



System II Software Interface Suite

For Matlab, Visual C++, Visual
Basic & Delphi

S232 Software Interface Suite User's Guide – Version 1.0
01 January 1999

Copyright

1998 TDT. All rights reserved.

No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose without the express written permission of TDT.

Licenses and Trademarks

Microsoft, MS-DOS, Windows, Windows 95/98, and Windows NT are registered trademarks of Microsoft Corporation.

Matlab is a registered trademark of The MathWorks Inc.

Delphi is a registered trademark of Inprise

Printed in U. S. A.



**T U C K E R - D A V I S
T E C H N O L O G I E S**

Contents

Preface

Software Philosophy 0-1

Chapter 1 Introduction

What is the S232 Software Interface Suite? 1-1

S232 Software Interface Suite Capabilities 1-1

Hardware Support 1-1

Before You Begin 1-1

What you need 1-1

Installing the Library 1-2

Requirements 1-2

Installation 1-2

Files 1-2

Chapter 2 Programming with the S232 Software Interface Suite

Multitasking with the S232 Software Interface Suite 2-1

The Lock-Unlock Protocol 2-2

Executing a Lock-Unlock Session 2-2

Nested Lock-Unlock Sessions 2-4

Dos and Don'ts 2-5

Defining an Application 2-5

Primary Applications 2-6

Secondary Applications 2-6

Which One To Use? 2-6

Dos and Don'ts 2-6

Building an Application 2-7

Initializing Hardware 2-7

Error Handling 2-8

Terminating the Application 2-9

The Scope of Lock-Unlock Sessions 2-10

Working with Memory 2-10

Programming the PD1 2-11

Dos and Don'ts 2-12

Porting System II Applications to the 32-bit Software Interface Suite 2-12

Incorporating the Basics 2-12

Adding Functionality 2-14

Chapter 3 New Function Reference

APOS/XBDRV Functions Not Supported by the 32-bit API 3-1

APOS/XBDRV Variables Not Supported by the 32-bit API 3-1

APOS/XBDRV Modes of Operation Not Supported by the 32-bit API 3-1

Initialization, Status, and Error Functions 3-2

S2init(dn, mode, apt) 3-2

S2close() 3-3

APactive() 3-3

getS2primary() 3-3

getS2err(err[]) 3-3

Locking and Unlocking Functions 3-4

APlock(mtry, fstart) 3-4

XBlock(mtry, fstart) 3-4

getAPlockstatus() 3-4

getXBlockstatus() 3-4

APunlock(fend) 3-5

XBunlock(fend) 3-5

PD1 Supplemental Functions 3-6

PD1export(varcode, ind[]) 3-6

Chapter 4 Using the Matlab Interface

The Mechanics of the S232 Software Interface Suite for Matlab 4-1

Connecting the Matlab Environment to the S232 Software Interface Suite 4-1

S232 Command Structure 4-2

Return Variables 4-2

String Constants 4-3

Transferring Data 4-3

Error Messages 4-4

Syntax Errors 4-4

System Errors 4-4

String Constant Errors 4-4

Working with Exported Variables for the PD1 4-5

Chapter 5 Matlab Examples

Example 1: A Secondary Dialog Process 5-2

Example 2: A Primary Dialog Process 5-4

Example 3: PD1 Supplemental Dialog Process 5-6

***Chapter 6* Using the Visual C++ Interface**

The Mechanics of Working with the S232 Software Interface Suite for Visual C++ 6-1

Working with Exported Variables for the PD1 6-1

Chapter 7 Visual C++ Examples

Example 1: A Secondary Application 7-2

Example 2: A Primary Application 7-4

Example 3: PD1 Supplemental Application 7-6

***Chapter 8* Using the Visual Basic Interface**

The Mechanics of Working with the S232 Software Interface Suite for Visual Basic 8-1

S232 Commands in Visual Basic 8-1

PD1 Support of HRTF Headers 8-2

Working with Strings 8-2

Working with Exported Variables for the PD1 8-2

***Chapter 9* Visual Basic Examples**

Example 1: A Secondary Application 9-2

Example 2: A Primary Application 9-4

Example 3: PD1 Supplemental Application 9-6

***Chapter 10* Using the Delphi Interface**

The Mechanics of Working with the S232 Software Interface Suite for Delphi 10-1

S232 Commands in Delphi 10-1

PD1 Support of HRTF Headers 10-2

Working with Exported Variables for the PD1 10-2

***Chapter 11* Delphi Examples**

Example 1: A Secondary Application 11-2

Example 2: A Primary Application 11-4

Example 3: PD1 Supplemental Application 11-6

Preface

Software Philosophy

TDT's philosophy on software development is simple. Design a comprehensive software platform that allows the customer to work at a level with which they are most productive. This software solution represent the continuing efforts of TDT to provide you with the latest tools for using System II.

System II includes a extensive high-level Applications Programming Interface (API). With minimal programming, the System II API enables scientists to produce high-quality signal processing applications tailored to meet their specific needs. In complementary fashion, a suite of "turn-key" signal processing applications exist that have a minimal learning curve and facilitate signal generation, presentation, and analysis. The continuing evolution of System II productivity tools provides a broad range of opportunity for the scientist, researcher, or engineer.

With this package it is now possible to use TDT on both Windows 95/98 and Windows NT platforms. Furthermore, the new 32-bit platform allows for multiple applications to access the AP2 Array Processor or XBUS hardware modules and works with the family of SigGen Solutions.

Organization of the Guide

The *S232 Software Interface Suite User's Guide* is divided into five parts:

- Introduction
- S232 for Matlab
- S232 for Visual C++
- S232 for Visual Basic
- S232 for Delphi

Introduction

- *Chapter 1* Introduction
- *Chapter 2* Programming with the S232 Platform
- *Chapter 3* New Function Reference

S232 for Matlab

- *Chapter 4* Using the Matlab Interface
- *Chapter 5* Matlab Illustrative Examples

S232 for Visual C++

- *Chapter 6* Using the Visual C++ Interface
- *Chapter 7* Visual C++ Illustrative Examples

S232 for Visual Basic

- *Chapter 8* Using the Visual Basic Interface
- *Chapter 9* Visual Basic Illustrative Examples

S232 for Delphi

- *Chapter 10* Using the Delphi Interface
- *Chapter 11* Delphi Illustrative Examples

Chapter 1 Introduction

What is the S232 Software Interface Suite?

The S232 Software Interface Suite is a collection of software productivity tools for Tucker-Davis Technologies System II. It was designed to allow access to hardware through a suite of software packages familiar to scientists, researchers, and engineers. The S232 Software Interface Suite serves as a conduit between the 32-bit System II Applications Programming Interface (API) and a number of popular programming environments.

S232 Software Interface Suite Capabilities

The S232 Software Interface Suite affords all the benefits of true 32-bit Windows programming, such as multi-tasking, and facilitates a direct exchange of data with System II hardware.

Hardware Support

S232 supports all System II programmable instrumentation, including any combination of converters, programmable attenuators, waveform generators, cosine switches and programmable filters.

Before You Begin

What you need

See your Microsoft Windows documentation.

- Windows fundamentals
You should be comfortable with Windows basics: starting Windows; using the mouse; manipulating windows; opening, closing, and saving files.

See the System II manual.

- Basic System II Concepts
You should recognize System II concepts including: buffer & DAMA management, stack operations, module commands, etc.

Installing the Library

Requirements

In order to use the S232 Software Interface Suite, you must have the following:

- Microsoft Windows 95/98 or Windows NT
- A monitor with at least VGA resolution graphics. Super VGA (1024 x 768) resolution graphics highly recommended
- TDT's AP2 Array Processor or OP2 Optical Controller
- TDT's System II 32-bit Platform

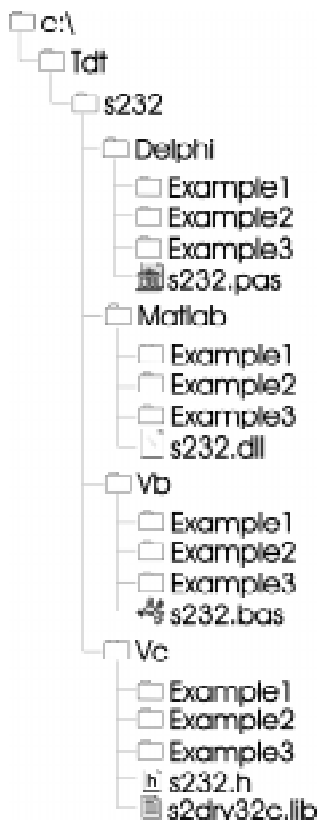
Installation

1. Make sure your TDT hardware (AP2 Array Processor, OP2 Optical Controller, and/or XBUS devices) is installed and functioning properly. Refer to the *System II Installation Guide*.
2. Insert the S232 Software Interface Suite diskette #1 into drive A:.
3. Run **setup.exe** to start the installation program.

Files

When installing System II software, it is recommend that you install the programs to the default folders suggested by the installation program. They will install all files to the standard TDT file structure as shown in the figure at left.

Using the default file structure will make it easier for TDT to support you if you require assistance.



***Chapter 2* Programming with the S232 Software Interface Suite**

The Windows multitasking framework allows for multiple programs to be active at any given time. This provides a more productive environment that affords the user an opportunity to apply several tools towards a single project. The S232 Device Driver & Controller is built within this framework and takes advantage of the multitasking services provided by Windows.

The S232 Device Driver & Controller architecture is described in Chapter 2 of "The System II 32-bit Device Driver & Controller User's Guide." The System II calls form the API that are contained within the DLLs. Application programmers are able to author System II software by using the API in their applications. There is a set of new commands that are introduced during this chapter. As they are presented to the reader they will appear to the left of the text. A quick glance through the document should indicate where that new call is discussed. All new System II commands are detailed in "Chapter 3 New Functions Reference."

Multitasking with the S232 Software Interface Suite

Having multiple applications that work with the same hardware is a powerful concept but not a new one. Anyone who has printed documents from different programs on the same printer has already proven the point. Multitasking with the S232 Software Interface Suite is essentially the same. Multiple applications can work with System II hardware by using an arbitration mechanism. This mechanism is known as the *lock-unlock protocol*.

The Lock-Unlock Protocol

In the Windows environment, multiple applications may be active that use System II resources. The lock-unlock protocol is a control mechanism that provides a method to avoid hardware contention between applications. That method is embodied in a resource *lock*. A lock is a hardware state which temporarily prevents additional programs from accessing hardware. During this time, the application in possession of the lock may work with System II resources without interruption.

A resource lock exists in two forms: AP2 and XBUS. Each resource lock is a status bit that resides on the AP2 card. A program can secure hardware resources by acquiring the resource's lock. The application then has control of the hardware until the resource is *unlocked*. Unlocked hardware resources are available to any System II application. The entire process of locking resources, working with hardware, and unlocking resources is called a *lock-unlock session*. During the lock-unlock session, only the application which has possession of the resource lock is permitted to access that specific hardware. For example, if an application successfully initiates an XBUS lock-unlock session, other applications are prohibited from accessing XBUS hardware.

While this multitasking approach provides a higher level of productivity to the user, it requires an additional layer of software intelligence in System II applications. Applications must be able to handle the possibility that they may request resource locks and not receive them. Applications can query the lock status by using *getAPlockstatus()* or *getXBlockstatus()*.

getAPlockstatus()
getXBlockstatus()

Executing a Lock-Unlock Session

A lock-unlock session is the basic building block in any System II application. A session consists of a *request-for-lock*, a software segment which may contain calls to System II hardware, and a *lock-release*. Any application that wants to access hardware must do so from within a lock-unlock session. A call to hardware from outside of the session generates an error and is denied.

```
APlock(int mtry, int fstart)
XBlock(int mtry, int fstart)
```

The request-for-lock is initiated by the System II functions *APlock()* or *XBlock()*. The lock functions request the resource locks whose state resides on the AP2 card. If the resource locks are available, the application is granted that resource lock and the state is set. If the lock resource is not available, the functions will wait the specified amount of time. If during this request period, the resource lock becomes available, the application is granted the resource lock and the state is changed. If the resource lock never becomes available, the function will “time-out” and return false.

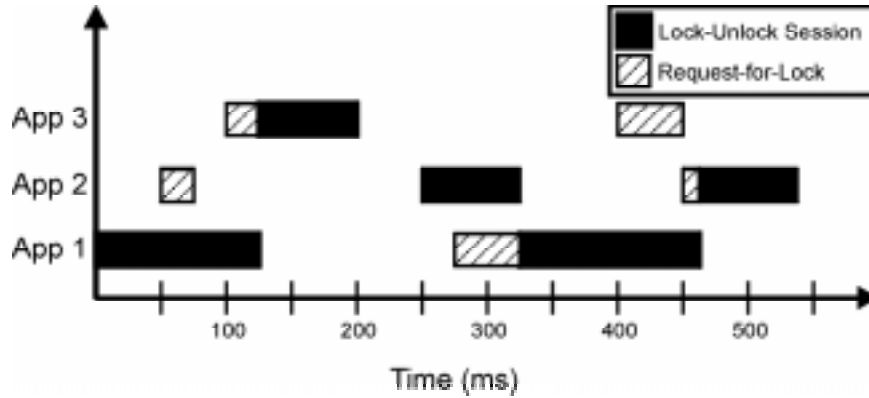
The application should not initiate an AP2 lock-unlock session and make calls to XBUS hardware, or vice versa. Locking the AP2 does not in any way provide the application the ability to make calls to XBUS hardware. If calls to both the AP2 and XBUS are going to be made, the application should request both the AP2 and XBUS locks. The application maintains control of the System II hardware resources until a lock-release is issued.

```
APunlock(int fstart)
XBunlock(int fstart)
```

The lock-release uses the System II functions *APunlock()* or *Xbunlock()*. When the unlock functions are called, the resource lock is released and the lock state is cleared. The hardware resources are now available to any System II application that requests them.

The lock-unlock protocol allows multiple applications to initiate lock-unlock session and share System II resources over different slices of time. The concept is best reinforced with an illustrative example. Consider the case where three applications are sharing the AP2.

At time 0, App1 is in possession of the resource lock and begins its lock-unlock session. During App1's session, App2 makes a request-for-lock. App2 waits the amount of time designated by its mtry variable, 25 ms, until it times-out without being granted the resource lock. App3 makes a request-for-lock and begins to wait the amount of time designated by its mtry variable, 50 ms. After 25 ms App1 completes its session with a lock-release. At this time App3's request-for-lock has not timed-out and his lock-unlock session begins. When App3 ends its session, the resource lock becomes available to any requesting application. After 50 ms of idle time, App2 initiates a request-for-lock and the interaction between the three applications continues.



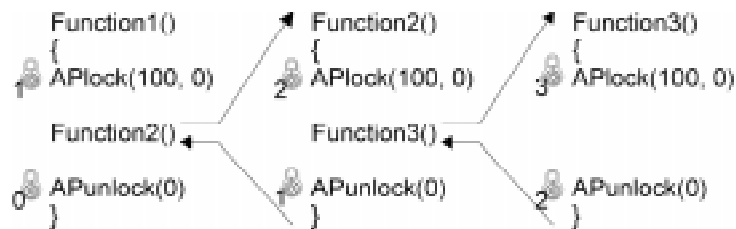
This example should convey some of the important points of working with the lock-unlock protocol. In particular, how lock-unlock sessions allow multiple applications to share System II resources.

Nested Lock-Unlock Sessions

Until now, lock-unlock sessions have only been discussed in the context of hardware control between applications. To simplify application development, the basic concept will be extended to include *nested lock-unlock sessions*. These sessions occur within a single application that already has control of the hardware.

Applications can be quite involved. In an attempt to alleviate some complexity, the programmer may opt for software reusability. To further that goal, functions are often built that support either specific events on their own or support other functions. As an illustration, consider a function that sets application variables based on module settings. Depending on who calls the function, the application or another function, the function may be required to request a hardware lock. Rather than require the programmer to build in a series of conditional statements that checks the state of the locks, the function is allowed to simply request another lock. Because the application knows that it is in possession of the hardware lock, all additional lock requests are automatically nested.

The application knows if a lock has been granted by the state of the *lock counter*. The lock counter will be zero until the first hardware lock is requested and granted. After receiving the first lock, the lock state is set on the AP2 card and the lock counter is incremented to 1. Subsequent lock requests increment the lock counter. Each time an unlock request is made, the lock counter is decremented. It is critical to make sure that a corresponding unlock is called each time a lock request is granted. When the lock counter is reduced to zero, that hardware resource becomes available to any requesting application. The following is an example that uses APlock() to illustrate how lock requests made by the application interacts with the lock counter.



The lock counter cannot be directly accessed. However, it can be forced to zero by the locking and unlocking commands. This presents an advantage to the programmer. The programmer can force the lock counter to zero by setting the fstart or fend variables in the lock and unlock calls to 1. The individual S2 commands are described in “Chapter 3 New Function Reference.”

Dos and Don'ts

- Do not terminate an application without unlocking resources.
- Do issue a corresponding unlock for each successive lock.
- Do not hold locks indefinitely. Instead, hold locks only when accessing hardware.
- Do not initiate an AP lock-unlock session and make calls to XBUS hardware or vice versa.

Defining an Application

The concept of multitasking introduces the question of hardware ownership. Applications can access System II hardware resources during their lock-unlock sessions. But what happens to those hardware resources between sessions depends on an application's classification. An application may be classified as either a *primary* or *secondary*. The classification implies certain ownership privileges of AP2 resources. The notion of application classification is not relevant if an OP2 is used instead of an AP2.

Primary Applications

A primary application may retain AP2 resources between lock-unlock sessions. The S232 platform allows a single primary application to be active at any time. Primary application assume that their System II *state* remains intact between lock-unlock sessions. The state of an application includes all AP2 buffers that have been allocated on the stack or in DAMA and their data.

When a primary application is launched, the initialization request is compared against the primary active state on the AP2 card. If no other primary application is running, the application is initialized as the primary application and the primary active state is set. If the primary active state is already set, indicating that a primary application is currently active, the initialization request is denied. Every primary application should consider the fact that another primary applications may be active. An alternative approach is to check the status of the primary active bit by using the *getS2primary()* command.

`getS2primary()`

Secondary Applications

The multitasking capabilities of the S232 platform are embodied by secondary applications. Contrary to the primary application, there is not a limit to the number of secondary applications that can be active at any given time. This affords the applications developer with the ability to build a suite of tools that either support a primary application or perform specific functions. Secondary applications can allocate AP2 buffers, but they cannot assume that those buffers will be present at the time of their next lock-unlock session.

Which One To Use?

This question is often answered by examining the requirements of the application. The only fundamental rule is the following: the application must be initialize as a secondary application if it will be used in conjunction with a primary application. Many programs could be written as secondary applications.

Dos and Don'ts

- Do not attempt to execute two primary applications at the same time.
- Do not allow secondary applications to corrupt the state of the system.

- Do not attempt to access the same XBUS module at the same time from two or more applications, either primary or secondary.

Building an Application

A System II application must include a minimal set of services to function correctly in the S232 platform. Basic services include application initialization, error handling, and termination. By providing these services, the application can work with System II hardware. After the minimal services have been provided, the application can be enhanced by including some additional concepts. The idea of buffer management and the scope of lock-unlock sessions will speak to the applications multitasking capabilities.

Initializing Hardware

`S2init(dn, mode, apt)`

Every application should have an initialization section. This is where the application initializes System II hardware using the *S2init()* command. *S2init()* supercedes the *apinit()* and *XB1init()* calls. The *dn* argument indicates with which logical device, typically 0, the application is working. See Chapter 4 of “The System II 32-bit Controller User’s Guide” for more information on dual-card systems.

The last two arguments indicate how the application will work with the card. The *apt* argument defines the overall system timeout. Should any System II command wait longer than the time specified by *apt*, an error will result. This should typically be set to around 5000, five seconds, or greater. This will provide your application enough time to make the call, the hardware to process the call, and any additional Windows latency to pass.

The second argument, *mode*, defines the classification of the application. Depending on which mode is chosen a hard-wire reset may be issued. This may not be desirable depending on the presence of other System II applications.

The following modes are available to the System II applications programmer.

INIT_PRIMARY mode will attempt to initialize a primary application. If a primary application is already running, this *S2init()* call will fail. If the *S2init()* call is successful, it will return 1. The caller becomes the primary application and asserts a hardware reset. This classification assumes that its state will be intact upon the next lock-unlock session.

INIT_SECONDARY mode will attempt to initialize a secondary application. If the `S2init()` call is successful, it will return a 2. Secondary applications do not issue a hardware reset. Secondary applications cannot assume that its state will be intact between lock-unlock session.

INIT_EITHER mode will first attempt to initialize a primary application. If a primary application is already running, the same `S2init()` call will attempt to initialize a secondary application. If a primary application was successfully initialized, `S2init()` will return a 1; if a secondary application was successfully initialized, `S2init()` will return a 2. This call is very useful in avoiding the error presented when a primary application is already running.

INIT_FORCEPRIM mode will initialize a primary application. If a primary application is already running, its primary status is revoked and the caller becomes the primary application. The `S2init()` will return a 1 and assert a hardware reset. Use this mode with extreme caution when multitasking is employed.

INIT_OP2 mode will initialize an application using the OP2 Optical Controller. If the `S2init()` call is successful, it will return a 5. OP2 applications do not assert a hardware reset.

INIT_OP2RESET mode will initialize an application using the OP2 Optical Controller. If the `S2init()` call is successful, it will return a 6. OP2RESET will assert a hardware reset.

Error Handling

The System II 32-bit platform supports error trapping. This provides the applications programmer with the opportunity to service any errors that may occur.

`getS2err(emess[])`

Error handling has been simplified to a single System II command, `getS2err()`. When an error occurs, the state of the System II hardware changes to error pending. This state persists and prevents all other System II calls from being processed. The `getS2err()` call is used to service that error and allow program flow to continue.

The function `getS2err()` can be used in two ways. The method chosen depends upon the argument passed to `getS2err()`. The argument is a pointer to a character string. If the argument is a null pointer, 0, then `getS2err()` only checks for an error, it does not service the error. As a result, the error pending state will persist and prevent subsequent System II commands from being processed. If the argument is a pointer to a valid string `getS2err()` checks for an error, clears the error if pending, and fills the string pointed to by the argument with the error message. In both methods, the function will return 0 if no error, 1 for an AP2 error, and 2 for an XBUS error. **Note: `getS2err()` does not require a resource lock before it can be called.**

Consider the following example in which `MyFunction()` is called without the AP2 lock. This example introduces the complexity of pending errors. In this example, `dpush()` causes an error that prevents `flat()` from being processed. Also notice that the lock was granted even though an error was pending.

```
MyFunction()
{
    char emess[256];        //Message Buffer
    dpush(1024)();         //Error, did not have
                           // the AP2 lock!

    if(APlock(100, 0))     //Lock granted
        AfxMessageBox('Cannot lock the AP2');
    flat();                //Function ignored due
                           // to pending error

    if(getS2err(emess))    //dpush() Error cleared
        AfxMessageBox(emess);
    APunlock(0);          //Resource unlocked
}
```

The programmer should at a minimum check for errors before releasing the resource locks. Additional error checking will afford the opportunity to service any errors that may occur and prevent critical instructions from being processed.

Terminating the Application

When the application is complete, there is often an opportunity to perform any “housecleaning” activities. This is the place where memory is released, files are written, etc. In System II applications, this is where it releases its self from the S232 platform.

`S2close()`

Every System II applications should include the `S2close()` command. Currently only primary applications actually require the use of `S2close()`. However, for future compatibility it is recommended that every System II application include the `S2close()` call.

The Scope of Lock-Unlock Sessions

Lock-unlock sessions are the building blocks of System II applications. It is during lock-unlock sessions that applications are required to work with System II hardware. The scope of a session includes all the calls between the lock-request and the lock-release.

The largest scope would be one that begins in the application's initialization and ends with the application's termination. This scope would prevent any additional applications from accessing System II hardware. This may be acceptable for the user who never intends to run multiple System II applications concurrently.

The session scope may be reduced to the function level or even smaller. The important point is that the session scope should be focused enough to allow for multitasking capabilities, but not focused to the point that it becomes a burden. The use of lock-unlock sessions should help the applications programmer and not hinder the application's development.

Working with Memory

The AP2 card contains memory that applications use to hold data. There are very few restrictions for memory usage. The cardinal rule is that primary applications must find their state intact between lock-unlock sessions. The state of the application includes the memory allocated for that application and its data. Secondary applications can allocate memory but must contend with the possibility that the memory may not be still allocated during the next lock-unlock session.

The memory that resides on the AP2 card is logically divided into two parts: Stack and DAMA. The stack is organized in a First-In First-Out (FIFO) manner. Buffers are allocated, pushed, on top of the stack. This action causes all current stack buffers to be pushed further down in the stack structure. The DAMA is organized into independent buffers that have their own unique id number. Buffers are allocated in DAMA without changing the properties of previously established buffers.

The DAMA buffers are requested from the AP2 OS by the application in one of two ways. The application can request a buffer with a specific id number, `allotx(dbn, length)` where x is either 16 or f. Or, the application can request a buffer and allow the AP2 OS to designate the id number, `_allotx(length)` where x is either 16 or f. The later method avoids the possibility that two System II applications may choose the same id number. If the former were chosen, the potential exists that two applications would attempt to allocate the same buffer and crash the AP2 OS.

Once the application is finished using the buffers, their memory should be freed for other applications. As with the allocation process, the de-allocation process can be approached in two different ways. The best way is to use the `deallot()` and `drop()` calls. These allow the AP2 OS to reallocate the memory associated with a specific buffer. The alternate method de-allocates all memory by way of the `trash()` and `dropall()` calls. In using these calls, a secondary application could potentially free the memory of a primary application and violate memory usage's cardinal rule. From a multitasking viewpoint, the primary application should choose the former method as well to allow all secondary applications to maintain their state even if the primary application is terminated.

Programming the PD1

The PD1 is a high performance signal processing device that is used for performing research in areas such as binaural hearing sound localization, 3D sound generation, and architectural acoustics. At the heart of the PD1 is a hardware module called the Real-time Router. This specialized piece of hardware uses software techniques to allow the programmer to implement a signal processing circuit of their design.

The software uses specific programming variables in System II calls that have been initialized by the S232 DLLs. These variables are then used in subsequent calls when programming the PD1. Consequently, those variables have been exported by the S232 DLLs. Some languages support the use of exported variables and some do not. As a result, a new System II call was added to the API in order to support those languages that do not allow the use of exported variables. The System II call `PD1export()` provides a means to return the contents of the exported variables, rather than the variables themselves. This allows the programmer to build local variables that are replicas of the exported variables that they cannot access. These replicas can then be passed to the System II API as though they were the original variables.

```
PD1export(int var, int ind[])
```

Dos and Don'ts

- Do use long system timeout values less than 5000 ms.
- Do perform error handling often. It will allow the application to recover from errors and not crash the system.
- Do not terminate your applications without S2close().
- Do consider the scope of lock-unlock sessions. This has implications on the applications multitasking capabilities.
- Do not use allotf() or allot16() commands to allocate specific DAMA buffers. Use _allotf() and _allot16().
- Do not use trash() or dropall() with secondary applications. Use deallot() and drop() to delete individual buffers. Remember, the primary applications may have left items in memory.
- Do not attempt to send PD1 variables to the System II API without initializing them.

Porting System II Applications to the 32-bit Software Interface Suite

Most users of the S232 Software Interface Suite will have written applications for System II hardware already. Since the S232 API has not changed, porting the software can be relatively easy. The programmer only needs to modify a few services to have the applications running under the new platform.

Incorporating the Basics

The basics include the applications initialization, error handling, and termination. The initialization and error handling were included more than likely in the applications. The initialization section often appeared similar to the following.

```
if(!XB1init(USE_DOS)){
    printf("\n\n\nXBUS Error!!!\n\n\n");
    exit(0);
}
if(!apinit(APb)){
    printf("\n\n\nAP2 Error!!!\n\n\n");
    exit(0);
}
```

This could be replaced with the following which is a simple initialization of all System II hardware. At this point, the hardware locks could be requested as well, initiating a lock-unlock session. It's that easy!

```
if(!S2init(0, INIT_SECONDARY, 5000)){
    AfxMessageBox("Cannot initialize the app");
    exit(0);
}
if(!APlock(200, 0)){
    AfxMessageBox("Did not receive the AP lock");
    exit(0);
}
if(!XBlock(200, 0)){
    AfxMessageBox("Did not receive the XB lock");
    APunlock(0); // The AP2 is locked!
    exit(0);
}
```

Error handling has also been simplified. Previously, the programmer would have been responsible for checking both AP2 errors and XBUS errors. The error checking routine often appeared similar to the following which does not include checking the XBUS for errors.

```
if(!ap_emode){
    printf("\n\n\nAP2 Error!!! \n\n\n");
    exit(0);
}
for(int i=0; i<5; i++)
    printf("Error: %s\n", ap_err[i]);
ap_emode = 1;
```

This can be replaced with the following which does check both the AP2 and XBUS errors.

```
char emess[256];
if(getS2err(emess))
    AfxMessageBox(emess);
```

The last piece, terminating the application, was not required before. At the point where the application exits, include the following.

```
S2close();
```

You can explicitly make the calls to unlock hardware before calling `S2close()`, but those commands are made from inside the `S2close()` procedure for you.

Adding Functionality

Once the application is successfully compiled, programmers can begin to build in additional functionality as was previously described. The scope of the lock-unlock sessions can be shortened enough to provide the multitasking capabilities for other applications to share System II resources. Memory usage can be revamped to allow the AP2 OS to control the id numbers. Checks for AP2 activity with the `APactive()` call, error handling, and lock states can be included. These all allow for another level of flexibility and performance.

`APactive()`

Chapter 3 New Function Reference

APOS/XBDRV Functions Not Supported by the 32-bit API

- apinit(dev), refer to S2init(dn, mode, apt)
- ap_select(dev), refer to S2init(dn, mode, apt)
- ap_present(dev), refer to S2init(dn, mode, apt)
- ap_active(dev), refer to APActive()

APOS/XBDRV Variables Not Supported by the 32-bit API

- ap_eflag, refer to getS2err(emess[])
- ap_err[], refer to getS2err(emess[])
- xb_eflag, refer to getS2err(emess[])
- xb_err[], refer to getS2err(emess[])

APOS/XBDRV Modes of Operation Not Supported by the 32-bit API

- COM ports are not supported. Refer to information regarding the OP2 Optical Controller in the "System II 32-bit Controller."
- AP2 macros are not supported.

Initialization, Status, and Error Functions

S2init(dn, mode, apt)

Prototype	int S2init(int dn, int mode, int apt);												
Operation	Initializes the specified AP2 and identifies the XBUS modules present in the IDed caddies. The apt value defines the global timeout for the application. Should any System II call take longer than the timeout, an error may result. S2init() supercedes apinit() and Xbinit(). Readers are referred to those calls and Chapter 2 for more information.												
Arguments	<p><i>dn</i> selects the logical device number</p> <p><i>mode</i> selects the application mode</p> <table> <tr> <td>INIT_PRIMARY</td> <td>1</td> </tr> <tr> <td>INIT_SECONDARY</td> <td>2</td> </tr> <tr> <td>INIT_EITHER</td> <td>3</td> </tr> <tr> <td>INIT_FORCEPRIM</td> <td>4</td> </tr> <tr> <td>INIT_OP2</td> <td>5</td> </tr> <tr> <td>INIT_OP2RESET</td> <td>6</td> </tr> </table> <p><i>apt</i> selects the global system timeout in ms</p>	INIT_PRIMARY	1	INIT_SECONDARY	2	INIT_EITHER	3	INIT_FORCEPRIM	4	INIT_OP2	5	INIT_OP2RESET	6
INIT_PRIMARY	1												
INIT_SECONDARY	2												
INIT_EITHER	3												
INIT_FORCEPRIM	4												
INIT_OP2	5												
INIT_OP2RESET	6												
Returns	The mode value of the application successfully initialized or 0 if unsuccessful.												
Example	The following code initializes the AP2 and XBUS in a Primary application. The global timeout is set to 5000 ms. If the initialization is unsuccessful, a dialog window will display an error message.												

```

if(!S2init(0, INIT_PRIMARY, 5000))
{
    AfxMessageBox("Unable to
initialize a primary app.");
    exit(0);
}

```

S2close()

Prototype void S2close(void);
Operation Closes a S232 application. Use this command at the end of all System II applications before they are terminated. This call contains APunlock(1) and XBunlock(1).

APactive()

Prototype int APactive(void);
Operation Checks the status of the AP2.
Returns 0 if the AP2 is ready
 1 if the AP2 is busy
 99 if the error flag is pending
Comments The AP2 device under query is the current logical device. This command supercedes ap_active() and readers are referred to that call for more information.

getS2primary()

Prototype int getS2primary(void);
Operation Returns true if a primary application is running.
Returns 0 no primary application running
 1 a primary application is running

getS2err(err[])

Prototype int getS2err(char err[]);
Operation Checks for an AP2 or XBUS error, resets the error state, and copies an error message to the string pointed to by err[]. If the argument is a null pointer, 0, the error flags are checked but not reset. The reader is referred to Chapter 2 for more detail.
Arguments err[] A pointer to a character array.
Returns 0 no error
 1 APOS error
 2 XBUS error
Example The following code checks for and displays an error if one exists.

```
char emess[255];
    if (getS2err(emess)!=0)
        AfxMessageBox(emess);
```

Locking and Unlocking Functions

APlock(mtry, fstart)

Prototype	int APlock(int mtry, int fstart);
Operation	Issues an AP2 request-for-lock for a period of <i>mtry</i> milliseconds. If <i>fstart</i> is 0 and the current application already has the AP2 lock, the lock counter will be incremented by 1. The reader is referred to Chapter 2 for more detail.
Arguments	<i>mtry</i> ms to try locking before returning 0 <i>fstart</i> 1 will reset the lock counter 0 will increment the lock counter
Returns	0 if the AP2 lock cannot be acquired 1 if the AP2 lock is acquired

XBlock(mtry, fstart)

Prototype	int XBlock(int mtry, int fstart);
Operation	Issues an XBUS request-for-lock for a period of <i>mtry</i> milliseconds. If <i>fstart</i> is 0 and the current application already has the XBUS lock, the lock counter will be incremented by 1. The reader is referred to Chapter 2 for more detail.
Arguments	<i>mtry</i> ms to try locking before returning 0 <i>fstart</i> 1 will reset lock counter for this app. 0 will increment lock counter for this app
Returns	0 if the XBUS lock cannot be acquired. 1 if the XBUS lock is acquired.

getAPlockstatus()

Prototype	int getAPlockstatus(void);
Operation	Test the state of the AP lock.
Returns	0 if the AP2 lock is not available 1 if the AP2 is lock is available

getXBlockstatus()

Prototype	int getXBlockstatus(void);
Operation	Tests the state of the XBUS lock.
Returns	0 if the XBUS lock is not available 1 if the AP2 is lock is available

APunlock(fend)

Prototype void APunlock(int fend);

Operation Releases the AP2 lock. If fend is 0, the lock counter is non zero, and the current application has the AP2 lock, the lock counter will be decremented by 1. The reader is referred to Chapter 2 for more detail.

Arguments *fend* 1 will reset the lock counter and release the AP2 lock
0 will decrement the lock counter

XBunlock(fend)

Prototype void XBunlock(int fend);

Operation Releases the XBUS lock. If fend is 0, the lock counter is non zero, and the current application has the XBUS lock, the lock counter will be decremented by 1. The reader is referred to Chapter 2 for more detail.

Arguments *fend* 1 will reset the lock counter and release the XBUS lock
0 will decrement the lock counter

PD1 Supplemental Functions

PD1export(varcode, ind[])

Prototype int PD1export(int varcpde, int ind[]);

Operation Provides a means to access program variable contents for languages that cannot use exported variables.

Arguments *varcode* variable index number

xV	0x01
xDSPID	0x02
xDSPIN	0x03
xDSPINL	0x04
xDSPINR	0x05
xDSPOUT	0x06
xDSPOUTL	0x07
xDSPOUTR	0x08
xCOEF	0x09
xDELIN	0x0A
xDELOUT	0x0B
xTAP	0x0C
xDAC	0x0D
xADC	0x0E
xIB	0x0F
xOB	0x10
xIREG	0x11

indicies array of indicies of the variable contents.

Returns The contents of the variables.

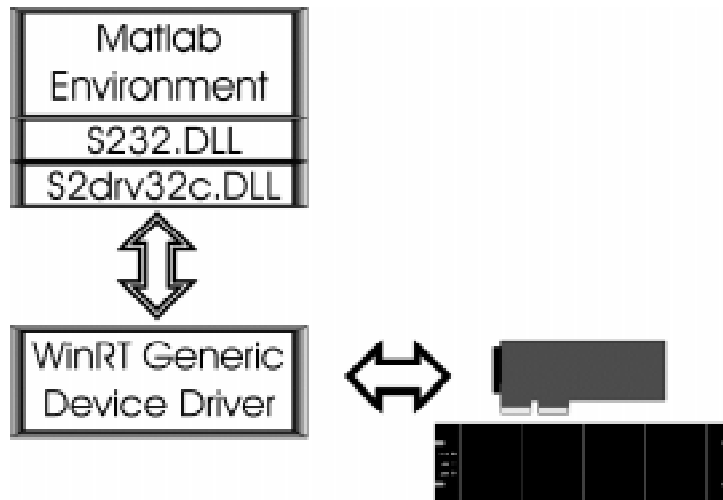
Example The following exports the contents of the DAC variable. Prior to calling this command, the user must have called PD1clear(1) to initialize the variables

```
int TAPadd[4][32];
int i, j, add[2];
PD1clear(1); // Must call first
for(i=0; i<4; i++)
{
    add[1] = i;
    for(j=0; j<32; j++)
    {
        add[2] = j;
        TAPadd[i][j]=PD1export(xTAP, &add);
    }
}
```

Chapter 4 Using the Matlab Interface

The Mechanics of the S232 Software Interface Suite for Matlab

Matlab provides a method to interface its environment with external software routines through DLLs. The S232 Software Interface Suite connects the System II API to the Matlab environment. The inclusion of a second layer DLL, S232, to the System II 32-bit architecture is shown below.



Connecting the Matlab Environment to the S232 Software Interface Suite

Matlab must know the directory of S232.DLL to access its contents. Matlab's path statement contains a list of the directories it searches. The steps to modify the path statement are shown below.



1. Click the Path Browser button.
2. Click the Add to Path button.
3. Select Add to Back
4. Find the directory that contains the S232.DLL file. The default is C:\Tdt\S232\Matlab.
5. Select the OK button
6. Select the Close button.

S232 Command Structure

The S232 command structure was adapted from the C function prototype to fit the Matlab environment. The general format is the following.

```
[return list] = S232('S2 Call', argument list);
```

Any return variables are enclosed in square brackets to the left of the Matlab command. The System II call name is bracketed by single quotations inside the Matlab S232 command. The argument variable list is separated from the System II call name by a comma. The S232 Software Interface Suite automatically checks to ensure that the correct number of arguments are passed into the Matlab S232 Command. The S232 Software Interface Suite will notify the user if the number of arguments is not correct.

```
Correct:      S232('PA4atten', 1, 20.7)
```

```
Incorrect:   S232('PA4atten', 20.7)
```

```
Error Message: ??? Proper Usage: [] =  
S232('PA4atten', int din, float atten)
```

Return Variables

C is limited in the number of return variables, one at most, that can be implemented with any function as illustrated by the following C example:

```
A = MyCFunction(B, C, D, E, F);
```

As a result, if multiple arguments are to be returned, the traditional solution is to pass pointers to locations where the contents can be returned. This could be done using the & operator as follows:

```
A = MyCFunction(B, &C, &D, &E, F);
```

In contrast, Matlab functions can and often do have multiple return elements. An interpretation of the above C function would lend itself to the following Matlab command:

```
[A, C, D, E] = MyMatlabFunction(B, F);
```

S232 utilizes the same method to implement the System II API calls as illustrated in the following example.

```
C :      void HTIreadAER(din, &az, &el, &roll);  
Matlab: [az,el,roll] = S232('HTIreadAER', din);
```

String Constants

Matlab does not currently support a #define statement as is implemented in most high-level languages. As a result, it was not possible to include system constants without cluttering the Matlab workspace with dozens of variables. Instead, S232 was designed to receive strings that will be interpreted as #define equates. Each string constant must be enclosed with single quotes. An example of which would be the following.

```
S232('DAmode', 1, 'DAC1');
```

Internally, S232 will replace DAC1 with the integer 1 and continue the command interpretation. While this can be seen as a convenience in most cases, it does not lend itself to the often used arithmetic combination of two or more #defines. The following, DAC1+ADC1, would be caught by Matlab as an undefined variable, and 'DAC1+ADC1' would be caught by S232 as an undefined string constant. Instead you must calculate the appropriate value as described in the System II Manual: ADC1 = 4, DAC1 = 1, ADC1+DAC1 = 5.

```
Matlab Error: S232('DAmode', 1, DAC1+ADC1 );
S232 Error:   S232('DAmode', 1, 'DAC1+ADC1');
Correct:      S232('DAmode', 1, 5);
```

Transferring Data

The S232 Software Interface Suite facilitates the direct exchange of data between the AP2 card and the Matlab environment. This exchange is made using the System II commands that previously moved data to and from PC memory.

```
C:          void popf(buf);
Matlab:     buf = S232('popf');
C:          void pushf(buf, npts);
Matlab:     S232('pushf', buf, npts);
C:          void pop16(buf);
Matlab:     buf = S232('pop16');
C:          void push16(buf, npts);
Matlab:     S232('push16', buf, npts);
```

Error Messages

Error handling is handled primarily by the S232.DLL. It will report errors such as attempting to access XBUS hardware without a lock. However, the programmer should check for errors periodically. The following is an example.

```
EDU> s232('PA4atten', 1, 20.7)
??? XBUS Error:
Device: PA4[1]
Call: PA4atten(...)
Message: Attempting to access XBUS without
LOCK.
```

Syntax Errors

Syntax errors occur as a result of a mistake in the structure of the commands and originates from S232. This can occur in a variety of forms, the most common two are the following.

- Incomplete argument list or too many arguments.
The call will report the error statement “Proper Usage” followed by the correct S232 command.
- Mistyped command name
The call will report “System II command not found.”

System Errors

System Errors occur as a result of a System II Error and originates from the System II operating system (OS). The error message is simply passed through S232 back to the Matlab command window.

- For example, requesting to control a device that isn't present.
The call will report the error statement generated by the System II OS such as “Message: Non-present XBUS device referenced.”

String Constant Errors

Unknown String Constant Errors occur as a result of a string match that could not be made. The error message originates from S232.

- For example, including the following string 'DAC1+ADC1'.
The call will report the error statement “Constant String Define not found.”

Working with Exported Variables for the PD1

The underlying framework for Matlab interface is the C programming language. As a result, it supports both direct and indirect access of exported variables for the PD1.

Direct access of exported variables can be performed as follows.

```
EDU> s232('ADC', 1)
ans = 2065
```

Indirect access of the exported variables can be performed as follows by using the PD1export command.

```
EDU> s232('PD1export', 'xADC', [1])
ans = 2065
```


Chapter 5 Matlab Examples

The examples in this chapter familiarize the user with the structure, definition, and implementation of the S232 Software Interface Suite. Although basic in nature, they assume a familiarity with the TDT System II software interface.

The user is encouraged to try all examples. The relationship of a primary and secondary application is illustrated in using Examples 1 & 2 together. *However, because Matlab creates dialogs and not stand-alone applications, the user will have to execute another Matlab command window. Example 1 and Example 2 must be run from different command windows or the dialogs will not work together correctly.*

Example 3 shows how a simple PD1 application could be written.

Example 1: A Secondary Dialog Process

Example 2: A Primary Dialog Process

Example 3: A PD1 Supplemental Dialog Process

Example 1: A Secondary Dialog Process

This exercise implements a dialog window. The application reports the length of a specific buffer.

Files of Interest

- Example1.m Matlab script file for the dialog box
- Ex1OnCreate.m Matlab script file for initializing the dialog
- Ex1OnGo.m Matlab script file for the Go! procedure

Required Hardware

- AP2 Array Processor

Running the Application

At the Matlab prompt type “Example1” and Enter.

Points of Interest

This bulk of this code was generated by the GUI wizard, guide.m, but there are two important areas to examine.

- Dialog initialization
- What happens when OK button is pressed?

Application initialization

The initialization of the dialog is performed when the window is created. The Matlab script file Ex1OnCreate.m is executed.

```
if(S232('S2init', 0, 'INIT_SECONDARY', 1000)==0)
    disp('Cannot initialize a secondary application');
end
```

What happens when OK button is pressed?

The Ex1OnGo function gets called each time the button is pressed. Before any System II calls can be made, an AP lock is requested. If successful, the routine checks for the presence of the buffer and copies that buffer on the stack. The length is determined and the buffer is dropped to prevent data being left on the stack. The display is set with the buffer length or “???” if no buffer is present. All of the error handling is performed with getS2err.

```
hDisplay = findobj('Tag', 'Display');
if(S232('APlock', 100, 0) == 0),
    disp('Cannot aquire an AP lock')
else,
    if S232('getaddr', 1)
        S232('qpush16', 1);
        BufferLength = S232('topsize');
        set(hDisplay, 'String', strcat('Buffer
            Length is : ',
            num2str(BufferLength)))
        S232('drop');
    else
        set(hDisplay, 'String', 'Buffer Length is
            : ???')
    end
    end
    S232('APunlock', 0)
end
```

Example 2: A Primary Dialog Process

This exercise builds a dialog window. The Go! button builds two stimuli and presents them using generic DA calls. See the previous information on working with Example 1 & 2 together.

Files of Interest

- Example2.m Matlab script file for the dialog box
- Ex2OnCreate.m Matlab script file for initializing the dialog
- Ex2OnExit.m Matlab script file for exiting the dialog
- Ex2OnGo.m Matlab script file for the Go! procedure

Required Hardware

- AP2 Array Processor
- D/A Digital-to-analog converter

Running the Application

At the Matlab prompt type "Example2" and Enter.

Points of Interest

This bulk of this code was generated by the GUI Wizard, guide, but there are three important areas to examine.

- Dialog initialization
- What happens when the Go button is clicked?
- Terminating the dialog

Dialog initialization

The Matlab script file Ex1OnCreate.m "initializes" the dialog is performed when the window is created. The buffers are allocated and as always, error checking is performed after calls to hardware are made.

```
if(S232('S2init', 0, 'INIT_PRIMARY', 1000)==0),
    disp('Cannot initialize a primary process')
else,
    if(S232('APlock', 100, 0)==1),
        SRATE = 20;
        NPTS = 10000;
        BUF1 = S232('_allot16', NPTS);
        BUF2 = S232('_allot16', NPTS);
        S232('APunlock', 0)
    else,
        disp('Cannot allocate buffers')
    end
end
```

What happens when the Go button is clicked?

The Ex2OnGo function is called and the stimuli are built. The tone stimulus is created on the stack while the noise stimulus is created by the Ex2OnGo routine. Using the rand function, the noise pip is built with a maximum value of 32000. The data is then transferred to the AP2 card by using the System II call, push16. Take note that the hardware locks are requested and calls to hardware are dependent on receiving those locks. As always, error checking is performed whenever calls to hardware are made.

```
% Build stimulus for BUF2
buf2 = 32000.*(2.*rand(1,NPTS)-1);
S232('push16', buf2, length(buf2))
S232('qwind', 2.0, SRATE);
S232('qpop16', BUF2);
```

Terminating the dialog

In order to terminate the dialog, some System II housecleaning must be performed. Specifically, all allotted buffers must be de-allocated. To ensure that this is done properly, a while loop informs the user that an AP lock could not be acquired and therefore the application cannot be fully terminated. After the lock is received, the buffers are de-allotted and the program ends with the obligatory S2Close().

```
S232('APlock', 100, 1);
S232('deallot', BUF1);
S232('deallot', BUF2);
S232('S2close');
hDb = gcf;
close(hDb)
```

Example 3: PD1 Supplemental Dialog Process

This exercise implements a dialog window. The Go! button implements a PD1 circuit.

Files of Interest

- Example3.m Matlab script file for the dialog box
- Ex2OnCreate.m Matlab script file for initializing the dialog
- Ex2OnExit.m Matlab script file for exiting the dialog
- Ex3OnGo.m Matlab script file for the Go! procedure
- Ex3OnGoB.m Alternate Matlab script file for the Go! procedure

Required Hardware

- AP2 Array Processor
- PD1 Power SDac

Running the Application

At the Matlab prompt type “Example3” and Enter.

Points of Interest

This bulk of this code was generated by the Application Wizard, but there are two important areas to examine.

- Application Initialization
- What happens when the Go button is clicked?
- An alternate approach using PD1export
- Terminating the application

Application initialization

The initialization of the application is performed by Ex3OnCreate when the window is created.

```
if(S232('S2init', 0, 'INIT_PRIMARY', 1000)==0)
    disp('Cannot initialize a primary
        application');
end
```

What happens when the Go button is clicked?

The Ex3OnGo function is called and the PD1circuit is routed. The circuit includes a delays, filtering, and the summing of multiple inputs. The circuit remains active until the message box is clicked. In general, the most obvious results appear with a broadband noise source. The PD1export() command could have been used for the exported variables, but is not the most direct approach for this language.

An alternate approach using PD1export

As was previously mentioned, the underlying structure of the S232.DLL is C. As a result, Matlab has the luxury, of two choices to support the building of PD1 routing schematics. An excerpt of the alternate approach, using the PD1export command, is included below. While it may seem it requires no additional work, it does make the code easier to read.

```
% Step-3a
ADC = s232('PD1export', 'ADCEXP', 0);
DAC = s232('PD1export', 'DACEXP', 0);
DELin = s232('PD1export', 'DELINEXP', 0);
for i = 1:2
    DSPin(i) = s232('PD1export', 'DSPINEXP', i-1);
    DSPout(i) = s232('PD1export', 'DSPOUTEXP', i-1);
end
for i = 1:3
    DELout(i,1) = s232('PD1export', 'DELOUTEXP', [i-1, 0]);
end
```

Terminating the application

The program ends with the required S2Close().

Chapter 6 Using the Visual C++ Interface

The Mechanics of Working with the S232 Software Interface Suite for Visual C++

The application must know the procedure declaration and the procedure location in the DLL before the routine can be accessed. The information that the application requires is present in the API header file and library. The header file, S232.h, contains all the procedure definitions. The library file, S2drv32c.lib, provides the DLL entry points.

If the header file is not included, the compiler will not recognize the procedure call. If the library is not included, the linker will not be able to resolve what it assumes to be an external reference.

Including the files is relatively straight forward. Development environments have lists of directories that are searched for different file types, e.g. headers(.h) or libraries (lib). In these search paths include the C:\Tdt\S232\Vc directory.

The individual files, s232.h and s2drv32c.lib, may also be included in the project directly.

Working with Exported Variables for the PD1

The Visual C++ interface supports both direct and indirect access of exported variables for the PD1. Direct access of exported variables can be performed as follows.

```
ADC[1];
```

Indirect access of the exported variables can be performed as follows by using the PD1export command.

```
int indicies = 1  
PD1export(xADC, &indicies);
```


Chapter 7 Visual C++ Examples

The examples in this chapter familiarize the user with the structure, definition, and implementation of the S232 Software Interface Suite. Although basic in nature, they assume a familiarity with the TDT System II software interface.

The user is encouraged to try all examples. The relationship of a primary and secondary application is illustrated in using Examples 1 & 2 together. Example 3 shows how a simple PD1 application could be written.

Example 1: A Secondary Application

Example 2: A Primary Application

Example 3: A PD1 Supplemental Application

Example 1: A Secondary Application

This exercise implements a dialog window. The application has a timer that upon expiring reports the length of a specific buffer.

File of Interest

- Example1Dlg.cpp C++ file for the dialog box

Required Hardware

- AP2 Array Processor

Running the Application

Either double click the Example1.exe icon, or execute the program from within Visual C++.

Points of Interest

This bulk of this code was generated by the Application Wizard, but there are two important areas to examine.

- Application initialization
- What happens when the timer expires?

Application initialization

The initialization of the application is performed when the window is created. The next two lines attempt to initialize the System II secondary application and present a message box if one cannot be created.

```
if(!S2init(0, INIT_SECONDARY, 5000))
    AfxMessageBox("Cannot initialize a secondary
    process");
```

What happens when the timer expires?

The OnTimer function gets called each time the 1 second timer expires. Before any System II calls can be made, an AP lock is requested. If successful, the routine checks for the presence of the buffer and copies that buffer on the stack. The length is determined and the buffer is dropped to prevent data being left on the stack. The display is set with the buffer length or "???" if no buffer is present. All of the error handling is performed with getS2err.

```
void CExample1Dlg::OnTimer(UINT nIDEvent)
{
    int nBufferLength;
    char szBufferLength[10];
    char szErrMsg[255];

    if(APlock(100, 0))
    {
        if(!getaddr(1))

        m_DisplayText.SetWindowText("???");
        else
        {
            qpush16(1);
            nBufferLength = topsize();
            drop();

            _itoa(nBufferLength,
                szBufferLength, 10);
            m_DisplayText.SetWindowText(
                szBufferLength);
        }
        if(getS2err(szErrMsg))
            AfxMessageBox(szErrMsg);
        APunlock(0);
    }

    CDialog::OnTimer(nIDEvent);
}
```

Example 2: A Primary Application

This application builds a dialog window. The Go! button builds two stimuli and presents them using generic DA calls. Example1 should still be running to see the primary/secondary interaction.

File of Interest

- Example2Dlg.cpp C++ file for the dialog box

Required Hardware

- AP2 Array Processor
- D/A Digital-to-analog converter

Running the Application

Either double click the Example2.exe icon, or execute the program from within Visual C++.

Points of Interest

This bulk of this code was generated by the Application Wizard, but there are three important areas to examine.

- Application Initialization
- What happens when the Go button is clicked?
- Terminating the application

Application initialization

The initialization of the application is performed when the window is created. It is here that the buffers are allocated. As always, error checking is performed after calls to hardware are made.

```
if(!S2init(0, INIT_PRIMARY, 1000))
    AfxMessageBox("Cannot initialize a
primary process");

if(APlock(100, 1))
{
    nBuf1 = _allot16(NPOINTS);
    nBuf2 = _allot16(NPOINTS);
    APunlock(0);
}
else
{
    getS2err(szEmess);
    AfxMessageBox(szEmess);
    exit(0);
}
```

What happens when the Go button is clicked?

The OnGo function is called and the stimuli are built. The tone stimulus is created on the stack while the noise stimulus is created by the OnGo routine. Using the rand function, the noise pip is built with a maximum value of 32000. The data is then transferred to the AP2 card by using the System II call, push16. Take note that the hardware locks are requested and calls to hardware are dependent on receiving those locks. As always, error checking is performed whenever calls to hardware are made.

```
//      Build stimulus for Buf2
      srand( (unsigned)time( NULL ) );
      for(i=0; i<NPOINTS; i++)
          sData[i] = 2*(rand()-
              16384)*(32000/32767);
      push16(sData, NPOINTS);
      qwind(2.0, SRATE);
      qpop16(nBuf2);
```

Terminating the application

In order to terminate the application, some System II housecleaning must be performed. Specifically, all allotted buffers must be de-allocated. To ensure that this is done properly, a while loop informs the user that an AP lock could not be acquired and therefore the application cannot be fully terminated. After the lock is received, the buffers are de-allotted and the program ends with the obligatory S2Close().

```
while(!APlock(100, 1))
{
    AfxMessageBox("Cannot close Example
        2.\n Release AP lock and
        click OK.")
}
deallot(nBuf1);
deallot(nBuf2);
S2close();
```

Example 3: PD1 Supplemental Application

This exercise implements a dialog window. The Go! button implements a PD1 circuit.

File of Interest

- Example3Dlg.cpp C++ file for the dialog box

Required Hardware

- AP2 Array Processor
- PD1 Power SDac

Running the Application

Either double click the Example3.exe icon, or execute the program from within Visual C++.

Points of Interest

This bulk of this code was generated by the Application Wizard, but there are two important areas to examine.

- Application Initialization
- What happens when the Go button is clicked?
- Terminating the application

Application initialization

The initialization of the application is performed when the window is created.

```
if(!S2init(0, INIT_PRIMARY, 1000))
    AfxMessageBox("Cannot initialize a
    primary process");
```

What happens when the Go button is clicked?

The OnGo function is called and the PD1 circuit is routed. The circuit includes a delays, filtering, and the summing of multiple inputs. The circuit remains active until the message box is clicked. In general, the most obvious results appear with a broadband noise source. The PD1export() command could have been used for the exported variables, but is not the most direct approach for this language.

Terminating the application

The program ends with the required S2Close().

Chapter 8 Using the Visual Basic Interface

The Mechanics of Working with the S232 Software Interface Suite for Visual Basic

The application must know the procedure declaration and the procedure location in the DLL before it can be accessed. This information is present in the module file S232.bas.

All the System II API calls are declared as global public functions and procedures. They may be called from any point in the application. If the module file is not included, the compiler will not recognize the procedure call.

Including the file is relatively straight forward. Development environments have lists of directories that are searched for different file types, e.g. modules(.bas). In these search paths include the C:\Tdt\S232\Vb directory.

The individual file, s232.bas, may also be included in the project directly.

S232 Commands in Visual Basic

As a language, Basic has a set of rules to guide programmers in declaring function names and variable names. In translating the System II API from the C language, some of the commands are lost in the Basic context due to the incompatibility of language constructs.

- Basic does not allow names to begin with an under bar (_*).
- Basic is not case sensitive.
- Basic reserves some keywords.

Any System II function or constant that begins with an under bar is now preceded with "UB". For example, _allot16 becomes UB_allot16.

Any System II command/constant that shares its name either with a reserved word or with another command/constant is preceded with "q". The following is a list of System II commands/constants whose names incorporate the above change.

Command/Constant	Name	Modified Name
System II Command:	scale	qscale
System II Constant	ANY	qANY
System II Constant	GAUSS	qGAUSS
System II Constant	ON	qON
System II Constant	SINE	qSINE

PD1 Support of HRTF Headers

HRTF file headers contain all the information about the HRTFs themselves and how they are stored. The System II commands that utilized this structure, LoadHRTF and PushHRTF, now use a call by reference to a variant data type. Simply declare enough room for 1024 16-bit words and pass that variable by reference into the function calls.

Working with Strings

Strings in Visual Basic are treated differently than strings in C. Visual Basic uses the Unicode string format. As a result, the programmer is required to make the appropriate conversions to ASCII before transmitting strings to the System II API.

Working with Exported Variables for the PD1

The Visual Basic interface does not support direct access of exported variables. As a result, only the indirect access method is allowed as follows by using the PD1export command.

```
Dim ADC As Long
ADC = PD1export(ByVal xADC, 0)
```

***Chapter 9* Visual Basic Examples**

The examples in this chapter familiarize the user with the structure, definition, and implementation of the S232 Software Interface Suite. Although basic in nature, they assume a familiarity with the TDT System II software interface.

The user is encouraged to try all examples. The relationship of a primary and secondary application is illustrated in using Examples 1 & 2 together. Example 3 shows how a simple PD1 application could be written.

Example 1: A Secondary Application

Example 2: A Primary Application

Example 3: A PD1 Supplemental Application

Example 1: A Secondary Application

This exercise implements a dialog window. The application has a timer that upon expiring reports the length of a specific buffer.

File of Interest

- frmMain.frm Visual Basic form file for the dialog box

Required Hardware

- AP2 Array Processor

Running the Application

Either double click the Example1.exe icon, or execute the program from within Visual Basic.

Points of Interest

This bulk of this code was generated by the Application Wizard, but there are two important areas to examine.

- Application initialization
- What happens when the timer expires?

Application initialization

The initialization of the application is performed when the window is created. The next two lines attempt to initialize the System II secondary application and present a message box if one cannot be created.

```
lngRet = S2init(ByVal 0, ByVal INIT_SECONDARY,
ByVal 1000)
    If lngRet = 0 Then
        Call MsgBox("Cannot initialize a
secondary process")
    End If
```

What happens when the timer expires?

The tmrBufferLength_Timer function gets called each time it expires. Before any System II calls can be made, an AP lock is requested. If successful, the routine checks for the presence of the buffer and copies that buffer on the stack. The length is determined and the buffer is dropped to prevent data being left on the stack. The display is set with the buffer length or "???" if no buffer is present. All of the error handling is performed with getS2err.

```
Private Sub tmrBufferLength_Timer()  
    Dim strBufferLength As String  
    Dim lngRetAP As Long  
    Dim lngRetBuf As Long  
  
    lngRetAP = APlock(ByVal 100, ByVal 0)  
    If lngRetAP = 1 Then  
        lngRetBuf = getaddr(ByVal 1)  
        If lngRetBuf <> 0 Then  
            Call qpush16(ByVal 1)  
            strBufferLength = topsize()  
            drop  
            Call ErrorCheck  
            lblDisplay.Caption =  
                strBufferLength  
        Else  
            lblDisplay.Caption = "???"  
        End If  
        Call APunlock(ByVal 0)  
    End If  
End Sub
```

Example 2: A Primary Application

This application builds a dialog window. The Go! button builds two stimuli and presents them using generic DA calls. Example1 should still be running to see the primary/secondary interaction.

File of Interest

- frmMain.frm Visual Basic form file for the dialog box

Required Hardware

- AP2 Array Processor
- D/A Digital-to-analog converter

Running the Application

Either double click the Example2.exe icon, or execute the program from within Visual Basic.

Points of Interest

This bulk of this code was generated by the Application Wizard, but there are three important areas to examine.

- Application Initialization
- What happens when the Go button is clicked?
- Terminating the application

Application initialization

The initialization of the application is performed when the window is created. It is here that the buffers are allocated. As always, error checking is performed after calls to hardware are made.

```
'Allocate memory
lngRet = APlock(ByVal 100, ByVal 0)
If lngRet = 1 Then
    lngBuf1 = UB_allot16(ByVal NPTS)
    lngBuf2 = UB_allot16(ByVal NPTS)
    Call APunlock(ByVal 0)
Else
    Call MsgBox("Cannot allocate buffers")
End If
```

What happens when the Go button is clicked?

The cmdGo_Click function is called and the stimuli are built. The tone stimulus is created on the stack while the noise stimulus is created by the cmdGo_Click routine. Using the Rnd function, the noise pip is built with a maximum value of 32000. The data is then transferred to the AP2 card by using the System II call,

push16. Take note that the hardware locks are requested and calls to hardware are dependent on receiving those locks. As always, error checking is performed whenever calls to hardware are made.

```
'Build noise stimulus
  Randomize
  For lngI = 1 To NPTS
    intNoise(lngI) = 32000 * (2 * (Rnd -
      0.5))
  Next
  Call push16(intNoise(0), ByVal NPTS)
  Call qwind(ByVal 2#, ByVal SRATE)
  Call qpop16(ByVal lngBuf2)
```

Terminating the application

In order to terminate the application, some System II housecleaning must be performed. Specifically, all allotted buffers must be de-allocated. To ensure that this is done properly, a while loop informs the user that an AP lock could not be acquired and therefore the application cannot be fully terminated. After the lock is received, the buffers are de-allotted and the program ends with the obligatory S2Close().

```
lngRet = APlock(100, 0)
  Do While lngRet = 0
    Call MsgBox("Cannot close Example2.
      Release AP lock.")
    lngRet = APlock(100, 0)
  Loop

  Call deallot(lngBuf1)
  Call deallot(lngBuf2)
  Call S2close
```

Example 3: PD1 Supplemental Application

This exercise implements a dialog window. The Go! button implements a PD1 circuit.

Files of Interest

- frmMain.frm Visual Basic form file for the dialog box

Required Hardware

- AP2 Array Processor
- PD1 Power SDac

Running the Application

Either double click the Example3.exe icon, or execute the program from within Visual Basic.

Points of Interest

This bulk of this code was generated by the Application Wizard, but there are two important areas to examine.

- Application Initialization
- What happens when the Go button is clicked?
- Terminating the application

Application initialization

The initialization of the application is performed when the window is created.

```
Ret = S2init(ByVal 0, ByVal INIT_PRIMARY,  
            ByVal 1000)  
If Ret = 0 Then  
    Call MsgBox("Cannont initialize a  
               primary process")  
End If
```

What happens when the Go button is clicked?

The cmdGo_Click function is called and the PD1circuit is routed. The circuit includes a delays, filtering, and the summing of multiple inputs. The circuit remains active until the message box is clicked. In general, the most obvious results appear with a broadband noise source. The PD1export() command must be used for the exported variables as shown.

Terminating the application

The program ends with the required S2Close().

Chapter 10 Using the Delphi Interface

The Mechanics of Working with the S232 Software Interface Suite for Delphi

The application must know the procedure declaration and the procedure location in the DLL before it can be accessed. The information the application requires is present in the unit file S232.pas.

All the System II API calls are declared as functions and procedures. They may be called from any point in the application. If the unit file is not included, the compiler will not recognize the procedure call.

Including the files is relatively straight forward. Development environments have lists of directories that are searched for different file types, e.g. units(.pas). Include the C:\Tdt\S232\Delphi directory in these search paths.

The individual file, s232.pas, may also be included in the project directly.

S232 Commands in Delphi

As a language, Delphi has a set of rules to guide programmers in declaring function names and variable names. In translating the System II API from the C language, some of the commands are lost in the Delphi context due to the incompatibility of language constructs.

- Delphi is not case sensitive.
- Delphi reserves some keywords.

Any System II command/constant that shares its name either with a reserved word or with another command/constant is preceded with “_”. The following is a list of System II commands or constants whose name incorporates the above change.

Command/Constant	Name	Modified Name
System II Command	record	_record
System II Constant	EXTERNAL	_EXTERNAL
System II Constant	GAUSS	_GAUSS
System II Constant	ON	_ON
System II Constant	SINE	_SINE

PD1 Support of HRTF Headers

HRTF file headers contain all the information about the HRTFs themselves and how they are stored. The System II commands that utilized this structure, LoadHRTF and PushHRTF, now use a call by reference to a variant data type. Simply declare enough room for 1024 16-bit words and pass that variable by reference into the function calls.

Working with Exported Variables for the PD1

The Visual Basic interface does not support direct access of exported variables. As a result, only the indirect access method is allowed as follows by using the PD1export command.

```
ADC: Integer;  
Indicies: Integer  
Indicies[1] := 1;  
ADC := PD1export(xADC, Indicies[1]);
```

***Chapter 11* Delphi Examples**

The examples in this chapter familiarize the user with the structure, definition, and implementation of the S232 Software Interface Suite. Although basic in nature, they assume a familiarity with the TDT System II software interface.

The user is encouraged to try all examples. The relationship of a primary and secondary application is illustrated in using Examples 1 & 2 together. Example 3 shows how a simple PD1 application could be written.

Example 1: A Secondary Application

Example 2: A Primary Application

Example 3: A PD1 Supplemental Application

Example 1: A Secondary Application

This exercise implements a dialog window. The application has a timer that upon expiring reports the length of a specific buffer.

Files of Interest

- Example1.pas Delphi file for the dialog box

Required Hardware

- AP2 Array Processor

Running the Application

Either double click the Ex1.exe icon, or execute the program from within Delphi.

Points of Interest

This bulk of this code was generated by the Application Wizard, but there are two important areas to examine.

- Application initialization
- What happens when the timer expires?

Application initialization

The initialization of the application is performed when the window is created. The next two lines attempt to initialize the System II secondary application and present a message box if one cannot be created.

```
Process:= INIT_SECONDARY;
Ret := S2init(0, Process, 5000);
if Ret = 0 then
    ShowMessage('Cannot initilize a
                secondary process');
```

What happens when the timer expires?

The TForm1.tmr200msTimer function gets called each time it expires. Before any System II calls can be made, an AP lock is requested. If successful, the routine checks for the presence of the buffer and copies that buffer on the stack. The length is determined and the buffer is dropped to prevent data being left on the stack. The display is set with the buffer length or “???” if no buffer is present. All of the error handling is performed with getS2err.

```
procedure TForm1.tmr200msTimer(Sender:
TObject);
var
  Ret: Integer;
  BufferLength: Integer;
  StrBufferLength: string;

begin
  Ret := APlock(100, 0);
  if Ret = 1 then
    Ret := getaddr(1);
    if Ret <> 0 then
      begin
        qpush16(1);
        BufferLength := topsize();
        StrBufferLength :=
          IntToStr(BufferLength);
        edtDisplay.Text :=
          StrBufferLength;
        drop();
      end
    else
      edtDisplay.Text := '???';
  APunlock(0);
end;
```

Example 2: A Primary Application

This application builds a dialog window. The Go! button builds two stimuli and presents them using generic DA calls. Example1 should still be running to see the primary/secondary interaction.

Files of Interest

- Example2.pas Delphi file for the dialog box

Required Hardware

- AP2 Array Processor
- D/A Digital-to-analog converter

Running the Application

Either double click the Ex2.exe icon, or execute the program from within Delphi.

Points of Interest

This bulk of this code was generated by the Application Wizard, but there are three important areas to examine.

- Application Initialization
- What happens when the Go button is clicked?
- Terminating the application

Application initialization

The initialization of the application is performed when the window is created. It is here that the buffers are allocated. As always, error checking is performed after calls to hardware are made.

```
Process := INIT_PRIMARY;
Ret := S2init(0, Process, 5000);
if Ret = 0 then
    ShowMessage('Cannot initialize a primary
                process.');
```

```
Ret := APlock(100, 0);
if Ret = 1 then
    begin
        Buf1 := _allot16(NPTS);
        Buf2 := _allot16(NPTS);
        APunlock(0);
    end
else
    ShowMessage('Cannot allocate buffers');
```

What happens when the Go button is clicked?

The TForm1.btnGoClick function is called and the stimuli are built. The tone stimulus is created on the stack while the noise stimulus is created by the TForm1.btnGoClick routine. Using the Random function, the noise pip is built with a maximum value of 32000. The data is then transferred to the AP2 card by using the System II call, push16. Take note that the hardware locks are requested and calls to hardware are dependent on receiving those locks. As always, error checking is performed whenever calls to hardware are made.

```
// Build a stimulus for Buf2
  SetLength(Ch2, NPTS);
  Randomize;
  for i := 1 to NPTS do
    Ch2[i] := Trunc(32000*(2*Random-1));
  push16(Ch2[1], NPTS);
  qwind(2.0, SRATE);
  qpop16(Buf2);
```

Terminating the application

In order to terminate the application, some System II housecleaning must be performed. Specifically, all allotted buffers must be de-allocated. To ensure that this is done properly, a while loop informs the user that an AP lock could not be acquired and therefore the application cannot be fully terminated. After the lock is received, the buffers are de-allotted and the program ends with the obligatory S2Close().

```
Ret := APlock(100, 0);
while Ret = 0 do
  begin
    ShowMessage('Cannot Close Example2.
      Release AP lock.');
```

Ret := APlock(100, 0);

```
  end;
deallot(Buf1);
deallot(Buf2);
S2Close();
```

Example 3: PD1 Supplemental Application

This exercise implements a dialog window. The Go! button implements a PD1 circuit.

Files of Interest

- Example3.pas Delphi file for the dialog box

Required Hardware

- AP2 Array Processor
- PD1 Power SDac

Running the Application

Either double click the Ex3.exe icon, or execute the program from within Delphi.

Points of Interest

This bulk of this code was generated by the Application Wizard, but there are two important areas to examine.

- Application Initialization
- What happens when the Go button is clicked?
- Terminating the application

Application initialization

The initialization of the application is performed when the window is created.

```
Process := INIT_PRIMARY;  
Ret := S2init(0, Process, 1000);  
if Ret = 0 then  
    ShowMessage('Cannot initialize a primary process');
```

What happens when the Go button is clicked?

The TForm1.btnGoClick function is called and the PD1 circuit is routed. The circuit includes a delays, filtering, and the summing of multiple inputs. The circuit remains active until the message box is clicked. In general, the most obvious results appear with a broadband noise source. The PD1export() command must be used for the exported variables.

Terminating the application

The program ends with the required S2Close().