLIAMS.R  
MS: 82D

Apple CONFIDENTIAL

A graphic consisting of a central rectangular area with the text "Section 1" in a bold, sans-serif font. This central area is surrounded by a series of horizontal lines above and below it, creating a striped effect.

## Section 1

# Wankel and Wilson Management

---

---

## Wankel Management

All the presently known work of managing Wankel will be handled by bootstrap code, diagnostics (both covered in the Low-Level Software section of the DS), and by the drivers and interrupt handlers for individual devices. This section stands as a placeholder for any Wankel management issues that don't fall in those categories.

---

---

## Wilson Management

The Wilson hardware provides many functions in support of graphics operations:

- Blitting: rectangular regions of pixels in various pixel formats can be copied from one place to another in memory, in the frame buffer, or between the two.
- Alpha blending: Two rectangular pixel regions can be blended according to the alpha values stored with one of the regions.
- Pixel munging: Alpha values stored in one location can be attached to pixel values stored in another.

Wilson is the most efficient way to perform most operations which involve transfers to expansion cards or the frame buffer, and is often useful in writing and moving pixels in main memory as well. It thus affects the video toolbox, the rendering code in Albert, the View System, and the Layer Server. Wilson has limited numbers of internal resources, and these resources have limited bandwidth (as does main memory and the frame buffer), so software must schedule the use of these resources.

---

## Video

### Rendering

### View System Changes

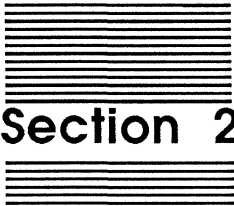
### Layer Server Changes

### Bandwidth Scheduling

Wilson is scheduled by a piece of software called the Wilson Manager. The Wilson Manager must balance two key goals: The first is to schedule Wilson to accomplish as much work as quickly as possible. The second is to minimize Wilson's impact on cpu execution speed caused by the limited memory bandwidth available. In extreme cases, heavy activity by Wilson could cause the cpus to slow down by 50% or more.

Since many uses of the most bandwidth-intensive uses of Wilson will be for predictable, repeated transfers such as displaying decompressed video or compositing animation onto the screen, the Wilson Manager will provide bandwidth reservation services which a task can use to ensure the availability of bandwidth before beginning an operation. We believe it's also useful to allow for degradation: that is, to handle a new request for bandwidth by negotiating a reduction in bandwidth either with the new requester or with an existing bandwidth user.

If degradation is useful, a negotiation protocol can provide the Wilson Manager with accurate bandwidth requirements. While it is difficult for a task to compute a percentage by which it could reduce its cpu use, it is easy to compute the possible Wilson bandwidth reduction. An animation task computing 30 frames per second might not reduce its cpu use by 50% if it only generated 15 frames per second (the reduction might be much less, depending on what the animation task is doing), but the bandwidth needed to display that animation is directly proportional to the number of frames displayed per second.

The graphic for Section 2 consists of a central text label 'Section 2' in a bold, sans-serif font. Above and below the text are two sets of horizontal lines, each set containing ten lines of varying lengths, creating a stylized, symmetrical border around the text.

## Section 2

# Real Time Scheduling

---

### Scope of Problem

Many of Jaguar's capabilities imply the need for real time scheduling: each frame of compressed video must be decompressed in 1/30th of a second, each buffer full of sound or modem data must be produced before an underrun occurs, and so forth<sup>1</sup>. If there is not enough cpu time to complete all the real time tasks, graceful degradation should occur. For instance, rather than simply stop animation or slowly lose synchronization with sound, it is preferable to produce 15 frames per second of animation instead of 30 or reduce the smoothness of the shading applied. The ability to support graceful degradation is a key requirement for Jaguar's scheduler.

Some tasks are of higher criticality than others, because the consequences of missing deadlines are more severe. A software modem task may drop the telephone connection if it misses its deadline and causes an underrun, and recovery may be difficult or impossible. Skipping a frame of animation is less serious. A task which will fail catastrophically if it cannot get the cpu time it needs may wish to check for the availability of that cpu time before it begins execution .

Finally, certain aspects of the user interface must remain highly responsive even though their computation time is not very predictable and the relevant code may be spread across several tasks. Control tracking should remain immediate even if the system is overloaded with real time tasks, so that the user never loses the feeling of being in control of the machine and so that the basic

---

<sup>1</sup>The amount of time between successive invocations of a regularly invoked real time task is called the task's *frame* or *frame time*.



interaction metaphors don't break down. While it doesn't matter if spreadsheet contents update more slowly when a fax transmission is occurring, it is not acceptable for a long delay to occur before buttons highlight or menus drop down no matter what else is happening.

---

## Precedents - None

The real time scheduling problem described here has never been solved before. The novel aspect is the requirement for graceful degradation in response to cpu overload. Since we do not know how to model such a system analytically, the algorithm presented here is based on simulations. The real Jaguar hardware and software will have more sources of variability than can be simulated (e.g. contention for Wilson bandwidth and the interaction of memory bandwidth consumed by Wilson with the memory bandwidth needs of the cpus), so there is a risk that the scheduler will not perform as well in practice as in simulation. Simulation results suggest that additional sources of variability can be covered by adjusting scheduling parameters, at the cost of lower overall cpu utilization.

---

## Dedicated Applications

No matter how much we encourage applications to share resources gracefully, there will always be some developers who wish to completely control the machine. These developers have no interest in giving up cpu utilization in return for graceful degradation and they do not care what other applications they disrupt, they want every available machine cycle. This is quite reasonable in a situation where the machine is dedicated to a particular application. In today's world such developers bypass as much of the operating system as possible. Jaguar's basic scheduling algorithm is known to be theoretically sound, and the method for handling graceful degradation can be limited to advice from the scheduler to the real time tasks. A knowledgeable application developer can therefor take full advantage of the hardware without subverting the operating system.

---

## Real Time Basics

There are several approaches to scheduling real time tasks. Each is discussed briefly in light of Jaguar's requirements.

### Definition of Terms

Frame :	The time between successive deadlines for the task. Animation running at 15 fps would have a 67 ms. frame.
Run time:	The computation time of a task for a single frame, e.g. the amount of time the sound server takes to fill a single buffer.
Quantum:	A task's estimate of its run time for a frame.
Deadline:	The time by which the task must complete its frame.
Overrun:	What happens if the task completes after its deadline.
Reschedule:	Moving a task's deadline to a later time.
Completion:	What happens when a task completes its run time for a frame. It may or may not have overrun.
Slack time:	The extra time a task may consume over its quantum without affecting the ability of other tasks to complete by their deadlines.
Degrade:	Notifying a task that it cannot run for the amount of time it requested.
Warning:	Degrading a task before it starts its quantum.
Cleanup time:	The time required by a task to complete execution of the current frame after it has been degraded.

Preempt:	The process of taking the cpu away from one task to give it to another. This differs from a reschedule in that the preempted task's deadline is not changed and it may still complete without overrunning.
Release:	The process of first making a task runnable (available for scheduling). This would usually be caused by a completion interrupt of some sort and would recur every frame.
Criticality:	The importance of the task ( <u>not</u> its scheduling priority). The least critical tasks are selected for degradation.

## Preemptive Priority Scheduling and Rate Monotonic Scheduling

A preemptive priority scheduler always guarantees that the highest priority process (or processes in a multiprocessor system) ready to run is actually running. Preemptive priority scheduling is the method currently implemented in the Opus/2 kernel. Process priorities are changed only by explicit request; they are not modified by the kernel in an attempt to be "fair". Preemptive priority is a good algorithm when tasks have different priorities but do not have hard deadlines which must be met, but it will not suffice for Jaguar because it provides no way to set limits on how much time the highest priority tasks can consume, and thus no way to handle degradation.

If highest priorities are assigned to those tasks with the severest consequences for failure and the frames are not proportional to priorities (e.g. if fax has the highest priority but its frame exceeds that of any of the lower priority tasks), failures can occur even at low cpu utilizations. This can be repaired only by splitting the longer, more critical tasks into two shorter subtasks.

If the relative frames of the real time tasks maintain their order (e.g. the sound server always fills buffers more frequently than the software modem), and priorities are assigned according to frame time (shortest frame time gets highest priority) and the cpu is never overloaded, then a preemptive priority scheduler is equivalent to a deadline scheduler and performs optimally. This algorithm is called rate monotonic scheduling. Jaguar's real time tasks do not follow this pattern; the relative amount of time consumed by tasks such as sound or animation can vary widely.

## Time Slot Scheduling

With a time slot scheduler, tasks are assigned discrete, repetitive scheduling slots — for instance a task might get 1 ms. of time every 10 ms. while another gets 5 ms. of time every second. Various rules can be used for dealing with tasks which overrun their slots. Time slot scheduling requires advance knowledge of each task's frame time, and those times must be relatively consistent over the entire time the system is running. The assignment of slots is based on each task's worst case run time. Such schedulers are often used for embedded systems with static task schedules and requirements that certain tasks absolutely must be scheduled regularly (e.g. aircraft control systems).

Most Jaguar real time tasks will not take exactly (or even nearly, in some cases) the same amount of cpu time over long intervals, nor will their start and stop times be exactly the same since some are synchronized to external clocks, so this scheme is unsuitable.

## Deadline Scheduling

A deadline scheduler runs tasks based on the time by which they must be completed. The simple strategy of "nearest deadline first" works optimally in the absence of system overloading, but there are no best solutions in the presence of system overload. The key requirement for using a deadline

scheduler and gracefully handling overload conditions is an ability to make reasonable estimates for cpu processing requirements in the short term. While discontinuities will certainly occur, Jaguar's real time tasks will tend to use approximately the same amount of time from frame to frame.

---

## Jaguar CPU Scheduler

The Jaguar cpu scheduler relies on a combination of deadline scheduling for real time tasks and preemptive priority scheduling for non-real time tasks. Real time tasks have priority over all non-real time tasks, but a portion of the cpu time is reserved for non-real time tasks so real time activity cannot completely lock out non-real time activity. Tasks essential to good performance of the user interface are given the highest priority of the non-real time tasks, and the portion of cpu time reserved for non-real time tasks is chosen to be sufficient for good user interface performance.

Proper scheduling of real time frames which include synchronous I/O activity in the middle of the frame, such as audio sample rate conversion or Wilson transfers, must be handled specially. Given an estimate of the time to be spent on this I/O activity the cpu scheduler can still schedule the task properly, but it cannot control the scheduling of the I/O resource to ensure deadlines are met nor can it gracefully degrade the use of such a resource. Those resources can be scheduled by passing the task's frame deadline for use by the resource's manager. Graceful degradation can be handled by waiting to discover from the cpu scheduler that the task has missed its frame and then performing an overall degradation. In other words, since the cpu scheduler knows the overall frame deadline, any scheduling failure will eventually be visible to the cpu scheduler, which can tell the task to reduce its processing requirements.

The interaction of these other activities with cpu scheduling is not fully understood, particularly since Wilson transfers can use so much memory bandwidth that the cpus could be slowed by 50% or more. This effect is highly dependent on how the Wilson manager schedules transfers, and on the cache reference behavior of the currently executing task. We are depending on the adaptive nature of the graceful degradation algorithm to cover such variables: no matter what the cause of the problem, the scheduler will keep degrading tasks until deadlines are met.

The cpu scheduling function is implemented as two separate pieces: the scheduler itself and the routines or methods which real time tasks call to request scheduling services.

## Graceful Degradation

The cpu scheduler implements graceful degradation by detecting when there is insufficient time to complete all tasks by their deadlines and telling the least critical task to clean up its current frame as quickly as possible and thereafter reduce its execution time by a significant amount. Each real time task must poll for this notification. The polling operation does not require a call to the kernel and is very inexpensive; for best results the polling should be frequent. The scheduler also allows tasks which have previously been degraded to be upgraded (increase their cpu usage) when additional cpu time becomes available.

The scheduling routines provide additional help in managing graceful degradation: they keep a degradation level counter which is incremented each time the task is degraded and decremented each time the task is upgraded. This counter can be initialized to any value by the task and used to control the amount of computation performed in each frame.

## Information a Task Must Provide the CPU Scheduler

A real time task must supply certain information to the cpu scheduler for each frame: its deadline, the amount of cpu time it expects to use, and the amount of other (non-cpu) time it expects to use. Non-cpu time is time the task must spend waiting for other activities such as sample rate conversion or Wilson transfers. The task need only compute a cpu time for its first frame. Estimates for subsequent frames will be computed by the scheduling routines. The routines give no help in computing non-cpu time; the task must either measure this time each frame or obtain assistance from the manager providing that service.

## Reserving CPU Time

Some tasks may not wish to run at all if they cannot be assured of doing so without degradation. These tasks wish to discover in advance whether sufficient resources are available. The exact scheme for doing this is currently under investigation.

## CPU Scheduler Performance

The real time scheduler can be implemented either inside the Opus kernel or as a separate task. If implemented as a separate task, it must schedule its client real time tasks by exchanging IPC messages with them and sending requests to the Opus kernel to change task priorities. Such an implementation could spend significant amounts of time sending such messages. Using existing unwrapped Opus performance numbers on a Mac IIx, a single scheduling request would carry an overhead of about 1.7 ms exclusive of scheduling or changing task priorities. If times are measured to include wrappers, the overhead is 3.6 ms<sup>2</sup>. Assuming a 10X speedup on XJS, this is still a very significant amount of overhead. Task switching, particularly for real time tasks, has a critical impact on overall system performance. It would be a major mistake to needlessly inflate the scheduling overhead.

## Routines

The following routine descriptions are only examples to demonstrate how the functionality *could* be provided. They are written in the C style of existing Opus kernel routine declarations. Following the routine descriptions is an example showing how such routines could be used in a real time task.

All routines return `K_NOERR` if no error occurred. The types `Chrono` and `TaskID` are as described in the Opus specifications. `Chrono` is a 64 bit integer absolute or relative time stamp — it can represent either an absolute date and time to sub-millisecond accuracy, or it can represent a certain amount of time (like 10 milliseconds). `TaskID` is a unique task identifier. `Crit` is an unsigned char with 0 indicating highest criticality.

## Structure of a Real Time Task

A real time task must start by computing estimates of its cpu time and non-cpu time for the first frame. It then repeatedly calls `RTschedule` or `RTwait` to begin each frame and indicate completion of the previous frame. The first `RTschedule` or `RTwait` call causes the task to become real time, and it stays real time until a `SetTaskPriority` call is made to change it back to non-real time.

---

<sup>2</sup>On a Mac II.

At the beginning of each frame the task must call `RTschedule` if it can begin execution of the frame immediately, or `RTwait` if execution of the frame must be triggered by another task or ISR. The other task or ISR begins the real time task's frame execution by calling `RTrelease`. A task would use `RTschedule` if it always knows the deadline for its next frame at the completion of the previous one doesn't require external events to complete before it can begin its next frame. A task would use `RTwait` and `RTrelease` if it cannot begin execution of the next frame until some external event or interrupt has occurred. This might be the case for an animation task which ends each frame by requesting a Wilson transfer of the rendered image to the frame buffer; the next frame cannot begin execution until the transfer has completed. Since the task or ISR which releases the real time task for execution can specify the deadline, this method would also be used when the task does not know exactly when its next deadline should be.

During computation of a frame, the task should call `RTshouldDegrade` often. This routine is fast (it doesn't involve a kernel call). If `RTshouldDegrade` returns true, the scheduler has decided there's not enough time for all tasks to complete by their deadlines, and the calling task should immediately clean up execution of the current frame and significantly reduce its execution time for subsequent frames.

A task keeps track of the amount of computation it should be doing during each frame with a degradation level value. This integer is initialized to any desired value by the task. It is incremented by `RTshouldDegrade` whenever degradation is requested, and decremented by `RTschedule` or `RTwait` when it's safe to *upgrade*— that is, when the task may resume using a greater amount of cpu time. For example, a task might initialize its degradation level to 1 and use 10 milliseconds per frame. When the scheduler degrades this task, the degradation level would be 2 and the task might use only 5 milliseconds per frame. At some later time the task might be upgraded: the degradation level would return to 1 and the execution time would return to 10 milliseconds.

## RTsetTaskParams

```
KernErr RTsetTaskParams(  
    TaskID task;  
    Crit    criticality  
    Chrono cleanupTime);
```

### Description

Set the criticality and required cleanup time for a task. No change is made to the task's scheduling as a result of this call; the values are just saved in the kernel for later use. The task does not become real time if it is not already. If a task calls `RTwait` or `RTschedule` without first calling `RTsetTaskParams`, its criticality will be set to 255 and its `cleanupTime` to a system - determined value equivalent to approximately 1 millisecond on a Mac IIx.

### Arguments

<code>task</code>	The identifier of the task whose criticality and cleanup time are to be set
<code>criticality</code>	The new criticality for the task. Lower criticality values represent more critical tasks. The least critical tasks are selected for degradation.
<code>cleanupTime</code>	The minimum time which the task will be guaranteed to run after the scheduler tries to degrade it. This value should be slightly larger than the sum of the task's <code>RTshouldDegrade</code> polling interval and its actual cleanup time.

### Error Returns

<code>K_BADTASKID</code>	Nonexistent task
--------------------------	------------------

## RTwait

```

KernErr RTwait(
    Chrono    deadline;
    Chrono    *cpuTime;          /* In-Out */
    Chrono    otherTime;
    int       waitToUpgrade;
    int       *degradationLevel); /* In-Out */

```

### Description

Wait until the task is released by another task or ISR calling `RTrelease`. If `RTsetTaskParams` has not been called for this task, its criticality will be set to 255 and its `cleanupTime` to a system determined value equivalent to approximately 1 millisecond on a Mac IIX. The caller must supply an initial value for `cpuTime` as an estimate of the amount of cpu time the caller will consume during the following frame. After the first call, `RTwait` keeps a running average of the cpu time used between successive calls to provide estimates to the kernel for scheduling.

`RTwait` also keeps track of whether an upgrade is possible yet after a degradation. Upgrades are possible only when 1) the scheduler says so (which it will do only when this is the highest criticality task which has previously been degraded), 2) the number of frames (calls to `RTwait`) specified by `waitToUpgrade` have elapsed since the task was degraded, and 3) the cpu utilization is at least 10% less than it was when the task was last degraded. These rules combine to prevent rapid oscillations of degradation and upgrading.

### Arguments

<code>deadline</code>	The time by which this task should complete its processing and call <code>RTwait</code> again. If it hasn't called <code>RTwait</code> before the deadline, an overrun has occurred. Deadline should be greater than the current time + <code>cpuTime</code> + <code>otherTime</code> . If non-zero, this value overrides any deadline specified in an <code>RTrelease</code> call.
<code>cpuTime</code>	The address of a location containing the amount of cpu time the task estimates it will use during the next frame. The first time the task calls <code>RTwait</code> , it must supply a value for <code>cpuTime</code> , preferably by timing a sample computation. Thereafter, the <code>RTwait</code> routine will update the value.
<code>otherTime</code>	The amount of time the task estimates it will spend waiting synchronously during execution of its frame (e.g. doing sample rate conversions, Wilson transfers, etc).
<code>waitToUpgrade</code>	The number of frames to wait after a degradation before attempting to upgrade. The purpose of this parameter is to prevent the task from immediately trying to upgrade after being degraded.
<code>degradationLevel</code>	The level of degradations and upgrades. The caller must initialize this counter before the first call to <code>RTwait</code> . <code>RTshouldDegrade</code> will increment it whenever degradation should occur, and <code>RTwait</code> will decrement it whenever upgrading is possible as controlled by the scheduler and the <code>waitToUpgrade</code> parameter.

The task should use the `degradationLevel` value to control the amount of calculation it performs.

## **Error Returns**

`K_RELEASED`

The task has already been released by another task or ISR.



## RTrelease

```
KernErr RTrelease(  
    TaskID    task;  
    Chrono    deadline);
```

## Description

Release the specified task for real time scheduling. That task must have previously called RTwait.

## Arguments

task	The task to be released. It should be waiting.
deadline	The deadline by which the target task must again call RTwait. If the task specified a non-zero deadline in its call to RTwait, this deadline is ignored.

## Error Returns

K_NOTWAITING	The specified task is not currently waiting. It will be notified when it next waits that a release was attempted while it was executing. If the task was relying on the RTrelease call to specify a deadline, this information will be lost.
--------------	--

## RTschedule

```
KernErr RTschedule (
    Chrono    deadline;
    Chrono    *cpuTime;           /* In-Out */
    Chrono    otherTime;
    int       waitToUpgrade;
    int       *degradationLevel); /* In-Out */
```

### Description

Combines the functions of `RTwait` and `RTrelease`. If `RTsetTaskParams` has not been called for this task, its criticality will be set to 255 and its `cleanupTime` to a system determined value equivalent to approximately 1 millisecond on a Mac Ix. `RTschedule` indicates that the calling task has completed its current frame and immediately starts a new frame with the specified deadline. The caller must supply an initial value for `cpuTime` as an estimate of the amount of cpu time the caller will consume during the following frame. After the first call, `RTschedule` keeps a running average of the cpu time used between successive calls to provide estimates to the kernel for scheduling.

`RTschedule` also keeps track of whether an upgrade is possible yet after a degradation. Upgrades are possible only when 1) the scheduler says so (which it will do only when this is the highest criticality task which has previously been degraded), 2) the number of frames (calls to `RTschedule`) specified by `waitToUpgrade` have elapsed, and 3) the cpu utilization is at least 10% less than it was when the task was last degraded. These rules combine to prevent rapid oscillations of degradation and upgrading.

### Arguments

<code>deadline</code>	The time by which this task should complete its processing and call <code>RTschedule</code> again. If it hasn't called <code>RTschedule</code> before the deadline, an overrun has occurred. Deadline should be greater than the current time + <code>cpuTime</code> + <code>otherTime</code> .
<code>cpuTime</code>	The address of a location containing amount of cpu time the task estimates it will use during the next frame. The first time the task calls <code>RTschedule</code> , it must supply a value for <code>cpuTime</code> , preferably by timing a sample computation. Thereafter, the <code>RTschedule</code> routine will update the value.
<code>otherTime</code>	The amount of time the task estimates it will spend waiting synchronously during execution of its frame (e.g. sample rate conversions, Wilson transfers).
<code>waitToUpgrade</code>	The number of frames to wait after a degradation before attempting to upgrade. The purpose of this parameter is to prevent the task from immediately trying to upgrade after being degraded.
<code>degradationLevel</code>	The level of degradations and upgrades. The caller must initialize this counter before the first call to <code>RTschedule</code> . <code>RTshouldDegrade</code> will increment it whenever degradation should occur, and <code>RTschedule</code> will decrement it whenever upgrading is possible as controlled by the scheduler and the

waitToUpgrade parameter. The task should use the degradationLevel value to control the amount of calculation it performs.

## **Error Returns**

K\_RELEASED

The task has already been released by another task or ISR.

## **RTshouldDegrade**

```
Boolean RTshouldDegrade(  
    int *degradationLevel);
```

### **Description**

Used by real time tasks to find out if they should degrade. If `RTshouldDegrade` returns true, the task should clean up its current frame as quickly as possible and reduce its computational requirements for future frames by a significant amount. It is possible that the deadline will pass before the task gets a true result from `RTshouldDegrade` or while it is cleaning up.

This routine should be called often during the real time task. It is very fast and does not call the kernel.

`RTshouldDegrade` always returns false for a non-realtime task.

### **Arguments**

`degradationLevel` Pointer to an integer which is incremented if `RTshouldDegrade` returns true.

### **Error Returns**

None

## **CpuTime**

Chrono CpuTime()

### **Description**

Returns the amount of cpu time used by the calling task since it was created.

### **Arguments**

None

### **Error Returns**

None

## **RTsetNonRTpercentage**

```
KernErr RTsetNonRTpercentage(  
    int *nonRTpercentage); /* in/out */
```

### **Description**

Sets the percentage of cpu time reserved for non real time tasks. If this percentage is set too low, system response to user actions may be erratic.

### **Arguments**

`nonRTpercentage` The address of an integer containing a number between 0 and 100. Non real time tasks are guaranteed at least this percentage of the cpu if they want it. If they don't, the time is available to real time tasks. If the number is outside the range 0 to 100, the percentage value is unchanged. The previous value is returned.

### **Error Returns**

`K_BADPCT` `nonRTpercentage` is not between 0 and 100.

## Sample Code

The following code shows an example of how the real time scheduling routines described above could be used. In addition to the routines described above, this code uses the following Pink time routines: `ToAbsTime` builds a timestamp (of type `Chrono`) representing a specific amount of time (20 milliseconds in the first use below). `AddTime` and `SubTime` add and subtract timestamps, respectively (since `Chrono` items are 64 bits long, they cannot be added directly in C).

```
/* When this routine is called, the task is not yet a real time task.*/
/* The input parameter is the time of the first deadline.          */
/* This task has a deadline every 20 ms.and calls ComputeFrame once */
/* per frame. It waits 50 frames after a degradation before trying  */
/* to upgrade.                                                    */

void DoRealTime(Chrono deadline) {
Chrono startTime, endTime, cpuEstimate, otherEstimate;
Chrono deadlineIncr, fudgeTime;
int    degradationLevel = 1, waitTilUpgrade = 50;

ToAbsTime(20, Millisec, deadlineIncr);
ToAbsTime(5, Millisec, fudgeTime);

/* Compute estimate of first frame time and add a 5 ms. fudge factor */
/* The third parameter to ComputeFrame tells it not to write output */

CpuTime(&startTime);
ComputeFrame(&otherEstimate, &degradationLevel, false);
CpuTime(&endTime);
SubTime(endTime, startTime, cpuEstimate);
AddTime(cpuEstimate, fudgeTime, cpuEstimate);

/* Schedule frames 20 milliseconds apart */

while (1) {
    RTSchedule(deadline, &cpuEstimate, otherEstimate,
               waitTilUpgrade, &degradationLevel);
    ComputeFrame(&otherEstimate, &degradationLevel, true);
    AddTime(deadline, deadlineIncr, deadline);
}

/* ComputeFrame() routine is on next page */
```

```
/* Compute a frame of stuff. This routine computes some mythical */
/* rows and columns, with a synchronous I/O request in the middle.*/
/* The degradationLevel determines how many rows and columns will */
/* be computed. The time to do the I/O (plus 2 milliseconds of */
/* fudge time) is returned in otherEstimate. If doFinalOutput is */
/* true, DoOutput is called. If doFinalOutput is false, */
/* ComputeFrame has been called just to collect timing info and */
/* so output should not be written. */

void ComputeFrame(
    Chrono *otherEstimate;
    int *degradationLevel;
    Boolean doFinalOutput) {

    int row, col;
    Chrono fudgeTime;

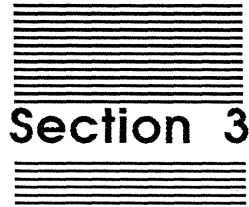
    for (row = 0; row < numRows / degradationLevel; ++row) {
        DoRowStuff();
        if RTshouldDegrade(&degradationLevel) {
            DoCleanup();
            return;
        }
    }

    DoExternalIO(&otherEstimate);
    ToAbsTime(2, Millisec, fudgeTime);
    AddTime(otherEstimate, fudgeTime, otherEstimate);

    for (col = 0; col < numCols / degradationLevel; ++col) {
        if RTshouldDegrade(&degradationLevel) {
            DoCleanup();
            return;
        }
        DoColStuff();
    }

    if (doFinalOutput)
        DoOutput();
}
```





## Section 3

# 68K Emulation

---

## Introduction

In order to run Macintosh applications on the Jaguar, it is first necessary to be able to execute 68k (68000, 68020, 68030, etc.) code on the Jaguar's 88k (88110) processors. Obviously, such an endeavor is useful only if Macintosh emulation is provided on top of the processor emulation. Therefore, we assume that such a Mac emulator, the Blue Adapter, will exist.

There are several methods by which we can implement 68k emulation. Each of the options are discussed below, along with the tradeoffs that each present. A final section describes the most optimal implementation schemes.

---

## Option 1: Full Decoding

Full decoding involves a very straightforward procedural algorithm for mapping the opcode into its generic instruction. This has the benefit of a minimal RAM footprint, although it is unquestionably at the cost of performance. It will also be fairly difficult to modify, since each piece of code handles a potentially large set of cases. Because of this it almost would have to be implemented in a high level language, thus further affecting performance. Portability via 68k to C emulation (i.e. no destination machine model) would be unreasonable slow.

---

## Option 2: Table-driven

This approach should be much faster than a straightforward decoding, at the expense of a very large code and data working set. In addition, it can be coded directly in 88k, because of the fine grain modularity caused by the breakdown of handler code by 68k opcode; each handler should only be about 4 to 8 instructions long. However, because there are so many routines a code generating tool will be required.

Of course, because the working set is huge there are cases where this type of emulator will run more slowly than the one described above. It will no doubt slow the rest of the system more, by hogging resources (cache and physical memory). Hopefully most instructions will never be executed and hopefully the handlers and handler pointers will exhibit good locality of reference.

### Costs

The full blown approach involves a table with 65536 4 byte entries, each a pointer to a routine that handles that opcode. The memory cost is  $s_{init} + 65536 * (s_{loop} + s_{handler} + 4) \approx 2.25$  MBytes, where:

$s_{init}$  = size of init code in bytes  
 $\approx 18$  instructions \* 4 bytes/instruction = 72 bytes  
 $s_{loop}$  = size of generic loop code in bytes  
 $\approx 4$  instructions \* 4 bytes/instruction = 16 bytes  
 $s_{handler}$  = size of average 68k opcode handler in bytes  
 $\approx 4$  instructions \* 4 bytes/instruction = 16 bytes

The loop code cost is counted per instruction because in order to eliminate the extra branch (to the top of the loop) and to get maximal parallelism we put the generic loop code, getting the next opcode and dispatching to the proper handler, inline. The parallelism comes from the fact we make sure that in every case this code uses different registers than the opcode specific code.

Note that the whole memory cost is fixed. Like a shared library its size is amortized over the number of clients. The size can be decreased considerably by folding entries for the various opcodes of a particular instruction into one handler. That is, all of the handler pointers will point to the same routine. For example, all of the A-line and F-line traps could be handled this way, which would remove  $12.5\% \approx 160k$ . The performance hit in doing so should be minor, especially since these are decoded on the 68k itself anyway. We can do the same thing with the inline generic loop code, which would save up to 1 megabyte.

The speed of the emulator is completely dependent on the instruction mix. Measurements by the Systems Technologies Group show that **move**, **bcc**, **movea**, and **add** account for over half of the instructions executed. The two **move** handlers below take 2 and 4 cycles respectively. A pessimistic initial guess is that the average **move** handler will take 5 cycles. **Add** should take at most the same number of cycles as **move**, since the emulator uses 88k **adds** to implement **move**. The one catch is that we must track the overflow bit, which is set iff the two sources have the same msb (most significant bit) as each other and a different one than the result. This is a bit of a nuisance, but if we assume the 88k overflow exception does not happen often then it should be  $\approx 5$  cycles. The cost of the loop code is 2 cycles.

Another fact to remember is that the 88k assumes that all memory operations require alignment, or an exception is generated. This is unfortunate as a great number of 68k longword accesses occur on 2 mod 4 addresses. For example, an informal perusal in MacsBug shows the stack to be longword aligned 2/3 of the time.

There are three ways to handle this problem. We can change all code that references longwords to reference two halfwords. We can check for alignment and handle the appropriate case. We can assume alignment and clean up in an exception handler. At this point the second option above seems most efficient. Although it is not included in the code below, we assume the overhead for adding this averages to 1 CPI (cycle/instruction), which incorporates the fact that only longword **moves** are affected by alignment problems.

So how what does all of this add up to, in terms of performance? Assuming a 50 MHz 88k, the emulator should be able to execute on average 1 68k instruction in  $50M \text{ cycles/sec} / 8 \text{ cycles/instruction} = 6.25$  MIPS. In contrast, my Mac Ix runs at about 4.5 MIPS.

Now consider the case where the loop code has been threaded through each handler. The two move handlers below take 3 and 4 cycles each. A pessimistic guess is that the average move handler will take 5 cycles. Add should take at most 5 cycles too, because it is less likely to have complex addressing modes than move. Adding 1 cycle for alignment problems the whole thing should run at 50M cycles/sec/6 cycles/instruction = 8.33 MIPS.

## Code

```

RunInterpreter:                ; RunInterpreter(void *pc)
InterpreterBegin:
    add.u r15,r0,r2            ; Get pc into r15
    lda.u r14,r0,HandlerTable ; Get offset to handler table
                                ; Note: Not correct. How do
                                ; you load an absolute or PC-
                                ; relative address on the 88k?
; Zero out all the 68k register holders
    add.u r16,r0,r0
    add.u r17,r0,r0
    add.u r18,r0,r0
    add.u r19,r0,r0
    add.u r20,r0,r0
    add.u r21,r0,r0
    add.u r22,r0,r0
    add.u r23,r0,r0
    add.u r24,r0,r0
    add.u r25,r0,r0
    add.u r26,r0,r0
    add.u r27,r0,r0
    add.u r28,r0,r0
    add.u r29,r0,r0
    add.u r30,r0,r0
    add.u r31,r0,r0
;
; Main Loop: Loop on the pc and case off of its dereferenced word
opcode
;
; Register Usage:
; r1: return address
; r2: argument to interpreter (do we need to save this)
; r3-r10: handlers' scratch registers
; r11: opcode word
; r12: pointer to next handler
; r13 condition code storage
; r14: base of handler table
; r15: pc
; r16-r23: d0-d7
; r24-r31: a0-a7
;
rConditionCode EQU r13
rHandlerTable EQU r14
rPC EQU r15

rScratch1 EQU r3
...
rScratch1 EQU r10

InterpreterLoop:

```

```

rOpCodeWord EQU    r11
rHandlerPtr  EQU    r12

    ld.hu rOpCodeWord,rPC          ; Load the next word from the pc
    ld.u  rHandlerPtr,rHandlerTable,[rOpCodeWord]
                                           ; Use it to index the handler table
    jmp.n rHandlerPtr              ; And jump to it
    add   rPC,rPC,#2               ; Increment the pc

MisalignmentExceptionHandler
    ldcr  rExceptScratch1,nip       ; get next instruction ptr
    ldcr  rExceptScratch2,xip       ; get current instruction ptr
    stcr  rExceptScratch1,xip       ; and set the current to it
    rte

HandlerTable:
    Handle_Ori.b_0_d0
    Handle_Ori.b_0_d1
    ...
    Handle_FFFF

; Handler Routines          Condition Codes   Opcode
;                          X N Z V C
Handle_Move.l_d2_d3:      - * * 0 0          2602
    add.co    r19,r18,r0            ; Move r18 into r19, clear the carry
    br.n     InterpreterLoop        ; and done
    cmp      rConditionCode,r19,r0  ; Capture the condition codes
; With inline loop code it looks like this:
;    ld.hu rOpCodeWord,rPC          ; Load the next word from the pc
;    add.co    r19,r18,r0            ; Move r18 into r19, clear the carry
;    ld.u  rHandlerPtr,rHandlerTable,[rOpCodeWord]
;    cmp      rConditionCode,r19,r0  ; Capture the condition codes
;    jmp.n rHandlerPtr              ; And jump to it
;    add   rPC,rPC,#2               ; Increment the pc

Handle_Move.l_d2_minus(sp):  - * * 0 0          2F02
    subu    r31,r31,#4              ; move the stack down by 4
    ld      rScratch1,r31           ; Get stack word
    add.co    r19,rScratch1,r0       ; Move into r19, clear the carry
    br.n     InterpreterLoop        ; and done
    cmp      rConditionCode,r19,r0  ; Capture the condition codes
; With inline loop code it looks like this:
;    ld.hu rOpCodeWord,rPC          ; Load the next word from the pc
;    subu    r31,r31,#4              ; move the stack down by 4
;    ld      rScratch1,r31           ; Get stack word
;    add   rPC,rPC,#2               ; Increment the pc
;    ld.u  rHandlerPtr,rHandlerTable,[rOpCodeWord]
;    add.co    r19,rScratch1,r0       ; Move into r19, clear the carry
;    jmp.n rHandlerPtr              ; And jump to it
;    cmp      rConditionCode,r19,r0  ; Capture the condition codes

Handle_Add.l_d2_d3:      - * * 0 0          D682
    add      rOverflow,r0,r0         ; clear the overflow flag
    add      r11,r19,r18             ; Check for overflow (Might except)
    addu.co  r19,r19,r18             ; Move r18 into r19, set the carry
    cmp      rConditionCode,r19,r0  ; Capture the condition codes

```

```

    br    InterpreterLoop        ; and done
OverflowExceptionHandler
    ldcr  rScratch2,nip          ; get next instruction ptr
    set   rOverflow,r0,32<0>    ; set the overflow flag
    stcr  rScratch2,xip         ; and set the current to it
    rte

Handle_Adda.w_Immed_a3:        - * * 0 0        26FC xxxx
    ldcr  rScratch1,rPC         ; Load the immediate from the pc
    br.n  InterpreterLoop       ; and done
    addu  r26,r26,rScratch1     ; Move r18 into r19, don't change cc

Handle_Bcc.s_pc+70:
    ldcr  rScratch1,PSR        ; Get the control reg w/ carry in it
    bbl   #CarryBit,rScratch1,InterpreterLoop
                                ; and done if it's set
    br.n  InterpreterLoop       ; otherwise, done
    add   rPC,rPC,#70          ; and change the pc

Handle_Bcc:
    ldcr  rScratch1,rPC         ; Load the displacement from the pc
    ldcr  rScratch2,PSR        ; Get the control reg w/ carry in it
    add   rPC,rPC,#2           ; Increment the pc
    bbl   #CarryBit,rScratch2,InterpreterLoop
                                ; and done if it's set
    br.n  InterpreterLoop       ; otherwise, done
    add   rPC,rPC,rScratch1     ; and change the pc

```

---

## Option 2.5: Split-level tables

We need not have a giant table. We could have a 256 entry table and just case off of the high byte of the opcode. This will diminish the size of this table from 256k to 1k, although there will undoubtedly be numerous secondary tables. At least these tables can be allocated only to the commonly used instructions.

The table lookup costs 2 cycles. Therefore this emulation could run at 6.25 MIPS, at best.

---

## Option 3: Native Execution

There is a spectrum of models here, based on when the code translation is done. One extreme is to translate a Mac application from 68k to 88k statically. The speed of translation should be reasonable, so that it can occur at launch time the whenever the user runs an untranslated Mac app on the Jag. If this is not feasible then perhaps background translation can be done to hide this cost. The efficiency of the translated code should be tremendous, though, since it is simply native 88k. The Mac code could run at 100 MIPS peak, or even at 200 if Blue eventually becomes preemptible or (better yet) supports lightweight tasks, so that we could use both CPUs. Note that these MIPS are a measure of 88k instructions, rather than 68k. We would have to divide by the ratio of 88k instructions per 68k instruction to get the number of 68k MIPS. However, because of some of the powerful features of the 88110 (large register set, fast calling conventions, etc.) some would claim that in practice this ratio would be less than 1.

One of the problems with a translation approach is that it relies on a static analysis to determine exactly what is code. The separation of code and data is not well defined in code segments, much less other resources such as WDEFs. The static approach depends on a control flow analyzer to generate a graph of code fragments, given the entries in the code 0 jump table.

However, many control changes are not (easily) tracked. For example, how could the analyzer find foo2() in the code below?

```
typedef void      (*FooFncPtr) ();
void             foo1(), foo2(), foo3();
FooFncPtr       foos[] = {foo1, foo2, foo3};

void
CallFoos(int i)
    {
    (*foos[i]) ();
    }
```

If we had some idea of the bounds (upper and lower) of foos[] it might be possible...but we do not. There are numerous other examples of such problems, especially switch statement constructs. The problems are much worse with hand-assembled code; it is all too easy to create pc-relative data. At least with a high-level language the data is either between functions or in a certain recognizable pattern. Also, it is not as clear what constitutes an **rtts** equivalent, since it may be a **jmp (a0)** where register a0 was saved at the beginning of the routine. Conversely, some **rtts** are really just **jumps**. This means that the static analyzer must simulate the stack as well.

We are very optimistic that these types of constructs do not occur very often, though, and their use will become increasingly less frequent. We base our optimism solely on intuition, and also on the fact that the first 4k of FullWrite examined contained no offending constructs.

Besides, if we can use a static analyzer on system software (Mac Rom, system patches, and code-like resources) with some manual fixes, then at least all system functions will run at Jaguar speed. Much of this conversion will be de facto, as the Blue Adapter subsumes major portions of toolbox functionality; it will all be compiled into 88k code.

We should also explore the other end of the spectrum — generating 88k code at emulation time. This trivially solves the flow control problem; the code that is translated is exactly that code that needs to be. It also saves space in that only code that is executed is translated. However, this method has other complexities due to the size difference between 68k and 88k code.

This does better than a static analyzer if we feel we cannot reach significant portions of code with the latter. If most is reachable, the analyzer is a big win. It is a one time operation (unless we flush translated code for a full disk) and is inherently less complex.

### The Big Question

Whether we convert 68k to 88k statically or on the fly, we must still determine some way to actually allow native code and the emulator (or an on the fly trto coexist. This can be handled quite easily for the stack and registers; the translated code merely follows the same conventions as does the emulator

(e.g. d0-d7/a0-a7Æ r16-r31). The problem lies in the pc. Consider a run of code, a sequence of either 68k or 88k code.

If the size of a run of 88k code was the same size as a run of 68k, then we could simply replace the latter with the former inline as we translate. This could be done by copying the 68k code segment to a different resource of type '6888' and then just over-patching portions with the appropriate 88k. Unfortunately, we cannot do this. The 88k code will not be the same size and we cannot calculate the difference a priori.

So, we need to stick a run of 88k code in memory, but adjust all pc-relative code pointing into the run, and all data that is directly or indirectly generated by pc-relative data that points into it. The former is straightforward; simply keep a table of all pc-relative instructions and modify them as the 88k runs are inserted. The amount necessary is determined by a table mapping 68k runs to 88k runs. It turns out that patching pc-relative data is no work at all, because the data at some point must have a pc-relative instruction to generate it. Furthermore that pc-relative instruction must have come from the same code segment, since the relative arrangement of segments is not determined until runtime. We need only hold on to the mappings until the entire segment has been translated, so the eventual cost of them is 0.

And what about pc-relative code in the hidden 68k nuggets. Well, if we keep the mappings around at runtime then we know how much to patch in the emulator too. Every time the emulator runs an instruction that changes the pc it makes the change and then calls a routine that fixes up the pc, based on the run mappings.

Better yet, if we have the ability to call the 68k to 88k translator at runtime (and why not?), then we can simply convert that 68k nugget on the fly. We simply call the run mapper and then jump directly into the new code. Also, jumps into potential 68k code can be translated to a call to the run mapper with the address instead (thus causing the translation of the target 68k code). In this scenario we need not even have an emulator, at least for this code. Better yet, after the first few runs the code will run a full 88k speed!

### **Across Segments**

All of the above deal with code only in a particular code segment. We still have to determine how to handle jumps across segments. That is, we need a scheme to handle the app jump table.

Each entry in the jump table consists of 8 bytes, of which the last 6 are code. It holds the instruction that either loads in the code segment or else jumps to the routine if the segment is already loaded. When the segment is loaded the entry is created as a jump instruction to the sum of the beginning of the code segment and an offset already in the jump table. The offset was loaded originally from code segment 0, which also has the rest of the entry in it; the jump table is just an image of code 0 until the first segment is loaded. So, the mapper must change code 0 when it adds 88k runs (if it can fit the run in the code segment — see above). If it adds it at runtime it must also change the jump table.

But how can we put 88k instructions in the jump table? Well, it depends on how much trap patching we wish to do (and ain't it always the case?). Even with no patching at all, the only constant in the jump table that we need to observe is that the size of a jump table entry is 8 bytes long, with the last 6 containing the code to either load the code segment or jump to the routine. The latter is not possible

in 4 bytes on the 88k strictly speaking, because the jmp instruction requires a pre-loaded register, and the br is pc-relative, with a 26 byte offset, which only allows  $2^{25} = 33.5$  Mbytes (since we lose 1 byte offset for sign) between the entry and the code. Is this large enough for tomorrow's 32-bit apps, or will we have to change code 0 so that the entry jumps to some code at the end of the jump table, which does a full jump. This should have the effect of at most doubling the size of code 0 and the jump table. The current size of the code 0 for the MPW Shell is 8k. The size for MacWrite 5.0 is 2k. The size for FullWrite 1.1 is 23k, but this is unusually large.

Note that it is possible for an application (or more likely the development system that generated it) to put whatever it would like in code 0. The above translation may cause problems for it. For example, a jump table entry might branch off to extra code at the end of the jump table, similar to what we propose above. Therefore, we probably need to use the full power of the mapper on code 0, not just a simple translation of each entry.

### Other Code

Code can live in any resource, theoretically. Resources other than code present no special problems, as long as we keep the run mappings current. This is relatively easy for code, as it happens only at `_LoadSeg` and `_UnloadSeg` time. General resources are a bit trickier because they might move if not locked down. It would be a shame to have to patch `_HLock` to readjust mappings.

Perhaps the answer is to have the mappings be handle and offset based, rather than straight address range translations. Actually, the mapping would have to support either handles or pointers, since not all code resides in the former (e.g. the ROM). This would narrow the problem to patching `_DisposHandle`. Ideally we would patch `_ReleaseResource`, but this would not deal with detached resources. Unfortunately, the addition of handles in the mappings will slow mapping down considerably.

### How do it know?

The run mapping interface looks something like this:

```
typedef unsigned short  Instruction68k;    // Smallest 68k opcode 2 bytes
typedef unsigned long   Instruction88k;    // All 88k opcodes 4 bytes

class TRunMapping : public TCollection, public TCollectible
// Want to keep a set of them
{
private:
    TRunMappingType      fRunMappingType;
    Instruction68k       *fpBase68k;
    Instruction88k       *fpBase88k;
    unsigned long        fStart68k, fSize68k;
    unsigned long        fStart88k, fSize88k;
    Instruction88k       *fpEntryPoints88k[];    // Variable-sized
    Instruction88k       *fpExitPoints88k[];    // Variable-sized

    void                 Merge (TRunMapping&);
    Instruction88k       FixExitPoints88k (TRunMapping&);
};
```



```

public:
    TRunMapping(Instruction68k *pFirstInstruction68k);
    TRunMapping(Instruction68k **pFirstInstruction68k);
    ~TRunMapping();

    Boolean    In(Instruction68k *);    // Is the address in this run?
    Boolean    In(Instruction88k *);    // Is the address in this run?
    Instruction88k GetEntryPoint88k(Instruction68k *);
    Instruction88k PatchUp(TRunMapping&); // Fix this based on new run

    virtual Boolean IsEqual(const MCollectible*) const;

};

class TRunMappingCollection : public TCollection
{
public:
    TRunMappingSet ();
    ~TRunMappingSet ();
};

// Note: Need to decide the "span" of a segment. Is it part or all of
// a code resource? Then we need a boolean function to determine if a
// segment has been completely translated yet.

```

---

## Summary

We can achieve > 8 MIPS with a straightforward emulation, according to preliminary study and estimation. The variance of this figure is most highly dependent on the cache. This model will require at most 2.25 Mbytes to run, although space optimizations will probably cut this to 900k with little performance hit, and the working set should be only half of that.

With a more space efficient table lookup we could reclaim between 0 and 255k more bytes of space, at a cost of > 2 MIPS.

We can do static translation of 68k to 88k, runtime translation, or both. The two major problems are finding all of the 68k code statically and juggling pieces of 68k and 88k code at runtime. The former is not necessary in a combined scheme in which the latter is accomplished via a set of mappings.

---

## Conclusions

Not surprisingly, the amount of performance we can get out of the system is proportional to the development effort. Our least painful path to 68k is via straightforward emulation. The emulator will be easier to debug than the translator, to be sure. Also, it will be easier to get up and limping for demo purposes, especially on generic 68k code. So emulation is a good bet if we would have to do some demoing early on in development. Perhaps most importantly, it is self-testable. If the emulator handles all 65,536 68k opcodes correctly, then it is theoretically bug free. And testing each opcode can be automated; we can create a "quick brown fox" 68k script. Another important feature is that

straight emulation allows us to control all the 88k Mac code for all time, which would not be as easy with a translator. So, bug fixes would be trivially easy compared to the translator scenario. Finally, the sheer amount of code generated by the user's Mac apps will be smaller, since there will be at most 2 Meg of it (i.e. the 65536 handlers in the emulator). This is important because the size of the 88k will likely be more than quadruple the size of the 68k code.

Of course, the obvious drawback to the emulation is that it runs, with luck, at 8 MIPS, and only at 6 MIPS with reasonable working set requirements. With translated code we can run a Mac application at 50-100 (88k) MIPS. It could be argued that we really do not need to run a Mac program at more than 6 MIPS. This may be true, but consider that we would be wasting 44 MIPS while the Mac program is running. In fact, in the translated case we may only run the Mac program at 6 MIPS when the machine becomes heavily loaded (i.e. fax transmission and phone calls come in, user using voice recognition), but we would not be able to run it at all (or else super-slowly) with emulation if we needed the other 44 MIPS. The key is not the resources the Mac program requires, but rather those that it takes away from the rest of the system.

Doing the expanded emulation rather than a full decoding is sensible for two reasons. First, it is not clear whether the decoding will be even reasonably efficient. Second, the 88k code used in the expanded emulator for a particular 68k opcode is extremely similar to the code with which a translator would replace that opcode. This would allow a much easier migration path from emulator to translator.

Therefore, the best plan for implementation is to hire one engineer to work full time on the emulator. He should work closely with the Blue Adapter engineers in order to integrate it with Macintosh emulation. If he can finish quickly enough then he should begin work on the translator. If not, it will be a welcome feature for release 2.0.