

# APOS

---

The AP2 DSP Operating System

## Software Reference

Software version 2.12  
Printed 10/15/97



# Table of Contents

Introduction .....	1
APOS Basics .....	3
Multi-Card Operation.....	3
APOS to PC Interface .....	4
AP Address Conflict.....	5
Numeric Representation.....	5
Buffer Management.....	7
Using Operation Blocks .....	8
Handling Buffer Sizes .....	8
Complex Buffers .....	9
Floating-Point Overflow .....	10
Using the AP2's Optical Interface .....	10
Software Setup .....	11
Using APLD.....	11
An Example in C.....	12
Procedure Descriptions .....	14
Description Format.....	16
Description Conventions .....	17
Status and Control.....	19
apinit.....	19
ap_select .....	19
ap_present.....	19
ap_active.....	20
freewords.....	20
topsize.....	20
stackdepth.....	21
Buffer Management - Stack Operations .....	23
drop.....	23
dropall.....	23
swap.....	23
qdup.....	24
dupn.....	24
cat .....	24
catn .....	24
block.....	25
noblock.....	25

totop.....	25
extract.....	26
reduce.....	26
<b>Buffer Management - Push/Pop.....</b>	<b>29</b>
push16.....	29
pushport16.....	29
pushdisk16.....	29
pushf.....	30
pushportf.....	30
pushdiskf.....	31
pushdiska.....	31
qpushf.....	31
qpushpartf.....	32
qpush16.....	32
qpushpart16.....	32
dpush.....	33
pop16.....	33
popport16.....	33
popdisk16.....	34
popf.....	34
popportf.....	34
popdiskf.....	35
popdiska.....	35
poppush.....	35
qpopf.....	36
qpoppartf.....	36
qpop16.....	36
qpoppart16.....	37
parse.....	37
<b>Buffer Management - DAMA Operations.....</b>	<b>39</b>
_allotf.....	39
_allot16.....	39
allotf.....	39
allot16.....	40
trash.....	40
deallot.....	40
getaddr.....	41
setaddr.....	41
disk2dama16.....	41

dama2disk16.....	42
<b>Data Generators .....</b>	<b>45</b>
value .....	45
fill .....	45
qrand.....	45
flat.....	46
gauss (qgauss) .....	46
seed.....	46
tone.....	47
make .....	47
makedama16.....	47
makedamaf .....	48
whatis .....	48
<b>Basic Math .....</b>	<b>49</b>
add .....	49
radd.....	49
subtract .....	49
mult .....	50
absval.....	50
divide.....	50
shift.....	51
scale.....	51
inv.....	51
sqrt.....	51
power.....	52
square .....	52
logten.....	52
loge.....	52
logn.....	53
alogten .....	53
aloge .....	53
maxlim.....	54
minlim .....	54
maglim.....	54
<b>Other Functions .....</b>	<b>57</b>
cumsum .....	57
decimate .....	57
interpol .....	58
foldnadd.....	58

getnarts .....	59
Trigonometry .....	61
cosine.....	61
sine (qsine) .....	61
tangent.....	61
acosine.....	62
asine.....	62
atangent .....	62
atantwo .....	63
qwind.....	63
Complex Operators .....	65
polar.....	65
rect.....	65
ximag.....	65
xreal.....	66
shuf.....	66
split.....	66
cadd .....	67
cmult.....	67
cinv.....	67
Fast Fourier Transformation .....	69
cfft .....	69
cift.....	69
rfft.....	69
rift .....	70
hann .....	70
hamm.....	70
Digital Filtering .....	72
iir .....	72
_iir .....	72
fir.....	74
_fir.....	75
reverse .....	76
Summation Functions .....	77
sum .....	77
average .....	77
maxval.....	77
minval.....	78
maxmag.....	78

maxval_.....	78
minval_.....	78
maxmag_.....	79
<b>Play-Record Operations .....</b>	<b>81</b>
<b>Overview .....</b>	<b>81</b>
Using APOS Play and Record.....	82
Special Considerations and Comparisons .....	84
Cycle Usage Calculation .....	85
Simple Play and Record .....	86
play .....	86
record (qrecord).....	87
fastrecord.....	88
Dual Channel Play and Record .....	89
dplay .....	89
drecord.....	89
Sequenced Play and Record .....	90
seqplay.....	93
seqrecord .....	95
mrecord.....	97
mplay .....	99
Pause Feature in Sequence Play .....	100
pfireone.....	100
pfireall .....	101
ppausestat .....	101
Auxiliary Functions.....	102
playseg.....	102
recseg.....	102
playcount .....	103
reccount .....	103
chgplay .....	104
<b>Miscellaneous Functions.....</b>	<b>105</b>
plotmap.....	105
plotwith .....	107
<b>Macro Functions .....</b>	<b>109</b>
recordmac .....	111
endmac .....	111
runmac.....	112
whatmac .....	112
whatstep.....	112

stopmac.....	113
ls .....	113
stepval24.....	113
stepvalf .....	114
loopn.....	114
jump.....	114
usestepval .....	115
usedamalist.....	115
counter24.....	116
counterf.....	116
if24 .....	117
iff.....	117
constant.....	118
APOS Function Index .....	119

## Introduction

The TDT Array Processor (AP2/1) is a high performance numeric array processing system based on the AT&T DSP32C. The AP2 is equipped with a 50MHz DSP32C, a 128K Byte program memory bank, up to 512K of zero wait state static memory, and up to eight megabytes of DRAM. The AP2 also employs a PC interfacing system which allows for speedy data and control transfers between the AP2 and its host PC. The AP2 also includes a specialized optical fiber interface for connecting the AP2 to TDT's System II line of signal processing modules. These modules perform such functions as A/D and D/A conversion, as well as, a variety of analog signal processing functions from simple programmable attenuation to complex waveform discrimination.

Those who have programmed the DSP32C or similar array processors can attest to the difficulty of the task. When programming in assembly, effects from latency and pipe lining make generating working code a nearly impossible job for all but expert programmers. They will also attest that programming with a 'C' compiler results in inefficient code that runs far too slow. The APOS software system offers a solution to this problem by providing a full software interface between the AP2 and your favorite high-level language. Residing within the 128K program memory of the AP is a complete library of fast numeric processing routines tied together with a powerful operating system. This operating system is complete with a PC-host interface, a robust memory management system, and a full error handling system.

The APOS routine library consists of a group of efficient number crunching routines which perform many operations from complex FFTs to simple addition. In an effort to minimize programming overhead and to simplify program code, the APOS software employs a Memory Management System, which utilizes both a stack and a dynamically allocated memory area. This specialized system allows for the coding of very complex operations without unnecessary data transfers to and from the PC.

Included in APOS are powerful functions for sending and receiving data on the AP2's optical interface. When used in conjunction with an appropriate XBUS analog I/O module these functions allow the user to generate and acquisition analog signals. These play and record procedures include powerful sequencing functions allowing complex play and record scenarios to be programmed easily. There is also a group of macro functions which will allow the user to better utilize the power of the PC and the AP1/AP2 array processor.

APOS will program the AP2 and AP1 cards referred to in this document as Array Processors (APs).

## APOS Basics

The APOS driver package provides a wide variety of operations that are executed from a high-level language using simple procedure and function calls. The high-level language program containing these APOS procedure and function calls is called the 'host' or 'application' program. The operations are performed on arrays located in AP memory on a logical memory structure called the STACK. These arrays are transferred to and from PC memory via 'push' and 'pop' type instructions. STACK arrays can also be moved to and from a second AP internal memory area called the Dynamically Allocated Memory Area or DAMA. The DAMA allows for the temporary storage of AP STACK buffers without occupying PC memory or requiring redundant data transfers between the PC and the AP. Also, all play and record buffers used by the AP2's optical interface are held in the DAMA memory area. Special push and pop instructions also exist for transfers to DOS devices (i.e., disk) and I/O ports.

### Multi-Card Operation

APOS will control two APs within the same computer. The constants APa and APb are used to indicate devices at the lower and upper I/O locations respectively. The *apinit* procedure is used to initialize each AP. For example, calling *apinit(APa)* will reset the AP located at the lower address location. The *apinit* call returns true if the specified AP was detected and reset, otherwise it returns false. The *apinit* call must be executed by the application program before any APOS functions can be used. APOS employs a software switching paradigm to facilitate dual-card operation, thereby simplifying programming and making APOS's dual-card abilities transparent to the single card user.

An AP2 device is not specified in most APOS procedures--instead operations are performed on the current *Active* AP. An AP is made *Active* by a call to the *ap\_select* procedure. Once an AP card is selected (made *Active*), all subsequent APOS operation calls will be directed to this card. For those using a single card, this multi-card capability is completely

transparent with the *apinit* procedure returning with the specified AP selected.

## APOS to PC Interface

APOS procedures and functions are called with the standard 'C' calling sequence. These calls are translated into command codes by the APOS software and the codes are sent to the AP. Each command received by the AP invokes a unique process called an 'Operation.' The time it takes an operation to run is called the 'Operation Time' and is usually specified in microseconds per datum. After a command is sent to the AP, the AP is then given a number of seconds to acknowledge the command. When the acknowledgment is received by the PC, any necessary parameters are transferred and the PC returns to execute the next line of the application program. This command-response paradigm yields a few interesting results that one should be aware of:

- Although a few seconds are allowed for AP acknowledgment, in most cases the response time will be very short. This excessive time-out period allows the APOS procedures to be called sequentially in the application program without concern of operation times. Because nearly all APOS operations can be performed on many datum in less than a few seconds, one should never receive a time-out error as a result of a previous AP operation not finishing within the time-out period. Thus, if an APOS procedure is called while the AP is still working on a previously programmed operation, the APOS software will send the new command code and wait for a response. When the acknowledgment is received back from the AP it means the previous operation has been completed and the new one (just sent) is underway.
- Because the PC sends the command, waits for acknowledgment and then continues, the host computer is free to perform its own computations while the AP is performing the programmed operation. A Boolean function, *ap\_active*, can be used to poll the AP and detect for operation completion.

- A pitfall of this command-response paradigm, is that error messages resulting from an APOS command are not detected until the next APOS command is sent. Suppose a request is made to duplicate (qdup) the buffer at the top of the stack. The command is sent by the APOS software to the AP and the AP acknowledges the command. After checking available memory, the APOS onboard operating system realizes there is insufficient memory to perform the duplication, and an error sequence is initiated. This error sequence is not detected by the host software until the next APOS command is attempted. However, this error latency is typically not a problem because the offending procedure is always identified in the error message sent by the AP error handling system. But one should be aware it exists!

NOTE: When optical interface data is being processed at high speed, one will notice an increase in APOS Operation Times.

## AP Address Conflict

AP card does not use any PC interrupts or DMA channel, however, it uses an I/O address to communicate with the host PC. An AP card can use either APa (Hex 220-238), or APb (Hex 340-258). When a computer has a CD-ROM, sound card, network card or any other plug-in cards, the AP card may use an address which is occupied by another device. As a result, AP card or the other device may not function properly. If this happens, you can resolve it by changing AP address between APa and APb (see *System II Installation Guide*). If this does not solve the problem, you have to change the address of the other device to free either APa or APb for the AP card.

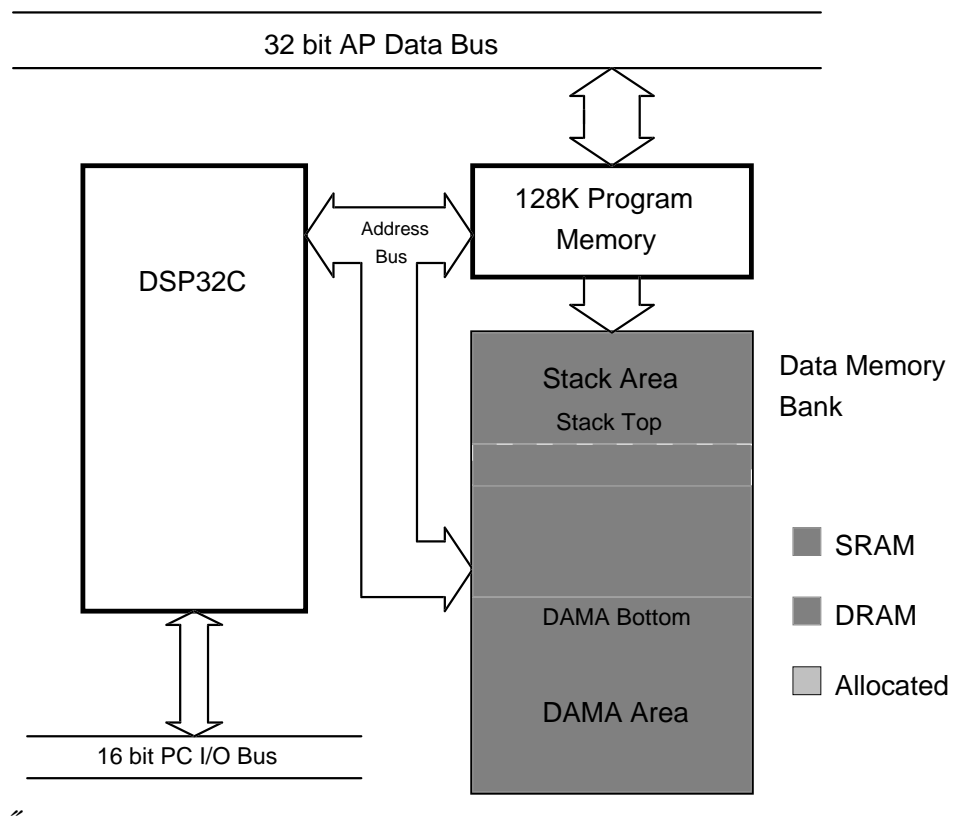
## Numeric Representation

On the AP, arrays are kept linearly in RAM and are made up of elements called datum. Often these data arrays are called buffers or vectors. All three terms will be used interchangeably in this document. 'Stacked' datum are stored in AP RAM in a special floating-point format; however, both single precision floats and 16-bit integers can be pushed to and popped from the AP stack, and the proper format conversion will be

performed automatically. The DAMA memory can hold two data types: the standard DSP32C floating-point type and 16-bit integer datum. Typically 16-bit integer buffers are used to handle operations on the AP2 optical interface. Conversions from integer to floating-point format can be performed by the DSP32C in a single cycle, which means that data is converted as it is moved without slowing the process. The DSP32C's floating-point format consists of a 24-bit mantissa and an 8-bit exponent resulting in a floating-point range from  $5.9 \times 10^{-39}$  to  $3.4 \times 10^{38}$ . The DSP32C's ability to perform floating-point versus fixed-point mathematical operations gives it an advantage of numeric representation over a 1500 decibel range. The maximum value which can be represented ( $3.4 \times 10^{38}$ ) will be referred to in this document as MAXFLOAT. The minimum value ( $5.9 \times 10^{-39}$ ) is called MINFLOAT.

## Buffer Management

The AP Memory Management System keeps data buffers in two logical data areas. The first memory area is called the STACK. Buffers are placed on and removed from the stack using 'push' and 'pop' type commands. Stack memory is allocated from low addresses upward. A second memory area is used to store temporary and play/record buffers. This Dynamically Allocated Memory Area (DAMA) is allocated from high addresses downward. If the top of the STACK meets the bottom of the DAMA, an out-of-memory error results.



**AP2 Memory Diagram**

Referring to the Memory Diagram shown above, note that the AP2 combines SRAM and DRAM to form a single contiguous Data Memory Bank. Because the SRAM forms the lower portion of the memory bank (the Stack Area), and the DRAM makes up the upper portion of memory

(the DAMA Area), the AP2 will always use the fast and slow memory most efficiently. This is because the faster SRAM will hold the stack buffers which are used in number crunching. Conversely, the DRAM will hold DAMA buffers, which are only used to hold temporary data and play/record data.

## Using Operation BLOCKS

By default, an APOS operation performs its particular function on every point in the buffer. Often it is desirable to have an operation work on only a portion of a buffer. The *block* procedure call is used to define an operation-block for the buffer located on top of the stack. Subsequent calls to block-sensitive procedures will result in that operation being performed on those points within the specified block. A block is easily changed with another call to *block* and can be removed with a call to *noblock*.

NOTE: A block placed on a stack buffer will remain with that buffer even if it is pushed down into the stack. Don't be surprised if a buffer emerges from the stack with a block previously applied. Some operations are not blocking-sensitive; and they will always work on the entire data buffer. Also, buffers popped to DAMA lose all blocking information.

## Handling Buffer Sizes

Buffers of nearly any size can be allocated and worked on. The lower limit for a buffer is two, while the upper limit is dictated by the amount of free memory available on the AP. Note that certain operations require that a buffer have a radix-2 number of points. This means a buffer must have a number of points equal to a power of two (i.e., 64, 128, 256 ). In most cases there is a lower limit on the number of radix-2 points around 32.

When operations are requested that involve buffers of different lengths, the buffer with the fewest number of points governs how many points the operation is performed upon. The resulting buffer's length is controlled by the length of the destination buffer. The following example illustrates this point:

APOS Code	AP2 STACK
dpush(20);	buf2[20].
value(2.0);	buf2[20]. (all equal to 2.0)
dpush(10);	buf1[10]   buf2[20].
value(1.0);	buf1[10]   buf2[20]. (buf1 = 1.0)
add();	buf2*[20].

After the above set of instructions is executed, the stack would contain one array of 20 elements. The first 10 elements of the array would contain the sum of  $1.0 + 2.0 = 3.0$ , while the last 10 data would contain the original value in buf2, or 2.0.

## Complex Buffers

As stated previously, a standard AP [STACK] element consists of an array of data at least two points long. Some APOS procedures operate on complex data and therefore require that a Complex Buffer Pair (CBP) be on the stack when the procedure is called. A CBP consists of two (non-shuffled) buffers of the same length located at the top of the stack. The buffers points, when matched, should form complex vectors in a rectangular format. The buffers should be pushed on the stack such that the buffer containing the imaginary data is on the top and the real buffer is just below (i.e., push the real buffer on first).

Other procedures require a radix-2 type buffer (RTB) or a complex radix-two buffer (CRTB). A radix-two buffer, as the name implies, must have a radix-2 number of data (i.e., 32, 64, 128, 256, etc.).

## **Floating-Point Overflow**

The APOS software system does not generate an error when a floating-point overflow occurs. Instead, the software attempts to return the most reasonable value depending on the type of overflow. For example, if a number is divided by zero, the value MAXFLOAT will be returned with the same sign as the numerator of the offending operation. Other times, the output cannot be assigned a logical value; thus, the results will be unpredictable. For example, calculating logarithms of negative numbers returns garbage results.

## **Using the AP2's Optical Interface**

As stated previously, the AP2 contains an optical interfacing system that allows it to work with TDT's line of System II modular devices. Data flows in and out of the optical interface in serial format at a very high rate of speed (12.5 million bits per second). APOS contains a variety of play (data output) and record (data input) procedures which greatly simplify the interface's use and enhance its power. Please refer to the "Procedure Descriptions" section for more information on using the APOS play and record procedures.

## Software Setup

Because the APOS software works with two microprocessors, the AP's DSP processor and the Intel processor running in your PC, it utilizes two different software patches. The software that runs the DSP32C resides on the AP card. The AP resident software is loaded to the AP card during system initialization and then becomes transparent to the user. The PC side of the APOS software consists of a library of high-level language functions that are linked to the application program. This library of functions provides a simple interface between the PC programmer and the AP hardware and onboard software.

APOS/XBDRV software installation is outlined in the System II Installation Guide. You may also perform the following steps to install APOS separately:

1. Insert APOS/XBDRV Disk 1 of 3 into your floppy drive, make this drive the current drive and type:
  - install
2. Choose the drive and directory to install APOS. The recommended directory is c:\tdt\drivers\apos.
3. Choose Install APOS.
4. Add the following lines to your autoexec.bat file:
  - c:
  - cd\tdt\drivers\apos\onboard
  - apld a (or apld b)
  - cd\

## Using APLD

The APLD Program is used to load the AP with its onboard operating system. This program reads the file AP.OBJ and loads the specified segments to the AP.

By default, APLD will load an AP2 located at the lower address location Hex 220-238. An argument can be used to load the upper AP2 I/O option.

- APLD APa                      Loads AP2 at Hex 220-238
- APLD APb                      Loads AP2 at Hex 240-258

## An Example in 'C'

As a first step, let's write a simple 'C' program that initializes the AP card using the *apinit()* procedure. Although you can use nearly any 'C' compiler, as well as other compilers which support the 'C' calling sequence, the following example will pertain to the Turbo 'C' compiler.

APOS.C contains all the AP2 drivers, and IO.OBJ is an optional I/O module for APOS.C. IO.OBJ is required for using Borland 'C' or Turbo 'C' compiler. If another 'C' compiler is used (such as Microsoft 'C'), this IO.OBJ is not required. Instead, following lines should be inserted to the top of APOS.H file:

```
#define MSC
#define INTMOVE
```

If you are using a 32-bit 'C' compiler, you should also add the following line to the top of APOS.H file:

```
#define COMP32
```

APOS software is provided in 'C' and Pascal source code. The user can use the APOS functions either in their source code, or in a compiled format. The following example uses the APOS 'C' source code directly in the programming:

1. Invoke the Turbo 'C' compiler and set it up to use proper memory model and other compiler/linker options.

2. Create a project file, make it the current project and add the following files to the project:

```
aptst1.c  
apos.c  
io.obj
```

3. Clear the editor and type in the following program:

```
#include <stdlib.h>  
#include <stdio.h>  
#include "c:\tdt\drivers\apos\c\apos.h"  
  
void main(void)  
{  
    clrscr();  
    if(apinit(APa))  
        printf('\n\nAPa Is Working!!!\n\n');  
    if(apinit(APb))  
        printf('\n\nAPb Is Working!!!\n\n');  
}
```

and save this program as APTST1.C. If this simple program compiles, links, and runs, then the software was installed properly. If you experience trouble, check all of your installation work and above procedures. If you find no problems, then call TDT.

# Procedure Descriptions

APOS procedures are divided into groups based on the type of function they perform. The following groups are included in this release of APOS:

## **Status and Control**

This group includes procedures for initializing the APOS system and resident hardware, as well as functions for returning important information about AP activities.

## **Buffer Management**

These procedures are used to move data to and from the AP Stack and DAMA. Procedures are included for moving data between the AP(s) and DOS devices (i.e., disk), as well as the PC I/O locations. Also included in this group are procedures for blocking data buffers.

## **Value Generators**

Use these procedures to fill a buffer with desired values. Included in this section are procedures for generating random numbers, tones, ramps, and other common signal functions.

## **Basic Math**

This group includes basic math functions, such as addition and subtraction, and single operand operations, such as logs and anti-logs.

## **Trigonometry**

The trigonometry group includes the sine, cosine, tangent functions along with their respective inverses. A four quadrant tangent function is also included.

**Complex Operators**

This section includes procedures for converting between Polar and Rectangular formats, and for performing operations such as complex multiplication and addition.

**Fast Fourier Transforms**

This group includes procedures for doing forward and inverse FFTs on complex and real data. Also included are hanning and hamming windowing procedures.

**Digital Filtering**

FIR and IIR filter routines including routines allowing initial conditions are included in this section.

**Summation**

This section includes procedures for computing buffer totals, averages, and maximum values.

**Play and Record**

This section contains the play and record procedures for driving the AP2 optical interface.

## Description Format

Each APOS procedure will be listed and described according to a specific format. This format is described below for the **add** operation:

- (1) **add()**
  - (2) **Prototype**      void                    add( void );
  - (3)                    PROCEDURE      add;
  - (4) **Operation**      st[2] = st[1] + st[2], [STACK] >> (void)
  - (5) **Speed**            0.4 µseconds per datum pair.
  - (6) **Limitations**    None.
  - (7) **Accuracy**        Full.
  - (8) **Example**
- ```

[ 1.0, 1.0, 1.0 ]
[ 2.0, 2.0, 6.0 ]
add();
[ 3.0, 3.0, 7.0 ]

```

The significance of each line is described below.

- (1) APOS operation format and name. In the example above, the operation is called **add** and has no parameters.
- (2) 'C' prototype.
- (3) Pascal prototype.
- (4) Functionality of the operation (refer to the conventions described in the following section). For the **add** operation, the top two stack buffers are added together and the stack is popped leaving the resulting 'sum' buffer at the top of the stack.
- (5) Speed bench-mark. In most cases, the bench-mark is given in time-per-datum format. For example, the **add** operation can add one million numbers in about 0.4 seconds. The times listed in this

format are derived by performing the operation 100 times on a buffer (or buffers) 10,000 points long.

- (6) Limitations associated with an operation. For example, the logarithm operations have the following statement on line (6): 'all buffer values must be greater than zero.'
- (7) Typical expected accuracy. For the **add** operation, no accuracy is lost during the operation, so the word "Full" appears on line (7). Full accuracy means the results are only limited by the floating-point representation used by the DSP32C, which is about 7-1/2 significant digits.
- (8) Example of operation usage. Selected APOS procedures will have examples provided. In this example, note the method used to indicate the STACK contents before and after the *add* operation:

[ 1.0, 1.0, 1.0 ] <= top of stack

[ 2.0, 2.0, 6.0 ]

(no other stacked buffers)

## Description Conventions

In order to simplify procedure explanations (line (4)), a few nomenclature standards will need to be adopted. There are three memory areas used by the APOS software -- the PC memory area, the AP Stack, and the AP Dynamically Allocated Memory Area (or DAMA). These three memory areas are each logically different but perform basically the same function. The following conventions of nomenclature will be used when discussing these memory areas and their elements:

- [STACK] Refers to the AP stack structure.
- st[N] Designates the Nth stack element.
- {st[N]} Designates the blocked portion of the Nth stack element.
- [DAMA] Refers to the DAMA structure.
- dama[n] Specifies a particular DAMA buffer 'n'.
- PC(buf) Buffer called 'buf' located in PC memory.

- --> Data transfer in the direction indicated.
- >> [STACK] Push onto the stack.
- [STACK] >> Pop off the stack.

## Status and Control

### apinit( dev )

|                    |                                                                                                                                                   |                                        |
|--------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------|
| <b>Prototype</b>   | int                                                                                                                                               | apinit( int <i>dev</i> );              |
|                    | FUNCTION                                                                                                                                          | apinit( <i>dev</i> :INTEGER): BOOLEAN; |
| <b>Operation</b>   | Initializes the specified AP device. Use APa, APb. Returns true if the specified device was detected and initialized, otherwise it returns false. |                                        |
| <b>Speed</b>       | NA.                                                                                                                                               |                                        |
| <b>Limitations</b> | NA.                                                                                                                                               |                                        |
| <b>Accuracy</b>    | NA.                                                                                                                                               |                                        |

### ap\_select( dev )

|                    |                                                                                                                                                                                                       |                                  |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------|
| <b>Prototype</b>   | void                                                                                                                                                                                                  | ap_select( int <i>dev</i> );     |
|                    | PROCEDURE                                                                                                                                                                                             | ap_select( <i>dev</i> :INTEGER); |
| <b>Operation</b>   | Makes the AP specified in <i>dev</i> 'Active'. Refer to this document's "Multi-Card Operation" section. Use the APOS defined constants/macros APa, APb, QAPa, and QAPb when specifying the AP device. |                                  |
| <b>Speed</b>       | 5.6 $\mu$ seconds.                                                                                                                                                                                    |                                  |
| <b>Limitations</b> | NA.                                                                                                                                                                                                   |                                  |
| <b>Accuracy</b>    | NA.                                                                                                                                                                                                   |                                  |

### ap\_present( dev )

|                    |                                                                                                            |                                            |
|--------------------|------------------------------------------------------------------------------------------------------------|--------------------------------------------|
| <b>Prototype</b>   | int                                                                                                        | ap_present( int <i>dev</i> );              |
|                    | FUNCTION                                                                                                   | ap_present( <i>dev</i> :INTEGER) :BOOLEAN; |
| <b>Operation</b>   | Returns TRUE if the AP specified in <i>dev</i> was detected by <b>apinit</b> , otherwise it returns FALSE. |                                            |
| <b>Speed</b>       | 3.2 $\mu$ seconds.                                                                                         |                                            |
| <b>Limitations</b> | NA.                                                                                                        |                                            |
| <b>Accuracy</b>    | NA.                                                                                                        |                                            |

## ap\_active( dev )

**Prototype**    int                    ap\_active( int *dev* );  
 FUNCTION       ap\_active( *dev* INTEGER) :BOOLEAN;

**Operation**    Returns TRUE while the AP specified in *dev* is not ready to receive the next operation command, otherwise returns FALSE. Use the APOS-defined constants/macros APa, APb, QAPa, and QAPb when specifying the AP device.

**Speed**         8.1  $\mu$ seconds.

**Limitations**   NA.

**Accuracy**     NA.

## freewords( )

**Prototype**    long                    freewords( void );  
 FUNCTION       freewords LONGINT;

**Operation**    Returns the number of unused 32-bit words available in AP memory.

**Speed**         59  $\mu$ seconds.

**Limitations**   NA.

**Accuracy**     NA.

## topsize( )

**Prototype**    long                    topsize( void );  
 FUNCTION       topsize LONGINT;

**Operation**    Returns the size (number of points, in 32-bit words) of the stack-top buffer. Zero is returned if the stack is empty.

**Speed**         59  $\mu$ seconds.

**Limitations**   NA.

**Accuracy**     NA.

## stackdepth( )

**Prototype** long stackdepth( void );

FUNCTION stackdepth :LONGINT;

**Operation** Returns the number of buffers currently pushed on the stack. Zero is returned if the stack is empty.

**Speed** 59  $\mu$ seconds.

**Limitations** NA.

**Accuracy** NA.

(this page intentionally left blank)

## Buffer Management - Stack Operations

### drop( )

**Prototype** void drop( void );  
PROCEDURE drop;

**Operation** [STACK] >> (void)

**Speed** 50 µseconds.

**Limitations** Calling drop on an empty stack will result in an error.

**Accuracy** NA.

### dropall( )

**Prototype** void dropall( void );  
PROCEDURE dropall;

**Operation** Clears AP stack.

**Speed** 50 µseconds.

**Limitations** Will **not** result in an error if called on an empty stack.

**Accuracy** NA.

### swap( )

**Prototype** void swap( void );  
PROCEDURE swap;

**Operation** st[1] <--> st[2]

**Speed** 1.7 µseconds per datum.

**Limitations** A temporary buffer is generated during swap; memory must be available.

**Accuracy** NA.

## qdup( )

**Prototype** void qdup( void );  
 PROCEDURE qdup;

**Operation** (void) >> [STACK], st[1] = st[2]

**Speed** 0.6 µseconds per datum.

**Limitations** Memory for new buffer must be available.

**Accuracy** NA.

## dupn( n )

**Prototype** void dupn( long n );  
 PROCEDURE dupn( n :LONGINT);

**Operation** Performs *qdup* n times.

**Speed** 0.6 µseconds per datum.

**Limitations** Memory for new buffers must be available.

**Accuracy** NA.

## cat( )

**Prototype** void cat( void );  
 PROCEDURE cat;

**Operation** Concatenates top two stack buffers into a single buffer.  
 This operation is NOT blocking-sensitive.

**Speed** 0.6 µseconds per datum.

**Limitations** Memory for new buffers must be available.

**Accuracy** NA.

## catn( n )

**Prototype** void catn( long n );  
 PROCEDURE catn( n :LONGINT);

**Operation** Concatenates top *n* stack buffers.

**Speed** 0.6 µseconds per datum.

**Limitations** Memory for new buffers must be available.

**Accuracy** NA.

## block( start , stop )

**Prototype** void block( long *start*, long *stop* );  
 PROCEDURE block( *start*, *stop* LONGINT );

**Operation** st[1] --> {st[1]} (based on *start* and *stop*).

**Speed** 75 µseconds.

**Limitations**  $0 \leq start \leq stop$ ,  $start \leq stop \leq$  (Number of points in buffer - 1).

**Accuracy** NA.

## noblock( )

**Prototype** void noblock( void );  
 PROCEDURE noblock;

**Operation** {st[1]} --> st[1] (removes blocking on st[1]).

**Speed** 40 µseconds.

**Limitations** None.

**Accuracy** NA

## totop( stn )

**Prototype** void totop( int *stn* );  
 PROCEDURE totop( *stn*: INTEGER );

**Operation** Copies specified stack buffer to top of stack  
 (void) >> [STACK], st[1] = st[*stn*]

**Speed** 0.6 µseconds per datum.

**Limitations** Memory for new buffer must be available.

**Accuracy** NA.

## extract( )

**Prototype** void extract( void );  
 PROCEDURE extract;

**Operation** Duplicates blocked portion of top of stack  
 (void) >> [STACK], st[1] = {st[2]}

**Speed** 0.6 µseconds per datum.

**Limitations** Memory for new buffer must be available.

**Accuracy** NA.

## reduce( )

**Prototype** void reduce( void );  
 PROCEDURE reduce;

**Operation** Reduce top of stack buffer to blocked portion of current top  
 of stack. **reduce** is faster than the "block-extract-swap-  
 drop" sequence done with earlier APOS versions.  
 st[1] = {st[1]}

**Speed** 0.6 µseconds per datum.

**Limitations** NA

**Accuracy** NA.

(this page intentionally left blank)

(this page intentionally left blank)

## Buffer Management - Push/Pop

### push16( buf , npts )

**Prototype** void push16( int \**buf*, long *npts* );  
 PROCEDURE push16( VAR *buf*; *npts* LONGINT);

**Operation** PC(*buf*) >> [STACK], st[1] = (float) st[1]  
 Push 16 bit integer PC buffer to the stack.

**Speed** Approximately 1.3 µseconds per datum.

**Limitations** A maximum of 100 buffers can be pushed on the [STACK].

**Accuracy** NA.

### pushport16( ioport , npts )

**Prototype** void pushport16( int *ioport*, long *npts* );  
 PROCEDURE pushport16( *ioport* INTEGER; *npts* LONGINT);

**Operation** Data from I/O port >> [STACK], st[1] = (float) st[1]. (Port is assumed to be 16 bits wide with the LSB at the address specified and the MSB at the next higher address location; *ioport* must be an even address.)

**Speed** 3.2 µseconds per datum.

**Limitations** A maximum of 100 buffers can be pushed on the [STACK].

**Accuracy** NA.

### pushdisk16( filename )

**Prototype** void pushdisk16( char \**filename* );  
 PROCEDURE pushdisk16( *filename* FSTRG );

**Operation** Data from disk *filename* >> [STACK], st[1] = (float) st[1]. (File data is assumed to be in a LSB first 16-bit wide binary format.)

**Speed** Determined by disk access time.

**Limitations** A maximum of 100 buffers can be pushed on the [STACK].

**Accuracy** NA.

## pushf( buf , npts )

**Prototype**    void                    pushf( float \**buf*, long *npts* );  
 PROCEDURE    pushf( VAR *buf*; *npts* LONGINT);

**Operation**    PC(*buf*) >> [STACK]  
 Pushes single precision floating-point PC buffer to stack.

**Speed**        Approximately 1.9 µseconds per datum.

**Limitations** A maximum of 100 buffers can be pushed on the [STACK].

**Accuracy**    NA.

## pushportf( ioport , npts )

**Prototype**    void                    pushportf( int *ioport*, long *npts* );  
 PROCEDURE    pushportf( *ioport* INTEGER; *npts*  
 LONGINT);

**Operation**    Data from I/O port >> [STACK]. (Port is assumed to be  
 16-bits wide with the LSB at the address specified and the  
 MSB at the next higher address location; *ioport* must be an  
 even address. Floating-point data is read in LSW first,  
 MSW second.)

**Speed**        5.7 µseconds per datum.

**Limitations** A maximum of 100 buffers can be pushed on the [STACK].

**Accuracy**    NA.

## pushdiskf( filename )

**Prototype** void pushdiskf( char \**filename* );  
 PROCEDURE pushdiskf( *filename* FSTRG );

**Operation** Data from disk *filename* >> [STACK].  
 (File data is assumed to be in a LSB first four-byte-wide binary format.)

**Speed** Determined by disk access time.

**Limitations** A maximum of 100 buffers can be pushed on the [STACK].

**Accuracy** NA.

## pushdiska( filename )

**Prototype** void pushdiska( char \**filename* );  
 PROCEDURE pushdiska( *filename* FSTRG );

**Operation** Data from disk *filename* >> [STACK].  
 (File data is assumed to be in floating-point ASCII format, separated by CR-LF's or spaces.)

**Speed** Determined by disk access time.

**Limitations** A maximum of 100 buffers can be pushed on the [STACK].

**Accuracy** NA.

## qpushf( dbn )

**Prototype** void qpushf( int *dbn* );  
 PROCEDURE qpushf( *dbn* INTEGER );

**Operation** dama[*dbn*] >> [STACK]  
 Pushes floating-point DAMA buffer to stack.

**Speed** 0.6 µseconds per datum.

**Limitations** A maximum of 100 buffers can be pushed on the [STACK].

**Accuracy** NA.

## qpushpartf( dbn, spos, npts )

|                    |                                                        |                                                                                                                                                 |
|--------------------|--------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Prototype</b>   | void                                                   | qpushpartf( int <i>dbn</i> ,<br>long <i>spos</i> , long <i>npts</i> );                                                                          |
|                    | PROCEDURE                                              | qpushpartf( <i>dbn</i> INTEGER;<br><i>spos</i> , <i>npts</i> :LONGINT );                                                                        |
| <b>Operation</b>   | ...{dama[ <i>dbn</i> ]}... >> [STACK]                  | Pushes specified portion of floating-point DAMA buffer to stack. The portion pushed starts at index <i>spos</i> and continues for <i>npts</i> . |
| <b>Speed</b>       | 0.6 μseconds                                           | per datum.                                                                                                                                      |
| <b>Limitations</b> | A maximum of 100 buffers can be pushed on the [STACK]. |                                                                                                                                                 |
| <b>Accuracy</b>    | NA.                                                    |                                                                                                                                                 |

## qpush16( dbn )

|                    |                                                        |                                                 |
|--------------------|--------------------------------------------------------|-------------------------------------------------|
| <b>Prototype</b>   | void                                                   | qpush16( int <i>dbn</i> );                      |
|                    | PROCEDURE                                              | qpush16( <i>dbn</i> INTEGER );                  |
| <b>Operation</b>   | (float)( dama[ <i>dbn</i> ] ) >> [STACK]               | Pushes 16-bit integer DAMA buffer to the stack. |
| <b>Speed</b>       | 0.6 μseconds                                           | per datum.                                      |
| <b>Limitations</b> | A maximum of 100 buffers can be pushed on the [STACK]. |                                                 |
| <b>Accuracy</b>    | NA.                                                    |                                                 |

## qpushpart16( dbn, spos, npts )

|                    |                                                        |                                                                                                                                                 |
|--------------------|--------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Prototype</b>   | void                                                   | qpushpart16( int <i>dbn</i> ,<br>long <i>spos</i> , long <i>npts</i> );                                                                         |
|                    | PROCEDURE                                              | qpushpart16( <i>dbn</i> INTEGER;<br><i>spos</i> , <i>npts</i> :LONGINT );                                                                       |
| <b>Operation</b>   | (float)...{dama[ <i>dbn</i> ]}... >> [STACK]           | Pushes specified portion of 16-bit integer DAMA buffer to stack. The portion pushed starts at index <i>spos</i> and continues for <i>npts</i> . |
| <b>Speed</b>       | 0.6 μseconds                                           | per datum.                                                                                                                                      |
| <b>Limitations</b> | A maximum of 100 buffers can be pushed on the [STACK]. |                                                                                                                                                 |
| <b>Accuracy</b>    | NA.                                                    |                                                                                                                                                 |

## **dpush( npts )**

**Prototype** void dpush( long *npts* );  
 PROCEDURE dpush( *npts* LONGINT);

**Operation** (void) >> [STACK] (Use to allocate stack space).

**Speed** 50  $\mu$ seconds.

**Limitations** A maximum of 100 buffers can be pushed on the [STACK].

**Accuracy** NA.

## **pop16( buf )**

**Prototype** void pop16( int \**buf* );  
 PROCEDURE pop16( VAR *buf* );

**Operation** st[1] = (int) st[1], [STACK] >> PC(*buf*)  
 Pops stack into 16-bit integer PC buffer.

**Speed** Approximately 1.4  $\mu$ seconds per datum.

**Limitations** None.

**Accuracy** NA.

## **popport16( ioport )**

**Prototype** void popport16( int *ioport* );  
 PROCEDURE popport16( *ioport* INTEGER );

**Operation** st[1] = (int) st[1], [STACK] >> *ioport*. (Port is assumed to be 16-bit wide with the LSB at the address specified and the MSB at the next higher address location; *ioport* must be an even address.)

**Speed** 3.1  $\mu$ seconds per datum.

**Limitations** None.

**Accuracy** NA.

## popdisk16( filename )

**Prototype**    void                    popdisk16( char \**filename* );  
                   PROCEDURE    popdisk16( *filename* FSTRG );

**Operation**    st[1] = (int) st[1], [STACK] >> disk *filename*.  
                   (File data will be in a LSB first, 16-bit wide binary format.)

**Speed**         Determined by disk access time.

**Limitations**   None.

**Accuracy**     NA.

## popf( buf )

**Prototype**    void                    popf( float \**buf* );  
                   PROCEDURE    popf( VAR *buf* );

**Operation**    [STACK] >> PC(*buf*)  
                   Pops stack into a single precision floating-point PC buffer.

**Speed**         Approximately 2.2  $\mu$ seconds per datum.

**Limitations**   None.

**Accuracy**     NA.

## popportf( ioport )

**Prototype**    void                    popportf( int *ioport* );  
                   PROCEDURE    popportf( *ioport* INTEGER );

**Operation**    [STACK] >> *ioport*. (Port is assumed to be 16 bits wide  
                   with the LSB at the address specified and the MSB at the  
                   next higher address location; *ioport* must be an even  
                   address. Floats are sent to the port LSW first and MSW  
                   second.)

**Speed**         5.6  $\mu$ seconds per datum.

**Limitations**   None.

**Accuracy**     NA.

## popdiskf( filename )

**Prototype** void popdiskf( char \**filename* );  
 PROCEDURE popdiskf( *filename* FSTRG );

**Operation** [STACK] >> disk *filename*.  
 (File data is assumed to be in an LSB first, word-wide binary format.)

**Speed** Determined by disk access time.

**Limitations** None.

**Accuracy** NA.

## popdiska( filename )

**Prototype** void popdiska( char \**filename* );  
 PROCEDURE popdiska( *filename* FSTRG );

**Operation** [STACK] >> disk *filename*.  
 (File data will be in floating-point ASCII format, separated by CR-LF's.)

**Speed** Determined by disk access time.

**Limitations** None.

**Accuracy** NA.

## poppush( sdev, ddev )

**Prototype** void poppush( int *sdev*, int *ddev* );  
 PROCEDURE poppush( *sdev*, *ddev* INTEGER );

**Operation** [STACK].*sdev*>> >> [STACK].*ddev*.  
 (Buffer is popped from AP *sdev* and pushed onto the stack of AP *ddev*.)

**Speed** 11 µseconds per datum.

**Limitations** None.

**Accuracy** NA.



## qpoppart16( *dbn*, *spos* )

**Prototype** void qpoppart16( int *dbn*, long *spos*);  
 PROCEDURE qpoppart16( *dbn* INTEGER; *spos*  
 :LONGINT );

**Operation** (int)( [stack] ) >> ...{dama[*dbn*]}...  
 Pops stack to specified portion of a 16-bit integer DAMA buffer. Stack contents are moved to specified DAMA buffer starting at index *spos*. Be careful, boundary checking is left to the programmer when using **qpoppart16**.

**Speed** 0.6 µseconds per datum.

**Limitations** None.

**Accuracy** NA.

## parse( *expression* )

**Prototype** void parse( char \**expression* );  
 PROCEDURE totop( *expression*: FSTRG );

**Operation** Parses string expression for numerical values and fills top-of-stack buffer with them. Characters interpreted as numeric are: 0..9 and e, E. All others can be used as delimiters.

st[1] = values in *expression* list.

**Speed** 0.6 µseconds per datum.

**Limitations** Memory for new buffer must be available.

**Accuracy** NA.

**Example** Parse("3>4, 12.5/7.2 5") fills the first five top-of-stack elements (starting at index 0) with 3, 4, 12.5, 7.2, and 5.

(this page intentionally left blank)

## Buffer Management - DAMA Operations

### **`_allotf( npts )`**

|                    |                                                                                                                                                                                                                                                                  |                                                                                                   |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| <b>Prototype</b>   | long<br>FUNCTION                                                                                                                                                                                                                                                 | <code>_allotf( long <i>npts</i> );</code><br><code>_allotf( <i>npts</i> LONGINT): LONGINT;</code> |
| <b>Operation</b>   | Allocates a floating-point DAMA buffer of length <i>npts</i> and returns a corresponding DAMA buffer number <i>dbn</i> reference. This command should be used instead of the older <b>allotf</b> , which requires the programmer to keep track of <i>dbn</i> 's. |                                                                                                   |
| <b>Speed</b>       | 70 $\mu$ seconds.                                                                                                                                                                                                                                                |                                                                                                   |
| <b>Limitations</b> | Memory must be available.                                                                                                                                                                                                                                        |                                                                                                   |
| <b>Accuracy</b>    | NA.                                                                                                                                                                                                                                                              |                                                                                                   |

### **`_allot16( npts )`**

|                    |                                                                                                                                                                                                                                                                    |                                                                                                     |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|
| <b>Prototype</b>   | long<br>FUNCTION                                                                                                                                                                                                                                                   | <code>_allot16( long <i>npts</i> );</code><br><code>_allot16( <i>npts</i> LONGINT): LONGINT;</code> |
| <b>Operation</b>   | Allocates a 16-bit integer DAMA buffer of length <i>npts</i> and returns a corresponding DAMA buffer number <i>dbn</i> reference. This command should be used instead of the older <b>allotf16</b> , which requires the programmer to keep track of <i>dbn</i> 's. |                                                                                                     |
| <b>Speed</b>       | 70 $\mu$ seconds.                                                                                                                                                                                                                                                  |                                                                                                     |
| <b>Limitations</b> | $1 \leq dbn \leq 500$ , and memory must be available.                                                                                                                                                                                                              |                                                                                                     |
| <b>Accuracy</b>    | NA.                                                                                                                                                                                                                                                                |                                                                                                     |

### **`allotf( dbn , npts )`**

|                    |                                                       |                                                                                                                            |
|--------------------|-------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| <b>Prototype</b>   | void<br>PROCEDURE                                     | <code>allotf( int <i>dbn</i>, long <i>npts</i> );</code><br><code>allotf( <i>dbn</i> INTEGER; <i>npts</i> LONGINT);</code> |
| <b>Operation</b>   | Declares and allocates a floating-point DAMA buffer.  |                                                                                                                            |
| <b>Speed</b>       | 70 $\mu$ seconds.                                     |                                                                                                                            |
| <b>Limitations</b> | $1 \leq dbn \leq 500$ , and memory must be available. |                                                                                                                            |
| <b>Accuracy</b>    | NA.                                                   |                                                                                                                            |



## getaddr( dbn )

**Prototype** long                   getaddr( int *dbn* );  
 FUNCTION       getaddr( *dbn* INTEGER): LONGINT;

**Operation** Returns the physical (AP2 memory) address of DAMA buffer *dbn* (or 0 if not allocated). Used with **setaddr**, or to determine if a *dbn* has been allocated.

**Speed** 58  $\mu$ seconds.

**Limitations**  $1 \leq dbn \leq 500$ .

**Accuracy** NA.

## setaddr( dbn, addr )

**Prototype** void                   setaddr( int *dbn*, long *addr* );  
 PROCEDURE    setaddr( *dbn* INTEGER, *addr* LONGINT);

**Operation** Forces *dbn* to point to a specific physical (AP2 memory) DAMA buffer address. Useful for switching between different play/record sequence list buffers while I/O is in progress.

**Speed** NA

**Limitations**  $1 \leq dbn \leq 500$ .

**Accuracy** NA.

## disk2dama16( dbn, filename, seekpos )

**Prototype** void                   disk2dama16( int *dbn*, char \**filename*,  
                                           long *seekpos* );  
 PROCEDURE    disk2dama16( *dbn* INTEGER, *filename*  
                                           FSTRG, *seekpos* LONGINT);

**Operation** Data from disk *filename* >> dama[*dbn*].  
 Data is read from *filename* beginning at word location *seekpos* until end-of-file or end-of-DAMA buffer (whichever is first) (File data is assumed to be in an LSB first, 16-bit wide binary format.)

**Speed** Determined by disk access time.

**Limitations** A maximum of 500 DAMA buffers can be loaded.

**Accuracy** NA.

**dama2disk16( dbn, filename, catflag)**

**Prototype**      void                      dama2disk16( int *dbn*, char \**filename*,  
int *catflag*);

PROCEDURE      dama2disk16( *dbn* INTEGER, *filename*  
FSTRG, *catflag* BOOLEAN );

**Operation**      dama[*dbn*] >> disk *filename*.  
*catflag* = 0/false, new file is created for contents of *dbn*.  
*catflag* = 1/true, *dbn* concatenated to end of *filename*.  
(File data will be in an LSB first, 16-bit wide binary  
format.)

**Speed**              Determined by disk access time.

**Limitations**      None.

**Accuracy**        NA.

(this page intentionally left blank)

(this page intentionally left blank)

## Data Generators

### value( *n* )

**Prototype**    void                    value( float *n* );  
                   PROCEDURE    value( *n* SINGLE );

**Operation**    {st[1]} = *n* (all elements of st[1] are set equal to *n*).

**Speed**        0.5 μseconds per datum.

**Limitations** Value must be within representable range.

**Accuracy**    Full.

### fill( *start* , *step* )

**Prototype**    void                    fill( float *start*, float *step* );  
                   PROCEDURE    fill( *start*, *step* SINGLE );

**Operation**    {st[1]} filled with ramp—starting at *start* with incrementer *step*.

**Speed**        0.5 μseconds per datum.

**Limitations** None.

**Accuracy**    Full.

### grand( )

**Prototype**    void                    grand( void );  
                   PROCEDURE    grand;

**Operation**    {st[1]} filled with flat pseudo-random numbers between 0.0 and 1.0.

**Speed**        1.9 μseconds per datum.

**Limitations** Pseudo-random numbers with  $2^{24}$  repeat loop.

**Accuracy**    Full.

## flat( )

**Prototype** void flat( void );  
 PROCEDURE flat;

**Operation** {st[1]} filled with flat pseudo-random numbers between -1.0 and 1.0.

**Speed** 2.3  $\mu$ seconds per datum.

**Limitations** Pseudo-random numbers with  $2^{24}$  repeat loop.

**Accuracy** Full.

## gauss( )/qgauss

**Prototype** void gauss( void );  
 PROCEDURE qgauss;

**Operation** {st[1]} filled with Gaussian numbers with zero mean and unity variance.

**Speed** 17  $\mu$ seconds per datum.

**Limitations** Pseudo-random numbers with  $2^{24}/12$  repeat loop.

**Accuracy** Full.

## seed( sval )

**Prototype** void seed( long *sval* );  
 PROCEDURE seed( *sval* LONGINT);

**Operation** Seeds the AP random number generator with *sval* (= 0 to  $2^{24} - 1$ ).

**Speed** 50  $\mu$ seconds.

**Limitations** None.

**Accuracy** NA.

## tone( freq, srate )

**Prototype** void tone( float *freq*, float *srate* );  
 PROCEDURE tone( *freq*, *srate* SINGLE );

**Operation** {st[1]} =  $\sin(2.0 * \pi * \text{freq} * \text{srate} * i / 1000000.0)$  (i = index of buffer)

**Speed** 0.6  $\mu$ seconds per datum.

**Limitations** Specify *srate* in  $\mu$ seconds;  $\text{freq} < (2 * \text{srate}) - 1$ . Starting phase is random. (Use fill()/cosine() for phase control)

**Accuracy** Harmonic distortions are down at least 80 dB.

## make( index, v )

**Prototype** void make( long *index*, float *v* );  
 PROCEDURE make( *index* LONGINT; *v* SINGLE );

**Operation** {st[1](*index*)} = *v*.  
 Element st[1] with index, *index*, is loaded with value *v*, where  $0 \leq \text{index} < \text{npts}$ .

**Speed** 73  $\mu$ seconds.

**Limitations** None.

**Accuracy** NA.

## makedama16( dbn, index, v )

**Prototype** void makedama16( int *dbn*, long *index*, int *v* );  
 PROCEDURE makedama16( *dbn* INTEGER; *index* LONGINT; *v* INTEGER );

**Operation** {dama[*dbn*](*index*)} = *v*.  
 (The element with index, *index* in DAMA buffer *dbn* is loaded with integer value *v*.)

**Speed** 84  $\mu$ seconds.

**Limitations** None.

**Accuracy** NA.

## makedamaf( *dbn*, *index*, *v* )

**Prototype**     void                    makedamaf( int *dbn*, long *index*, float *v*);  
                   PROCEDURE     makedamaf( *dbn* INTEGER; *index*  
                                          LONGINT; *v* SINGLE );

**Operation**     {dama[*dbn*](*index*)} = *v*.  
 (The element with *index*, *index* in DAMA buffer *dbn* is loaded with floating point value *v*.)

**Speed**            84 μseconds.

**Limitations**    None.

**Accuracy**        NA.

## whatis( *index* )

**Prototype**     float                    whatis( long *index* );  
                   FUNCTION     whatis( *index* LONGINT ):  
                                          SINGLE;

**Operation**     return ==> {st[1](*index*)}.  
 Returns the floating-point value of the specified element of the STACK top buffer.

**Speed**            73 μseconds.

**Limitations**    None.

**Accuracy**        NA.

## Basic Math

### add( )

**Prototype** void add( void );  
PROCEDURE add;

**Operation** {st[2]} = {st[1]} + {st[2]}, [STACK] >> (void)

**Speed** 0.7 μseconds per datum pair.

**Limitations** None.

**Accuracy** Full.

### radd()

**Prototype** void radd( void );  
PROCEDURE radd;

**Operation** {st[2]} = (float 32)( {st[1]} + {st[2]} ), [STACK] >> (void)  
This call is identical to **add** except the resulting sum is protected against 32-bit floating-point roll over. This procedure need only be used for when a resulting sum might overflow the MAXFLOAT boundary. Typically this will only occur when very large averages are being computed.

**Speed** ~2.0 μseconds per datum pair.

**Limitations** None.

**Accuracy** Full.

### subtract( )

**Prototype** void subtract( void );  
PROCEDURE subtract;

**Operation** {st[2]} = {st[1]} - {st[2]}, [STACK] >> (void)

**Speed** 0.7 μseconds per datum pair.

**Limitations** None.

**Accuracy** Full.

## mult( )

**Prototype** void mult( void );  
 PROCEDURE mult;

**Operation** {st[2]} = {st[1]} \* {st[2]}, [STACK] >> (void)

**Speed** 0.7  $\mu$ seconds per datum pair.

**Limitations** None.

**Accuracy** Full.

## absval( )

**Prototype** void absval();  
 PROCEDURE absval;

**Operation** Computes absolute value of st[1] and places result in st[1].

**Speed** ~2.0  $\mu$ seconds per datum.

**Limitations** None.

**Accuracy** Full.

## divide( )

**Prototype** void divide( void );  
 PROCEDURE divide;

**Operation** {st[2]} = {st[1]} / {st[2]}, [STACK] >> (void)

**Speed** 3.0  $\mu$ seconds per datum.

**Limitations** None.

**Accuracy** At least 8 significant digits.

## shift( offset )

**Prototype** void shift( float *offset* );  
 PROCEDURE shift( *offset* SINGLE );

**Operation** {st[1]} = {st[1]} + *offset*

**Speed** 0.6 µseconds per datum.

**Limitations** None.

**Accuracy** Full.

## scale( factor )

**Prototype** void scale( float *factor* );  
 PROCEDURE scale( *factor* SINGLE );

**Operation** {st[1]} = {st[1]} \* *factor*

**Speed** 0.6 µseconds per datum.

**Limitations** None.

**Accuracy** Full.

## inv( )

**Prototype** void inv( void );  
 PROCEDURE inv;

**Operation** {st[1]} = 1.0 / st[1]

**Speed** 2.6 µseconds per datum.

**Limitations** None.

**Accuracy** At least 8 significant digits.

## sqroot( )

**Prototype** void sqroot( void );  
 PROCEDURE sqroot;

**Operation** {st[1]} = {st[1]}<sup>0.5</sup>

**Speed** 4.0 µseconds per datum.

**Limitations** All buffer values must be greater than zero.

**Accuracy** At least 6 significant digits.

## power( x )

**Prototype** void power( float *x* );  
 PROCEDURE power( *x* SINGLE);

**Operation** {st[1]} = {st[1]}<sup>*x*</sup>

**Speed** 7.0 µseconds per datum.

**Limitations** All buffer values must be greater than zero.

**Accuracy** At least 6 significant digits.

## square( )

**Prototype** void square( void );  
 PROCEDURE square;

**Operation** {st[1]} = ({st[1]})<sup>2</sup>

**Speed** 1.0 µsecond per datum.

**Limitations** None.

**Accuracy** Full.

## logten( )

**Prototype** void logten( void );  
 PROCEDURE logten;

**Operation** {st[1]} = log<sub>10</sub>({st[1]})

**Speed** 4.0 µseconds per datum.

**Limitations** All buffer values must be greater than zero.

**Accuracy** At least 6 significant digits.

## loge( )

**Prototype** void loge( void );  
 PROCEDURE loge;

**Operation** {st[1]} = log<sub>e</sub>({st[1]})

**Speed** 4.0 µseconds per datum.

**Limitations** All buffer values must be greater than zero.

**Accuracy** At least 5 significant digits.

## logn( n )

**Prototype** void logn( float *n* );  
 PROCEDURE logn( *n* SINGLE );

**Operation** {st[1]} =  $\log_n(\{st[1]\})$

**Speed** 4.5  $\mu$ seconds per datum.

**Limitations** All buffer values and *n* must be greater than zero.

**Accuracy** At least 5 significant digits.

## alogten( )

**Prototype** void alogten( void );  
 PROCEDURE alogten;

**Operation** {st[1]} =  $10.0^{\{st[1]\}}$

**Speed** 3.2  $\mu$ seconds per datum.

**Limitations** All buffer values must have a magnitude less than 38.0.

**Accuracy** At least 6 significant digits.

## aloge( )

**Prototype** void aloge( void );  
 PROCEDURE aloge;

**Operation** {st[1]} =  $e^{\{st[1]\}}$

**Speed** 3.2  $\mu$ seconds be datum.

**Limitations** All buffer values must have a magnitude less than 88.0.

**Accuracy** At least 6 significant digits.

## maxlim( max )

**Prototype** void maxlim( float *max* );  
 PROCEDURE maxlim( *max* SINGLE );

**Operation** Sets all {st[1]} values greater than *max* to *max*.

**Speed** 1.1 µseconds per datum.

**Limitations** None.

**Accuracy** Full.

## minlim( min )

**Prototype** void minlim( float *min* );  
 PROCEDURE minlim( *min* SINGLE );

**Operation** Sets all {st[1]} values less than *min* to *min*.

**Speed** 1.1 µseconds per datum.

**Limitations** None.

**Accuracy** Full.

## maglim( mag )

**Prototype** void maglim( float *mag* );  
 PROCEDURE maglim( *mag* SINGLE );

**Operation** Limits all {st[1]} values to an absolute value less than or equal to *mag*.

**Speed** 1.7 µseconds per datum.

**Limitations** None.

**Accuracy** Full.

(this page intentionally left blank)

(this page intentionally left blank)

## Other Functions

### **cumsum( )**

**Prototype** void cumsum( void );  
PROCEDURE cumsum;

**Operation** Computes cumulative sum for stack top buffer:

$$\begin{aligned} X_0 &= x_0 \\ X_1 &= x_1 + X_0 \\ X_2 &= x_2 + X_1 \\ &\cdot \cdot \\ &\cdot \cdot \\ X_n &= x_n + X_{n-1} \end{aligned}$$

This call can be used for generating FM type signals.

**Speed** ~2.0  $\mu$ seconds per datum.

**Limitations** None.

**Accuracy** This procedure protects against floating-point overflow, but be careful because this procedure can generate huge numeric results.

### **decimate( fact )**

**Prototype** void decimate( int *fact* );  
PROCEDURE decimate( *fact* :integer);

**Operation**  $st[1](n) = st[1](n * 2^{fact})$  where  $n = 0..npts / (2^{fact})$ .

Decimates *st[1]* and leaves the result on top of the stack. The original buffer is destroyed. The *fact* parameter specifies the decimation factor as a power of two. For example, if *st[1]* is 100 points long, calling **decimate(2)** will reduce *st[1]* to 25 points.

**Speed** ~2.0  $\mu$ seconds per datum.

**Limitations** The size of *st[1]* must be an even multiple of  $2^{fact}$ .

**Accuracy** NA.

## interpol( fact )

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Prototype</b>   | void                    interpol( int <i>fact</i> );<br>PROCEDURE    interpol( <i>fact</i> :integer);                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Operation</b>   | Lengthens st[1] and uses linear interpolation to generate added data elements. The original buffer is destroyed. The <i>fact</i> parameter specifies the interpolation factor as a power of two. For example, if st[1] is 100 points long, calling <b>interpol(3)</b> will increase st[1] to 800 points.<br>NOTE: This call does <b>not</b> extrapolate to generate the last points of the output buffer, instead, it assumes the value of element $st[1](npts) = st[1](npts-1)$ . For example, if <b>interpol(1)</b> is called with:<br>$st[1] = \{ 0.0, 1.0, 2.0, 3.0 \}.$ The resulting stack top buffer would be:<br>$st[1]' = \{ 0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.0 \}.$ |
| <b>Speed</b>       | ~2.0 µseconds per datum.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <b>Limitations</b> | None.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <b>Accuracy</b>    | NA.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |

## foldnadd( art\_thresh )

|                  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Prototype</b> | void                    foldnadd( int <i>art_thresh</i> );<br>PROCEDURE    foldnadd( <i>art_thresh</i> :integer);                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <b>Operation</b> | Divides the top-of-stack buffer into sections and adds them together, producing a fold-and-add effect. The section length is determined by the st[2] buffer, and the result is also added to the st[2], the buffer right below the top-of-stack buffer. The original buffer will remain on the top-of-stack after the operation. This call is typically used for averaging repeated acquisitions recorded into a single buffer.<br>The parameter <i>art_thresh</i> is used for artifact rejection. If any data point magnitude higher than <i>art_thresh</i> is encountered within a buffer section, the <i>entire</i> buffer section is ignored and excluded from addition. |

|                    |                                                          |
|--------------------|----------------------------------------------------------|
|                    | Set <i>art_thresh</i> = 0 to disable artifact rejection. |
| <b>Speed</b>       | NA.                                                      |
| <b>Limitations</b> | None.                                                    |
| <b>Accuracy</b>    | NA.                                                      |
| <b>Example</b>     | See under <b>getnarts</b> .                              |

## getnarts( )

|                    |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|--------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Prototype</b>   | int                   getnarts( void);<br>FUNCTION           getnarts :LONGINT;                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <b>Operation</b>   | Returns the number of artifact rejections encountered during the <b>foldnadd</b> procedure.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
| <b>Speed</b>       | NA.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Limitations</b> | None.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| <b>Accuracy</b>    | NA.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <b>Example</b>     | The following code will average a 204800-point sample by dividing it into 100 2048-point sections, adding them into a 2048-point buffer and averaging them. The original buffer will be discarded, and the averaged sample will be left on the top-of-stack.<br><pre>dpush(2048);       /* length of result buffer */ value(0.0);       /* zero the buffer contents */ qp16(SAMPLE);     /* SAMPLE is the DAMA buffer to                   be averaged, which has 204800                   data points. */ foldnadd(32760); /* this call will section the                   TOS buffer, reject the sections                   containing values larger than                   32760 and add the result to the                   next buffer. */ drop();           /* discard the sample (TOS buffer) */ scale(1.0/(100.0-(float)getnarts()));</pre> |

(this page intentionally left blank)

## Trigonometry

### cosine( )

**Prototype** void cosine( void );  
 PROCEDURE cosine;

**Operation** {st[1]} = cos({st[1]})

**Speed** 3.9  $\mu$ seconds per datum.

**Limitations** None.

**Accuracy** At least 6 significant digits.

### sine( )/qsine

**Prototype** void sine( void );  
 PROCEDURE qsine;

**Operation** {st[1]} = sin({st[1]})

**Speed** 3.9  $\mu$ seconds per datum.

**Limitations** None.

**Accuracy** At least 6 significant digits.

### tangent( )

**Prototype** void tangent( void );  
 PROCEDURE tangent;

**Operation** {st[1]} = tan({st[1]})

**Speed** 14.8  $\mu$ seconds per datum.

**Limitations** None.

**Accuracy** At least 6 significant digits.

## acosine( )

**Prototype** void                   acosine( void );  
 PROCEDURE   acosine;

**Operation** {st[1]} =  $\cos^{-1}(\{st[1]\})$

**Speed** 2.3  $\mu$ seconds per datum.

**Limitations** None.

**Accuracy** For buffer values less than 0.5, error is less than  $10^{-7}$ . For buffer values greater than 0.5, error increases.

## asine( )

**Prototype** void                   asine( void );  
 PROCEDURE   asine;

**Operation** {st[1]} =  $\sin^{-1}(\{st[1]\})$

**Speed** 2.1  $\mu$ seconds per datum.

**Limitations** None.

**Accuracy** For buffer values less than 0.5, error is less than  $10^{-7}$ . For buffer values greater than 0.5, error increases.

## atangent( )

**Prototype** void                   atangent( void );  
 PROCEDURE   atangent;

**Operation** {st[1]} =  $\tan^{-1}(\{st[1]\})$

**Speed** 5.0  $\mu$ seconds per datum.

**Limitations** None.

**Accuracy** At least 3 significant digits.

## atantwo( )

**Prototype** void atantwo( void );  
 PROCEDURE atantwo;

**Operation** {st[2]} =  $\tan^{-1}(\{st[1]\}/\{st[2]\})$ , (void) <<[stack]

**Speed** 10.0  $\mu$ seconds per datum.

**Limitations** None.

**Accuracy** Arctantwo is a full four-quadrant arctangent function.

## qwind( rftime, srate )

**Prototype** void qwind( float *rftime*, float *srate* );  
 PROCEDURE qwind( *rftime*, *srate* SINGLE);

**Operation** {st[1]} is windowed with  $\cos^2$  function. (The onset/offset time is specified in *rftime* in milliseconds and the sampling rate is specified via *srate* in  $\mu$ seconds.) Note that the rise/fall time is defined as the 10% to 90% (on or off) points of the gating function.

**Speed** Dependent on window length, etc.

**Limitations** None.

**Accuracy** Full.

(this page intentionally left blank)

## Complex Operators

### polar( )

|                    |                                                                                                                    |
|--------------------|--------------------------------------------------------------------------------------------------------------------|
| <b>Prototype</b>   | void                    polar( void );<br>PROCEDURE    polar;                                                      |
| <b>Operation</b>   | $st[1] = \arctan2(st[1], st[2]), st[2] = (st[1]^2 + st[2]^2)^{0.5}$<br>Converts CBP from rectangular to polar form |
| <b>Speed</b>       | 16 μseconds per complex datum.                                                                                     |
| <b>Limitations</b> | Must have a Complex Buffer Pair (CBP).                                                                             |
| <b>Accuracy</b>    | At least 3 significant digits.                                                                                     |

### rect( )

|                    |                                                                                                       |
|--------------------|-------------------------------------------------------------------------------------------------------|
| <b>Prototype</b>   | void                    rect( void );<br>PROCEDURE    rect;                                           |
| <b>Operation</b>   | $st[1] = st[2]*\sin(st[1]), st[2] = st[2]*\cos(st[1])$<br>Converts CBP from polar to rectangular form |
| <b>Speed</b>       | 12 μseconds per complex datum.                                                                        |
| <b>Limitations</b> | Must have a Complex Buffer Pair (CBP).                                                                |
| <b>Accuracy</b>    | At least 6 significant digits.                                                                        |

### ximag( )

|                    |                                                                |
|--------------------|----------------------------------------------------------------|
| <b>Prototype</b>   | void                    ximag( void );<br>PROCEDURE    ximag;  |
| <b>Operation</b>   | Extracts odd elements of {st[1]} and pushes them on the stack. |
| <b>Speed</b>       | 0.7 μseconds per complex datum.                                |
| <b>Limitations</b> | Must have a Complex Buffer Pair (CBP).                         |
| <b>Accuracy</b>    | Full.                                                          |

## xreal( )

**Prototype** void xreal( void );  
 PROCEDURE xreal;

**Operation** Extracts even elements of {st[1]} and pushes them on the stack.

**Speed** 0.7 µseconds per complex datum.

**Limitations** Must have a Complex Buffer Pair (CBP).

**Accuracy** Full.

## shuf( )

**Prototype** void shuf( void );  
 PROCEDURE shuf;

**Operation** Shuffles st[2] | st[1] | st[2]... and pushes it on top of the stack. Inverse operation of split().

**Speed** 1.8 µseconds per complex datum.

**Limitations** Must have a Complex Buffer Pair (CBP).

**Accuracy** Full.

## split( )

**Prototype** void split( void );  
 PROCEDURE split;

**Operation** Extracts real and imaginary part of st[1] and pushes them on the stack. Inverse operation of shuf().

**Speed** 1.8 µseconds per complex datum.

**Limitations** None.

**Accuracy** Full.

## cadd( )

|                    |                                                                                                                      |
|--------------------|----------------------------------------------------------------------------------------------------------------------|
| <b>Prototype</b>   | void                    cadd( void );<br>PROCEDURE    cadd;                                                          |
| <b>Operation</b>   | $\{st[4]\} = \{st[4]\} + \{st[2]\}$ , $\{st[3]\} = \{st[3]\} + \{st[1]\}$ ,<br>[STACK] >> (void), [STACK] >> (void). |
| <b>Speed</b>       | 0.9 $\mu$ seconds per datum.                                                                                         |
| <b>Limitations</b> | Must have two Complex Buffer Pairs (CBP) pushed on the stack.                                                        |
| <b>Accuracy</b>    | Full.                                                                                                                |

## cmult( )

|                    |                                                                                                                                                                         |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Prototype</b>   | void                    cmult( void );<br>PROCEDURE    cmult;                                                                                                           |
| <b>Operation</b>   | $\{st[4]\} = \{st[4]\} * \{st[2]\} - \{st[3]\} * \{st[1]\}$ ,<br>$\{st[3]\} = \{st[4]\} * \{st[1]\} + \{st[3]\} * \{st[2]\}$ ,<br>[STACK] >> (void), [STACK] >> (void). |
| <b>Speed</b>       | 1.5 $\mu$ seconds per datum.                                                                                                                                            |
| <b>Limitations</b> | Must have two Complex Buffer Pairs (CBP) pushed on the stack.                                                                                                           |
| <b>Accuracy</b>    | Full.                                                                                                                                                                   |

## cinv( )

|                    |                                                                                                                     |
|--------------------|---------------------------------------------------------------------------------------------------------------------|
| <b>Prototype</b>   | void                    cinv( void );<br>PROCEDURE    cinv;                                                         |
| <b>Operation</b>   | $\{st[2]\} = \{st[2]\} / ( \{st[2]\}^2 + \{st[1]\}^2 )$ ,<br>$st[1] = -\{st[1]\} / ( \{st[1]\}^2 + \{st[1]\}^2 )$ . |
| <b>Speed</b>       | 9.4 $\mu$ seconds per datum.                                                                                        |
| <b>Limitations</b> | Must have a Complex Buffer Pair (CBP) on the stack.                                                                 |
| <b>Accuracy</b>    | Full.                                                                                                               |

(this page intentionally left blank)

## Fast Fourier Transformation

All transformations are limited to radix-2 lengths from 32 to 32768 points (real or complex).

### **cfft( )**

**Prototype** void cfft( void );  
PROCEDURE cfft;

**Operation** Performs in-place complex FFT on st[1] and st[2].  
**Speed** 9.0 milliseconds for 1024-point Complex Buffer Pair.  
**Limitations** Must have a radix-2 Complex Buffer Pair (RCBP).  
**Accuracy** At least 100dB.

### **cift( )**

**Prototype** void cift( void );  
PROCEDURE cift;

**Operation** Performs in-place complex inverse FFT on st[1] and st[2].  
**Speed** 9.0 milliseconds for a 1024-point Complex Buffer Pair.  
**Limitations** Must have a radix-2 Complex Buffer Pair (RCBP).  
**Accuracy** At least 100dB.

### **rfft( )**

**Prototype** void rfft( void );  
PROCEDURE rfft;

**Operation** Performs in-place real FFT on st[1].  
**Speed** 9.0 milliseconds for a 1024-point real buffer.  
**Limitations** Must have a radix-2 buffer (RTB).  
**Accuracy** At least 100dB.

## **rift( )**

**Prototype**    void                    rift( void );  
                   PROCEDURE    rift;

**Operation**    Performs in-place real inverse FFT on st[1] and st[2].

**Speed**        9.0 milliseconds for a 1024-point Complex Buffer Pair.

**Limitations** Must have a radix-2 Complex Buffer Pair (RCBP).

**Accuracy**    At least 100dB.

NOTE: Remember that all Complex Buffer Pairs (CBPs) should be pushed on the stack such that the IMAGINARY buffer is on top and the REAL buffer is just below (i.e., push the REAL buffer on first).

## **hann( )**

**Prototype**    void                    hann( void );  
                   PROCEDURE    hann;

**Operation**    Places a hanning window on st[1].

**Speed**        5.3  $\mu$ seconds per datum.

**Limitations** None.

**Accuracy**    Error less than  $10^{-7}$ .

## **hamm( )**

**Prototype**    void                    hamm( void );  
                   PROCEDURE    hamm;

**Operation**    Places a hamming window on st[1].

**Speed**        5.3  $\mu$ seconds per datum.

**Limitations** None.

**Accuracy**    Error less than  $10^{-7}$ .

(this page intentionally left blank)

## Digital Filtering

### **iir()**

**Prototype** void iir( void );  
PROCEDURE iir;

**Operation** Infinite Impulse Response (IIR) filtering.  
Uses the Direct Form II Equation:

$$y_k = b_0x_k + b_1x_{k-1} + \dots + b_mx_{k-m} \\ - a_1y_{k-1} - a_2y_{k-2} - \dots - a_my_{k-m}$$

or in the z-domain,

$$\frac{Y(z)}{X(z)} = \frac{b_0 + b_1z^{-1} + \dots + b_mz^{-m}}{1 + a_1z^{-1} + \dots + a_mz^{-m}}$$

The following buffers must be stacked before calling **iir**:

st[1] = data to be filtered: { $x_k$ }  
st[2] = a[0...m] coefficients  
st[3] = b[0...m] coefficients

The value of a[0] is always one. The filter input ( $x_k$ ) and output ( $y_k$ ) values are assumed to be zero for  $k$  less than zero. Upon completion, **iir** returns with the coefficient buffers unaltered and the stack-top buffer containing the filtered output data { $y_k$ }.

NOTE: **iir** is blocking-sensitive.

**Speed** 0.7 µseconds per tap-datum.

**Limitations**  $m \geq 3$ . Numerical instability may result for  $m > 7$ .

**Accuracy** Full.

### **\_iir()**

**Prototype** void \_iir( void );  
PROCEDURE \_iir;

**Operation** Infinite Impulse Response (IIR) filtering with initial conditions. Performs same operation as **iir** except allows specification of initial conditions. The following buffers must be stacked before calling **\_iir**:

st[1] = data to be filtered:  $\{x_k\}$   
 st[2] = a[0...m] coefficients  
 st[3] = b[0...m] coefficients  
 st[4] = y[0...m] initial values  
 st[5] = x[0...m] initial values

The initial value buffers, x and y, contain the input ( $x_k$ ) and output ( $y_k$ ) values for  $k \leq 0$ , i.e.,

$$x[0...m] = \{x_0, x_{-1}, \dots, x_{-m}\}$$

$$y[0...m] = \{y_0, y_{-1}, \dots, y_{-m}\}$$

(y[0] is an undefined place holder). Upon completion, **\_iir** returns with the coefficient buffers unaltered and the stack-top buffer containing the filtered output data  $\{y_k\}$ . Also, the initial value buffers are automatically loaded with the final values of the input and output data. This facilitates filtering of large data streams divided into multiple smaller buffers. The initial values are set to zero for the first input data buffer, and then automatically loaded with the proper values for each subsequent section of the input data.

NOTE: **\_iir** is blocking-sensitive.

**Speed**

0.7  $\mu$ seconds per tap-datum.

**Limitations**

$m \geq 3$ . Numerical instability may result if  $m > 7$ .

**Accuracy**

Full.

**Example**

The section of code below illustrates how **\_iir** is used to filter input data split between two buffers. A third order filter is used:

$$y_k = 3x_k + 2x_{k-1} + 5x_{k-2} - 2y_{k-1} - 4y_{k-2} - y_{k-3}$$

```

/* Place initial conditions on stack */
dpush(4);          /* Push on space for x init cond 's */
value(0.0);        /* and initialize to zero */

dpush(4);          /* Push on space for y initial cond's */
value(0.0);        /* and initialize to zero */

/* Place 'b' coefficients on stack */
dpush(4);
make(0,3.0);
make(1,2.0);
make(2,5.0);
make(3,0.0);

/* Place 'a' coefficients on stack */
dpush(4);
make(0,1.0);      /* a[0] is always 1 */
make(1,2.0);
make(2,4.0);

```

```

make(3,1.0);

qpush( SAMP1); /* Push input data buffer 1 ont */
_iir(); /* Run first part of input data through
filter (init. cond's are zero) */
qpopf( FILT1); /* Store filtered output data in buffer */
qpush( SAMP2); /* Push input data buffer 2 onto stack */
_iir(); /* Run second part of input data through
filter (init. cond's already set from
previous _iir call) */
qpopf( FILT2); /* Store filtered output data in buffer */

```

## **fir()**

**Prototype** void fir( void );

PROCEDURE fir;

**Operation** Finite Impulse Response (FIR) filtering.  
Uses the Direct Form II Equation.

$$y_k = b_0x_k + b_1x_{k-1} + \dots + b_mx_{k-m}$$

or in the z-domain,

$$\frac{Y(z)}{X(z)} = b_0 + b_1z^{-1} + \dots + b_mz^{-m}$$

The following buffers must be stacked before calling **fir**:

st[1] = data to be filtered:  $\{x_k\}$

st[2] = b[0...m] coefficients

Filter input values ( $x_k$ ) are assumed to be zero for  $k$  less than zero. Upon completion, **fir** returns with the coefficient buffer unaltered and the stack-top buffer containing the filtered output data  $\{y_k\}$ .

NOTE: **fir** is blocking-sensitive.

**Speed** 0.3  $\mu$ seconds per tap-datum.

**Limitations**  $m \geq 3$ .

**Accuracy** Full.

**\_fir( )****Prototype** void \_fir( void );

PROCEDURE \_fir;

**Operation** Finite Impulse Response (FIR) filtering with initial conditions. Performs same operation as **fir** except allows for initial conditions specification. The following buffers must be stacked before calling **\_fir**:st[1] = data to be filtered:  $\{x_k\}$ 

st[2] = b[0...m] coefficients

st[3] = x[0...m] initial values

The initial value buffer x contains the values of the input ( $x_k$ ) for  $k \leq 0$ , i.e.

$$x[0..m] = x_0, x_{-1}, \dots, x_{-m}$$

Upon completion, **\_fir** returns with the coefficient buffers unaltered and the stack-top buffer containing the filtered output data  $\{y_k\}$ . Also, the initial value buffer is automatically loaded with the final values of the input data. This facilitates filtering of large data streams divided into multiple smaller buffers: The initial values are set to zero for the first input data buffer, then automatically loaded with the proper values for each subsequent section of the input data.

NOTE: **\_fir** is blocking-sensitive.**Speed** 0.3  $\mu$ seconds per tap-datum.**Limitations**  $m \geq 3$ .**Accuracy** Full.

## **reverse( )**

**Prototype**    void                    reverse( void );  
                  PROCEDURE    reverse;

**Operation**    Reverses the order of the stack-top array elements:  
                   $\{st[1](k)\} = \{st[1](N - k)\}, k = 0...N$   
                  where  $N$  is the number of elements in the array.

**Speed**        0.55  $\mu$ seconds per datum.

**Limitations**   None.

**Accuracy**    NA.

## Summation Functions

### sum( )

**Prototype** float sum( void );  
 FUNCTION sum :SINGLE;

**Operation** Returns sum of all values within {st[1]}.

**Speed** 0.6  $\mu$ seconds per datum.

**Limitations** None.

**Accuracy** Full.

### average( )

**Prototype** float average( void );  
 FUNCTION average :SINGLE;

**Operation** Returns average of all values within {st[1]}.

**Speed** 0.6  $\mu$ seconds per datum.

**Limitations** None.

**Accuracy** Full.

### maxval( )

**Prototype** float maxval( void );  
 FUNCTION maxval :SINGLE;

**Operation** Returns maximum value within {st[1]}.

**Speed** 1.1  $\mu$ seconds per datum.

**Limitations** None.

**Accuracy** Full.

## minval( )

**Prototype** float minval( void );  
 FUNCTION minval :SINGLE;

**Operation** Returns minimum value within {st[1]}.

**Speed** 1.1 µseconds per datum.

**Limitations** None.

**Accuracy** Full.

## maxmag( )

**Prototype** float maxmag( void );  
 FUNCTION maxmag :SINGLE;

**Operation** Returns maximum magnitude within {st[1]}.

**Speed** 2.2 µseconds per datum.

**Limitations** None.

**Accuracy** Full.

## maxval\_()

**Prototype** long maxval\_( void );  
 FUNCTION maxval\_ :LONGINT;

**Operation** Returns index of maximum value within {st[1]}.

**Speed** 1.1 µseconds per datum.

**Limitations** None.

**Accuracy** Full.

## minval\_( )

**Prototype** long minval\_( void );  
 FUNCTION minval\_ :LONGINT;

**Operation** Returns index of minimum value within {st[1]}.

**Speed** 1.1 µseconds per datum.

**Limitations** None.

**Accuracy** Full.

## **maxmag\_()**

**Prototype**    long                    maxmag\_( void );  
                  FUNCTION        maxmag\_ :LONGINT;

**Operation**    Returns index of maximum magnitude within {st[1]}.

**Speed**        2.2  $\mu$ seconds per datum.

**Limitations**   None.

**Accuracy**    Full.

NOTE: The indices returned by **maxval\_**, **minval\_** and **maxmag\_** are relative to the lower end of any block placed on the buffer. For example, if block(10,20) were imposed and then a call to **minval\_** returned the number 6, this would mean the 7th element (buf[16]) in the specified block was the lowest. If **minval\_** had returned zero, the first element within the block would be smallest (buf[10]).

(this page intentionally left blank)

## Play - Record Operations

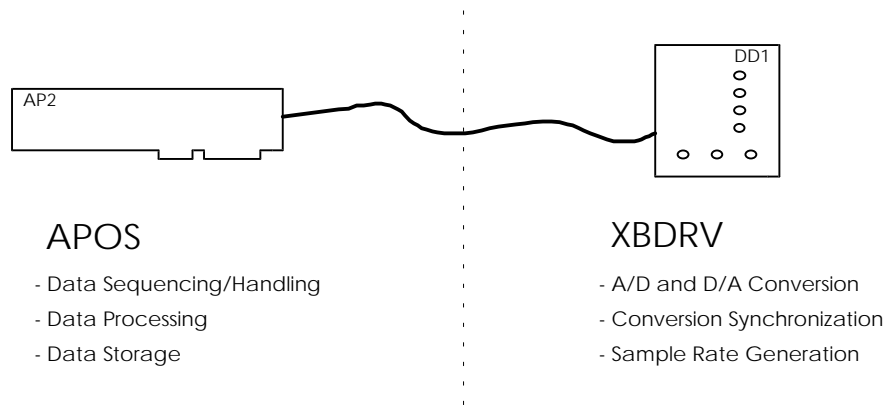
### Overview

The AP2 array processor is capable of performing powerful analog interfacing functions via TDT's external A/D and D/A conversion XBUS modules. These modules are available in a variety of performance grades that are equipped with various features. The modules are interfaced to the AP2 through the optical interface located on the AP2 card. This high-speed interface is capable of sending and receiving conversion data at 12.5 million bits per second. This bit rate translates to simultaneous A/D and D/A sampling rates of up to 500,000 samples per second.

APOS contains a powerful set of **play** and **record** procedures for sending and receiving data on the AP2's optical interface. These procedures support simple, single channel I/O operations as well as complex, segmented multi-channel I/O operations.

From the AP2/APOS "view point" these **play** and **record** procedures are simply data handling procedures. As data is received on the optical interface it is processed as specified by the **record** procedure. As data is requested on the optical interface, it is sequenced out as specified by the **play** procedure. This non-device-specific data handling method allows the same set of APOS operations to work with all TDT analog interface modules.

Just as APOS is indifferent to what happens to data beyond its optical interface, the XBUS analog interface modules use and produce data regardless of its source and destination. These modules handle the physical parameters associated with data conversion, such as sampling rate, overall conversion length, device synchronization and, of course, the final A/D and D/A conversion. The following figure illustrates how TDT's System II hardware divides the duties associated with analog interfacing:



It is up to the programmer to match the operation programmed in APOS to the operation programmed in XBDRV. For example, if you generate a 100-point single cycle of a sine wave in APOS and specify it as the **play** buffer, you could then program a DA1 to play 1,000,000 points at a 10 $\mu$ s sampling rate. This would give you 10,000 periods of a 1KHz sine wave from your D/A converter.

## Using APOS *Play* and *Record* Procedures

APOS play and record procedures work with 16-bit integer DAMA buffers. These buffers are allocated in APOS DAMA space using the **allot16** command. Data is moved between these DAMA buffers and the STACK using the **qpush16** and **qpop16** commands.

The following steps are typically used to play a waveform:

1. Allocate the DAMA buffer using the **allot16** command.
2. Load the waveform to be played onto the STACK or build it using APOS commands.
3. Move the waveform to the allocated DAMA buffer using the **qpop16** command.
4. Specify the DAMA buffer for playing using the **play** command.

5. Set up, arm and trigger the XBUS analog interface module using XBDRV commands (see *XBDRV Software Reference*).

A waveform is recorded using similar steps as illustrated below:

1. Allocate a DAMA buffer using the **allot16** command.
2. Specify the DAMA buffer as the destination for recording data using the **record** command call.
3. Set up, arm, and trigger the XBUS analog interface module using XBDRV commands.
4. When the conversion is complete move the recorded DAMA buffer to the STACK using the **qpush16** command.
5. Process the data as desired.

Note that the play and record steps outlined above can be combined in the same program to get simultaneous playing and recording. Also, APOS contains more powerful play and record procedures for performing segmented and multi-channel operations. They are named as **seqplay** and **seqrecord** for sequenced play and sequenced record, respectively. These procedures possess a high level of data processing power and will be explained in detail later in this document.

Both **play** and **record** operations will be described. If you only have an A/D module, disregard references to the **play** operations. If you only have an D/A module, disregard references to the **record** operations.

For information on getting started on using the play and record features of APOS, along with the XBDRV routines for your particular data acquisition modules, refer to *Applications Using TDT System II*. This document covers how the two driver packages work together, and discusses many sample programs in detail.

## Special Considerations and Comparisons

There are some special considerations associated with the APOS-AP2 side of play (D/A) and record (A/D) procedures. When called, all procedures invoke interrupt-driven data handling routines on the AP2 Array Processor. These routines are run (in the background) each time a connected XBUS device issues a request for data processing attention. Asynchronous to this, the AP2 also remains available for host-directed signal processing tasks. Because the interrupt routine will "cycle-steal," less time is available for foreground processing.

The following comparison illustrates the relative advantages and limitations of various play and record procedures:

### **play, record, fastrecord**

These calls are simply data handling procedures. They are useful for single channel operations on a single waveform or when high throughput rates are required. These procedures also can be used to handle multi-channel data, however, shuffling considerations must be addressed. The **play** and **record** calls can process up to 750,000 samples per second combined and support bi-directional sampling at up to 350 kHz. If bi-directional operations require a sample rate beyond 350 kHz, the **fastrecord** procedure must be used. Refer to the *CycleFactor* calculation described below to see which combination of play and/or record you must use.

**seqplay, seqrecord, dplay, drecord, mplay, mrecord**

All of these calls invoke 'sequenced' data processing routines, which support multi-channel, multi-segment play and record operations. Although these procedures are very powerful, they can process only 400,000 to 500,000 samples per second. Also, when concurrent A/D and D/A modules are operating, neither channel can be made to process more than 250,000 samples per second. For example, suppose simple **play** is used in conjunction with **seqrecord**. Even if the record speed is very slow, the sample rate for the single D/A channel cannot exceed 250 kHz. Use this rule in conjunction with the *CycleFactor* calculation shown below to determine the feasibility of a particular application.

**Cycle Usage Calculation**

The following formula can be used to calculate the usage-factor for your application. The equation will give an approximate indication of the percentage of processing time the AP2 will spend handling the D/A and A/D data. *CycleFactors* greater than 100 will not run correctly.

$$CycleFactor = \frac{N_{sequenced}}{4,000} + \frac{N_{simple}}{7,500} \%$$

Where:  $N_{sequenced}$  is the total number of samples per second being processed by a *sequenced* routine and  $N_{simple}$  is the total number of samples per second being handled by a *simple* processing routine.

For example, suppose you are playing from a single DAC channel at 200 kHz simply using **play** while recording on three A/D channels at 32 kHz using **seqrecord**. The corresponding cycle factor would be calculated as follows:

$$\begin{aligned} CycleFactor &= \frac{(3 \cdot 32,000)}{4,000} + \frac{200,000}{7,500} \% \\ &= 50.67\% \end{aligned}$$

This indicates that about half of the AP2's processing power will be used to handle the D/A and A/D data.

## Simple Play and Record

The single channel **play** and **record** procedures should be used whenever multi-channel operations are *not* required, especially if any concurrent processing (e.g., averaging) is to be done. You can combine simple and complex operations. For example, you can use the simple **play** procedure in conjunction with the **seqrecord** procedure.

The simple **play** and **record** procedures use a single DAMA buffer as a data source and destination, respectively. Data will flow to or from the specified buffer until the end of the buffer is reached. If more data is received or more data is requested, the data flow will loop back to the start of the buffer. NOTE: The **play** and **record** procedures work with 16-bit integer DAMA buffers only.

### play( dbn )

**Prototype**    void                    play( int *dbn* );  
                   PROCEDURE    play( *dbn* :INTEGER);

**Operation**    Initializes a single channel, single buffer play from the specified DAMA buffer.

**Example**        The following program will generate a 100Hz tone to be played from a single DAC channel with a 100kHz sampling rate. The waveform is 10,000 points long and has cos<sup>2</sup> gating on the onset and offset.

```
#define DACBUF 1            /* Logical name for DAMA buffer */
apinit(APa);                /* Initialize the AP2 */
allot16(DACBUF,10000);      /* Allocate DAMA buffer */

/*Build waveform to be played */
dpush(10000);               /* Create stack space */
tone(100.0, 10.0);         /* 100Hz sine, 10us srata */
scale(32000.0);            /* Scale for 16 bit D/A */
qwind(20.0, 10.0);         /* 20ms rise/fall window */

qpop16(DACBUF);            /* Pop tone into DAMA */
play(DACBUF);             /* Specify as play buffer */
{ D/A Converter Instructions }
```

## record( *dbn* )

**Prototype**    void                    record( int *dbn* );  
                   PROCEDURE    qrecord( *dbn* :INTEGER);

**Operation**    Initializes a single channel, single buffer record into the specified DAMA buffer.

**Example**        This sample program records 4096 points from a single ADC channel and computes a magnitude spectrum in dB.

```
#define ADCBUF 1            /* Logical name for DAMA buffer */
apinit(APa);                /* Initialize the AP2 */
allot16(ADCBUF,4096);        /* Allocate DAMA buffer */
record(ADCBUF);            /* Specify record buffer */

{ A/D Converter Instructions }
{ wait till A/D done        }

qpush16(ADCBUF);            /* Push recorded data on STACK */

/*Compute the magnitude spectrum...*/
hann();
rfft();
polar();
drop();
logten();
scale(20.0);

{ Magnitude spectrum is on
  top of stack }
```

## **fastrecord( dbn )**

**Prototype**    void                    fastrecord( int *dbn* );  
                  PROCEDURE    fastrecord( *dbn* :INTEGER);

**Operation**    Same as **record**, but will not loop when the end of the DAMA buffer is reached. CAUTION: This procedure will write out past the end of the specified DAMA buffer if too many A/D points are received.

## Dual-Channel Play and Record

For dual-channel play and record operations, the **dplay** and **drecord** procedures are provided. These procedures are similar to the single channel procedures above, except that buffers for two channels can be specified as arguments. NOTE: The **dplay** and **drecord** procedures work with 16-bit integer DAMA buffers only.

### **dplay( dbn1, dbn2 )**

**Prototype**    void                    dplay( int *dbn1*, int *dbn2* );  
                   PROCEDURE    dplay(*dbn1* :INTEGER,  
                                           *dbn2* :INTEGER);

**Operation**    Initializes a dual-channel, dual buffer play from the specified DAMA buffers.

### **drecord( dbn1, dbn2 )**

**Prototype**    void                    drecord( int *dbn1*, *dbn2* );  
                   PROCEDURE    drecord(*dbn1* :INTEGER,  
                                           *dbn2* :INTEGER);

**Operation**    Initializes a dual channel, dual buffer record into the specified DAMA buffers.

## Sequenced Play and Record

The sequenced play and record procedures, **seqplay** and **seqrecord**, are specially-designed APOS routines which support complex multi-channel analog I/O operations. Signals can be played out of or recorded into buffer segments, combined in any order desired. This unique scheme results in the ability to perform real time signal processing via double buffering and synthesis of specific stimulus patterns.

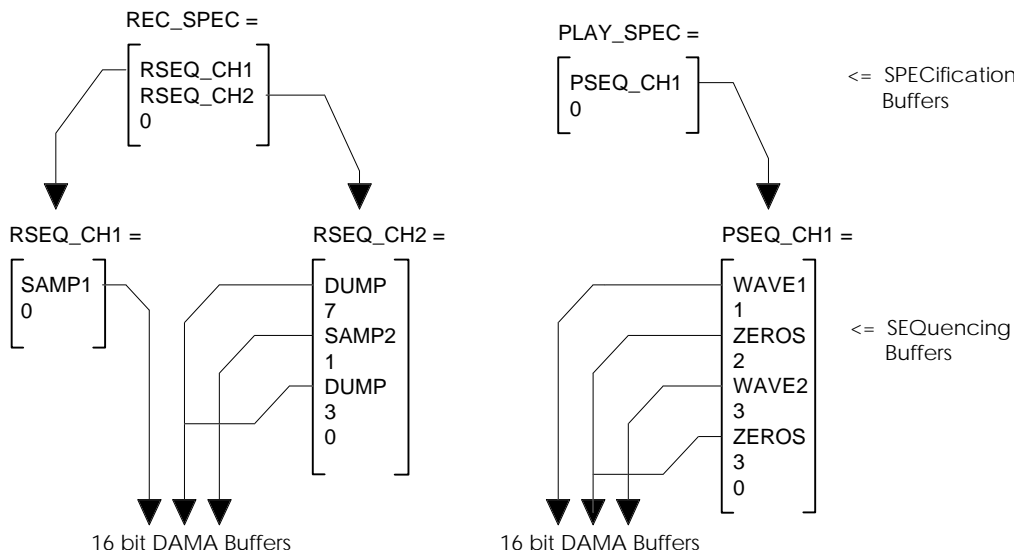
To use the sequenced play and record procedures, you must set up DAMA buffers containing a *play specification* list, and/or a *record specification* list, and *channel sequence* lists for each active channel. These buffers tell the AP2 how to manage data flow as it is sent to, and received from, the optical interface.

The **seqplay** procedure uses a *play specification* list. In this list, each entry indicates the DAMA buffer number of a *channel sequence* list. There must be a *channel sequence* list for each D/A channel to be used. The *channel sequence* list entries indicate the DAMA buffer numbers of waveform segments followed by a repeat factor for that segment.

The **seqrecord** procedure operates in the same way. The *record specification* list points to the *channel sequence* lists for each A/D channel. In turn, each *channel sequence* list points to the buffer segments to be used in the recording. The entries in the *specification* and *sequence* lists are simply DAMA buffer numbers. However, constant names can be defined and used for each buffer number to avoid confusion.

The figure below illustrates how the *specification* lists and *channel sequence* lists might be set up for a typical application. All DAMA buffer numbers have been assigned appropriate names.

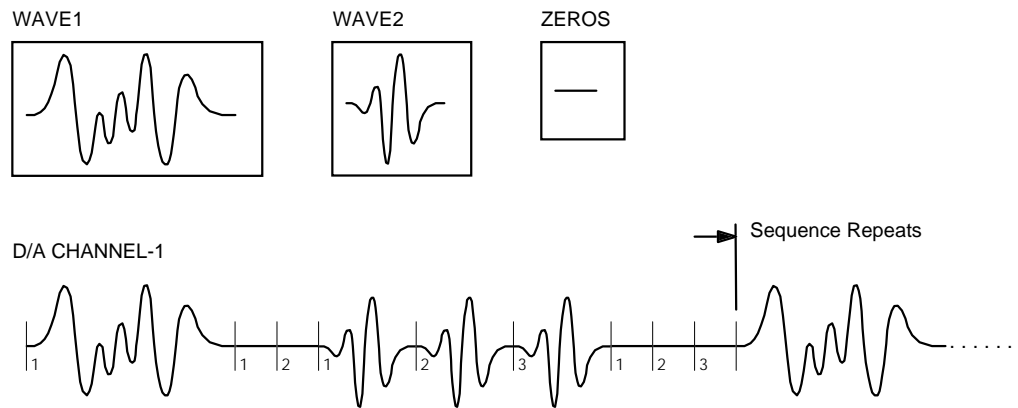
In this example, data buffer segment numbers, each followed by a repeat factor, are entered into *channel SEquence* lists for each I/O channel to be used. The *channel SEquence* list of DAMA buffer numbers are entered into their respective play and record *SPECification* lists. The end of each *SPECification* and *SEquence* lists is designated with a zero ("0").



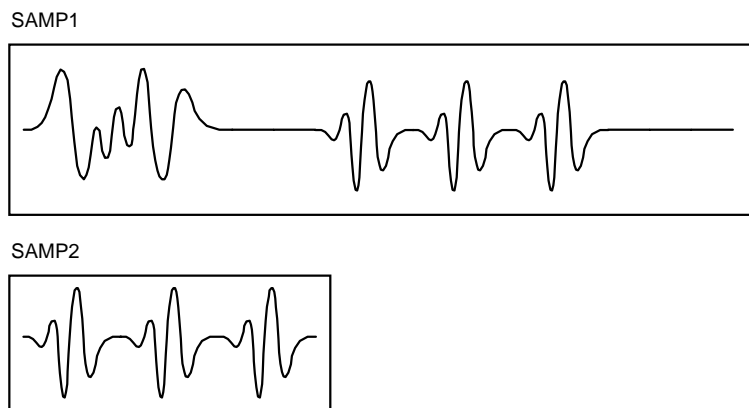
The DAMA buffers in the figure above are allocated using the *allot16* command. The contents of the SPEC and SEQ list buffers are first created on the AP2 stack using the *make* command, and then popped into their appropriate DAMA buffer locations. The contents of the play buffer segments are also created or loaded onto the AP2 stack, and then popped into DAMA. Of course, the record buffer contents do not need to be initialized. This process is clarified by programming examples in the document *Applications using TDT System II*.

When an external module begins D/A conversion, data will flow from the buffers listed in PSEQ\_CH1. Each buffer is repeated according to the number of times specified before proceeding down the list. When data flow reaches the end of the channel sequence list (signified by a zero), it loops back to the beginning of the sequence, until the external module stops requesting data.

Refer to the specification buffers described above and consider the following possible waveform contents for the WAVE1, WAVE2 and ZEROS buffers. The resulting output on D/A channel 1 would be as shown.



Meanwhile, A/D channels 1 and 2 record data into the buffers listed in RSEQ\_CH1 and RSEQ\_CH2, respectively. The diagram below shows the contents of the SAMP1 and SAMP2 record buffers at the end of the conversion sequence if the output of D/A channel 1 is fed into both A/D channels 1 and 2. The short DUMP buffer was used to 'throw out' unneeded data samples for channel 2 so that only the middle part of the signal is stored in SAMP2.



There are a few items to consider when using the **seqrecord** and **seqplay** procedures. Because the AP2 performs its play and record data handling on an interrupt basis, it "cycle-steals" from any APOS operations running concurrently. Therefore, one should always use the simple play and record procedures whenever complex sequenced I/O operations are not required. You can combine simple and sequenced operations, for example, using the simple **play** procedure in conjunction with the **seqrecord** procedure. The **seqplay** and **seqrecord** procedures work with 16-bit integer DAMA buffers only.

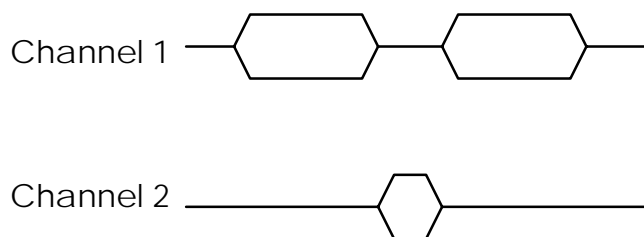
Two status procedures, **playseg** and **recseg**, return the DAMA buffer number presently being played from or recorded into on the specified channel. These procedures are used in double buffering applications to wait until one buffer is completed before proceeding to the next.

## seqplay( dbn )

**Prototype**    void                    seqplay( int *dbn* );  
 PROCEDURE    seqplay( *dbn* :INTEGER);

**Operation**    Initializes a multi-channel, sequenced play operation using the Play Specification List buffer *dbn*.

**Example**        The code shown below generates a narrow band noise signal with a temporal gap on Channel 1 and a short tone blip positioned in the center of the gap on Channel 2.



Both the pre- and post noises are 100 milliseconds long, while the tone blip and gap are 20 milliseconds in duration. All signals are scaled to play from a 16-bit D/A converter:

```

#define SRATE 20.0      /* Sampling rate (20 us) */

/* Define logical names for SPEC, SEQ and data buffers */
#define PLAY_SPEC      1
#define PLAY_SEQ1     2
#define PLAY_SEQ2     3
#define NOISEa        4
#define NOISEb        5
#define TONEBLIP      6
#define ZEROS         7

/* Calculate number of points using SRATE */
noise_pts = (int)(100.0 * 1000.0 / SRATE);
tone_pts = (int)(20.0 * 1000.0 / SRATE);
onemilli_pts = (int)(1000.0 / SRATE);

apinit(APa);          /* Initialize AP2 */

/*Allocate DAMA space for all buffers */
allot16(PLAY_SPEC, 10);
allot16(PLAY_SEQ1, 10);
allot16(PLAY_SEQ2, 10);
allot16(NOISEa, noise_pts);
allot16(NOISEb, noise_pts);
allot16(TONEBLIP, tone_pts);
allot16(ZEROS, onemilli_pts);

/* Build the play specification list */
dpush(10);
make(0,PLAY_SEQ1);
make(1,PLAY_SEQ2);
make(2,0);
qpop16(PLAY_SPEC);   /* Pop it to DAMA */

/* Build the channel-1 sequence list */
dpush(10);
make(0,NOISEa);
make(1,1);
make(2,ZEROS);
make(3,20);
make(4,NOISEb);
make(5,1);
make(6,0);
qpop16(PLAY_SEQ1);   /* Pop it to DAMA */

/* Do same for D/A channel 2 */
dpush(10);
make(0,ZEROS);
make(1,100);
make(2,TONEBLIP);
make(3,1);
make(4,ZEROS);
make(5,100);
make(6,0);
qpop16(PLAY_SEQ2);

/* Build noise signals: */
dpush(noise_pts);
gauss();
scale(32000.0/maxmag());
qwind(2.0,SRATE);
qpop16(NOISEa);

dpush(noise_pts);
gauss();
scale(32000.0/maxmag());
qwind(2.0,SRATE);
qpop16(NOISEb);

/* Build tone blip */
dpush(tone_pts);
tone(1000.0,SRATE);

```

```

scale(32000.0);
qwind(2.0,SRATE);
qpop16(TONEBLIP);

/* Build zeros buffer for gaps */
dpush(onemilli_pts);
value(0.0);
qpop16(ZEROS);

/* Initialize the play operation with
the play specification created above */

seqplay(PLAY_SPEC);

{ D/A converter instructions }
{ Tell the D/A to play       }
{ 220 * onemilli_pts points }

```

## seqrecord( dbn )

**Prototype**    void                    seqrecord( int *dbn* );  
PROCEDURE    seqrecord( *dbn* :INTEGER);

**Operation**    Initializes a multi-channel, sequenced record process using the Record Specification List buffer *dbn*.

**Example**        This program samples two ADC channels into double buffers (four buffers total). The A/D data is time-averaged 16 times and magnitude spectra are calculated and compared for the two channel averages.

```

/* This is the radix-2 number for FFTs */
#define NPTS 8192

/*Define logical names for all buffers */
#define REC_SPEC            1
#define REC_SEQ1            2
#define REC_SEQ2            3
#define SAMP1a              4
#define SAMP1b              5
#define SAMP2a              6
#define SAMP2b              7

#define AVG1                10
#define AVG2                11

apinit(APa);                /* Initialize AP2 */

/* Allocate DAMA space for 16-bit buffers */
allot16(REC_SPEC, 10);
allot16(REC_SEQ1, 10);
allot16(REC_SEQ2, 10);
allot16(SAMP1a, NPTS);
allot16(SAMP1b, NPTS);
allot16(SAMP2a, NPTS);
allot16(SAMP2b, NPTS);

/*Allocate DAMA space for floating-point buffers */
allotf(AVG1,NPTS);
allotf(AVG2,NPTS);

/*Build the record specification list */
dpush(10);

```

```

make(0,REC_SEQ1);
make(1,REC_SEQ2);
make(2,0);
qpop16(REC_SPEC);

/*Build the channel sequence for channel-1 */
dpush(10);
make(0,SAMP1a)
make(1,1);
make(2,SAMP1b);
make(3,1);
make(4,0);
qpop16(REC_SEQ1);

/*Do same for A/D channel-2 */
dpush(10);
make(0,SAMP2a)
make(1,1);
make(2,SAMP2b);
make(3,1);
make(4,0);
qpop16(REC_SEQ2);

/*Initialize averaging buffers */
dpush(NPTS);
value(0.0);
qdup();
qpopf(AVG1);
qpopf(AVG2);

/*Initialize the record operation with
the record specification created above*/

seq_record(REC_SPEC);

{ A/D Converter Instructions      }
{ Tell it to record NPTS*16 points}

/* Use double buffering to perform real
time averaging of 16 buffers total */

for(i=0; i<8; i++)
{
/* place current average sum on stack */
qpushf(AVG2);
qpushf(AVG1);

/* Wait until "a" buffers are full... */
while( rec_seg(1)==SAMP1a );

/* add SAMP1a to AVG1 */
qpush16(SAMP1a);
add();
qpopf(AVG1);

/* add SAMP2a to AVG2 */
qpush16(SAMP2a);
add();
qpopf(AVG2);

/* place current average sum on stack */
qpushf(AVG2);
qpushf(AVG1);

/* Wait until "b" buffers are full... */
while( rec_seg(1)==SAMP1b );

/* add SAMP1b to AVG1 */
qpush16(SAMP1b);
add();
qpopf(AVG1);

/* add SAMP2b to AVG2 */
qpush16(SAMP2b);
add();

```

```

    qpopf(AVG2);
}

/* Scale sum to average... */
qpushf(AVG1);
scale(1.0/16.0);

/* and compute spectrum */
hann();
rfft();
polar();
drop();
logten();
scale(20.0);

/* Do same for channel 2 */
qpushf(AVG2);
scale(1.0/16.0);
hann();
rfft();
polar();
drop();
logten();
scale(20.0);

subtract();    /*subtract the two spectra */

{Spectral difference is on top of stack }

```

## mrecord( dbn )

**Prototype**    void                    mrecord( int *dbn* );  
 PROCEDURE    mrecord( *dbn* :INTEGER);

**Operation**    Initializes a multi-channel, record process using the Channel Specification List buffer *dbn*. This list contains record buffer numbers for each A/D channel used. Unlike *seqrecord*, only one record buffer per channel can be specified (no buffer sequences).

**Example**        This program samples four ADC channels into four buffers. The data from the four channels is summed together and a magnitude spectrum is calculated.

```

/* This is the radix-2 number for FFTs */
#define NPTS 8192

/*Define logical names for all buffers */
#define CHAN_SPEC        1
#define SAMP1            2
#define SAMP2            3
#define SAMP3            4
#define SAMP4            5

#define SUM               6

apinit(APa);            /* Initialize AP2 */

/* Allocate DAMA space for 16-bit buffers */
allot16(CHAN_SPEC, 10);
allot16(SAMP1, NPTS);
allot16(SAMP2, NPTS);

```

```

allot16(SAMP3, NPTS);
allot16(SAMP4, NPTS);

/*Allocate DAMA space for floating-point buffer */
allotf(SUM,NPTS);

/*Build the record channel specification */
dpush(10);
make(0,SAMP1);
make(1,SAMP2);
make(2,SAMP3);
make(3,SAMP4);
make(4,0);
qpop16(CHAN_SPEC);

/*Initialize averaging buffers */
dpush(NPTS);
value(0.0);
qdup();
qpopf(SUM);

/*Initialize the record operation with
the record channel specification created above*/

mrecord(CHAN_SPEC);

{ A/D Converter Instructions:
  Tell it to use 4 channels and begin
  converting for NPTS samples. }

/* Wait until record buffers are full... */
while( {A/D status is inactive} );

/* place SUM onto stack */
qpushf(SUM);

/* add SAMP1 - SAMP4 together */
qpush16(SAMP1);
add();

qpush16(SAMP2);
add();

qpush16(SAMP3);
add();

qpush16(SAMP4);
add();

/* Store sum in SUM for other use...*/
qpopf(SUM);

/* and compute spectrum of SUM */
hann();
rfft();
polar();
drop();
logten();
scale(20.0);

{Spectrum of sum is on top of stack and the
summed time waveforms is in the floating
point buffer SUM}

```

## **mplay( dbn )**

- Prototype**    void                    mplay( int *dbn* );  
                   PROCEDURE    mplay( *dbn* :INTEGER);
- Operation**    Initializes a multi-channel, play process using the Channel Specification List buffer *dbn*. This list contains DAMA play buffer numbers for each D/A channel used. Unlike *seqplay*, only one play buffer per channel can be specified (no buffer sequences).
- Example**        This program plays four DAMA buffers out of four D/A channels.

```

#define NPTS 8192

/*Define logical names for all buffers */
#define CHAN_LIST 1
#define BUF1 2
#define BUF2 3
#define BUF3 4
#define BUF4 5

apinit(APa);            /* Initialize AP2 */

/* Allocate DAMA space for 16-bit buffers */
allot16(CHAN_LIST, 10);
allot16(BUF1, NPTS);
allot16(BUF2, NPTS);
allot16(BUF3, NPTS);
allot16(BUF4, NPTS);

/*Build the play channel specification list */
dpush(10);
make(0,BUF1);
make(1,BUF2);
make(2,BUF3);
make(3,BUF4);
make(4,0);
qpop16(CHAN_LIST);

/*Initialize DAMA play buffers with tones of
increasing frequency */
freq=1000.0;
for(i=0;i<4;i++)
{
  dpush(NPTS);
  tone(freq,SRATE);        /* make tone                    */
  qpop16(BUF1+i);        /* move to DAMA            */
  freq = freq*1.1;        /* increase freq by 10%    */
}

/*Initialize the play operation with
the play channel list created above: */
mplay(CHAN_LIST);

{ D/A Converter Instructions:
  Tell it to use 4 channels and begin
  converting for NPTS samples. }

/* Wait until playback is complete... */
while( {D/A status is active} );

```

## Pause Feature in Sequence Play

In APOS Version 2.0 and later, a pause feature in sequence play has been added. During a sequenced play it may be desirable to prevent buffers from being played to one or several channels until a key is hit or a certain condition is met. The pause feature provides a means to do this by playing a series of zeros during the pause. In programming this is accomplished by assigning a negative sign to the buffer number in the play sequence list. Upon reaching the negative sign during sequenced play, the sequence is halted and zeros are sent to the D/A channel. Zeros continue to be sent until the sequence is resumed by calling either **pfireone** or **pfireall**. **ppausestat** will return the current pause status of a D/A channel.

### pfireone( chan )

**Prototype**    void                    pfireone( int *chan* );  
                   PROCEDURE    pfireone( *chan* :INTEGER);

**Description**   *chan*        D/A channel number.

**Operation**     Resumes sequenced play of DAMA buffer number to D/A channel *chan* after the sequence has been paused with the pause feature.

**Example**        Below, part of the **seqplay()** example has been modified to illustrate the pause feature. By using the pause feature, it becomes very easy to halt playing at one or several channels.

```

/* Build the channel-1 sequence list */
dpush(10);
make(0,NOISEa);
make(1,1);
make(2,-NOISEb);
make(3,1);
make(4,0);
qpop16(PLAY_SEQ1);        /* Pop it to DAMA */
/* Remember: This buffer is to be played to D/A    */
/* channel 1    */

/* This section need not to be changed */

seqplay(PLAY_SPEC);

{ D/A converter instructions }
{ Tell the D/A to play        }
{ 220 * onemill_pts points   }
do { }while(!ppausestat(1)); /* wait until D/A channel 1
                              reaches the pause points */

delay(500);

```

```

    pfireone(1); /* Resume play to D/A channel 1 */
/* pfireall() can also be used in this case */

```

After NOISEa is played (ppausestat(1) returns 1), Ap2 will first send zeros to D/A channel 1 for about 500 milliseconds, and continue to send NOISEb to D/A channel 1 when **pfireone()** is called.

## pfireall( )

**Prototype** void pfireall( void );  
PROCEDURE pfireall;

**Operation** Resumes play of play sequences to all halted D/A channels after the D/A channels have been paused with the pause feature.

**Example** Refer to the example under **pfireone()**.

## ppausestat( chan )

**Prototype** int ppausestat( int *chan* );  
FUNCTION ppausestat( *chan* :INTEGER):  
INTEGER;

**Description** *chan* D/A channel number.

**Operation** Returns the pause status of the D/A channel *chan*. This function returns a 0 if the play sequence is being played to D/A channel *chan*. Otherwise, it returns 1.

**Example** Refer to the example under **pfireone()**.

## Auxiliary Functions

### playseg( chan )

**Prototype**    int                    playseg( int *chan* );  
 FUNCTION        playseg( *chan* :INTEGER):  
                                  INTEGER;

**Operation**    Returns the DAMA buffer number of the waveform segment currently being sent out to the D/A. This procedure is used whenever the PC program needs to know which DAMA buffer segment is being played from. During sequenced play, returns the actual DAMA buffer number being played (not the play list DAMA buffer).

**Example**        Refer to programs on the Examples disk to see how **playseg/recseg** is used to track play/record activity in order to facilitate double buffering.

### recseg( chan )

**Prototype**    int                    recseg( int *chan* );  
 FUNCTION        recseg( *chan* :INTEGER):  
                                  INTEGER;

**Operation**    Returns the DAMA buffer number of the waveform segment currently being recorded into. This procedure is used whenever the PC program needs to know which DAMA buffer segment is being recorded into.

**Example**        See **playseg**.



## chgplay( dbn )

**Prototype**    void                    chgplay( int *dbn* );  
                  PROCEDURE    chgplay( *dbn* :INTEGER);

**Operation**    Changes the currently played DAMA buffer to DAMA buffer *dbn*.

**Example**        /\* Generate two different tones and save them in  
                  TONE1 and TONE2 \*/  
                  /\* Setup D/A hardware \*/  
                  play(TONE1)            /\* play buffer TONE1        \*/  
                  delay(1000)           /\* play about 1 second    \*/  
                  chgplay(TONE2);      /\* stop playing buffer TONE1  
                                          and start playing buffer TONE2    \*/

The above program fragment plays TONE1 for about one second, then switches to TONE2.

## Miscellaneous Functions

### **plotmap( xx1, yy1, xx2, yy2)**

**Prototype** void plotmap( int xx1, int yy1, int xx2, int yy2)  
PROCEDURE plotmap( xx1, yy1, xx2, yy2 :INTEGER)

**Operation** (void) >> [STACK], st[1] = linedraw coord( st[2] )  
Creates a new stack buffer containing the line-draw coordinates necessary to plot the top-of-stack buffer in a PC graphics window.

The **plotmap** call executes one of two possible algorithms based on the number of elements in the buffer to be plotted and the size of the target plot rectangle. If the number of elements to be plotted is less than twice the number of pixels across the plot rectangle, a straightforward Y-scale-mapping is performed. A stack-top buffer is returned, equal in length to the plotted buffer containing screen Y-coordinates for use in line-draw type commands.

If the buffer to be plotted is large, having more than twice as many elements as there are pixels across the plot rectangle, a special X-compression procedure is employed. With this procedure, the stack-top buffer created will always have twice as many elements as there are pixels across the plot rectangle. A plot is generated by performing line-draw commands connecting these points.

Parameters:

xx1 = left window coordinate.

yy1 = top window coordinate.

xx2 = right window coordinate.

yy2 = bottom window coordinate.

**Speed** NA.

**Limitations** None

**Accuracy** NA.

**Example**

The following sample 'C' procedure illustrates how a general purpose plot procedure can be written using the **plotmap** APOS function. Note that the code does not check the size of the buffer to be plotted, because the **plotmap** procedure will never return a buffer with more elements than twice the pixels across the plot area.

```

void tplot( int xx1, int yy1, int xx2, int yy2)
{
    float xp,xs;
    int npts,i;
    int *buf;

    buf = (int *)malloc((xx2-xx1) << 2);
    if(buf == NULL)
    {
        printf("\n\nOut of memory running PLOT
                !!!\n\n");
        exit(0);
    }

    plotmap(xx1,yy1,xx2,yy2);

    npts = topsize();
    xs = (float)(xx2-xx1)/npts;
    xp = (float)xx1;
    pop16(buf);

    moveto(xx1,buf[0]);
    for(i=0; i<npts; i++)
    {
        lineto((int)xp,buf[i]);
        xp += xs;
    }
    free(buf);
}

```

## plotwith( *xx1*, *yy1*, *xx2*, *yy2*, *ymin*, *ymax*)

**Prototype** void plotwith( int *xx1*, int *yy1*, int *xx2*, int *yy2*,  
float *ymin*, float *ymax*)  
PROCEDURE plotwith( *xx1*, *yy1*, *xx2*, *yy2* :INTEGER;  
*ymin*, *ymax*: SINGLE);

**Operation** (void) >> [STACK], st[1] = linedraw coord( st[2] )  
Similar to **plotmap** except for the Y-Scaling, which is specified in the call.

X-Scaling will be identical to **plotmap**, with x-compression being employed when needed.

Parameters:

*xx1* = left window coordinate.

*yy1* = top window coordinate.

*xx2* = right window coordinate.

*yy2* = bottom window coordinate.

*ymin* = minimum Y value in plot window.

*ymax* = maximum Y value in plot window.

**Speed** NA.

**Limitations** None

**Accuracy** NA.

**Example** The following sample 'C' procedure illustrates how a general purpose plot procedure can be written using the **plotwith** APOS function. Note that the code does not check the size of the buffer to be plotted because the **plotwith** procedure will never return a buffer with more elements than twice the pixels across the plot area.

```
void tplot( int xx1, int yy1, int xx2, int yy2,
           float ymin, float ymax)
{
    float xp, xs;
    int npts, i;
    int *buf;

    buf = (int *)malloc((xx2-xx1) << 2);
```

```

if(buf == NULL)
{
    printf("\n\nOut of memory running PLOT
          !!!\n\n");
    exit(0);
}

plotwith(xx1, yy1, xx2, yy2, ymin, ymax);

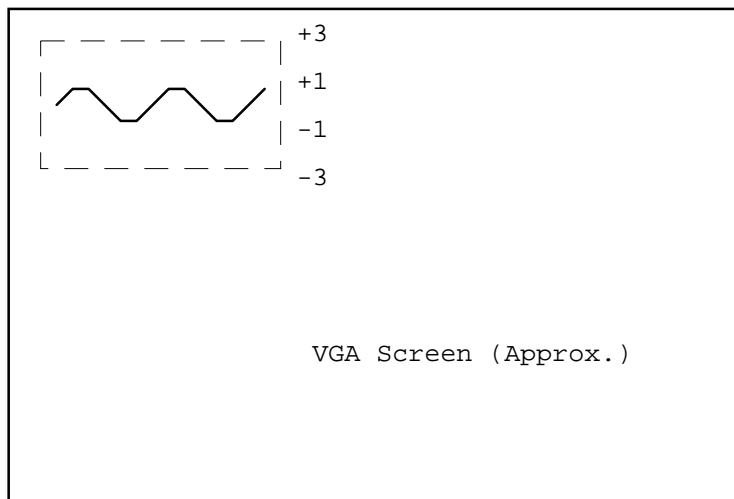
npts = topsize();
xs = (float)(xx2-xx1)/npts;
xp = (float)xx1;
pop16(buf);

moveto(xx1,buf[0]);
for(i=0; i<npts; i++)
{
    lineto((int)xp,buf[i]);
    xp += xs;
}
free(buf);
}

```

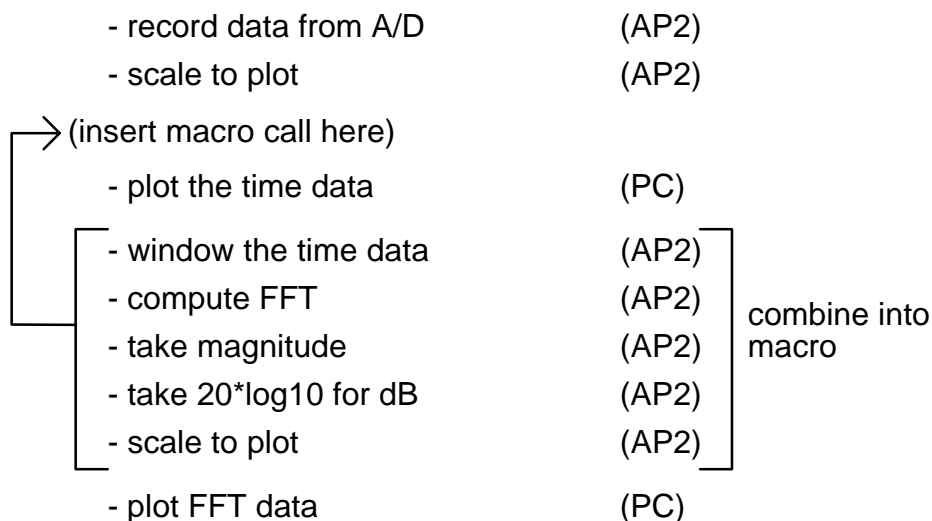
If the above procedure was called with a unity sine wave on top of the stack, the plot shown below would be generated:

*plotwith( 100, 50, 300, 150, -3.0, 3.0);*



## Macro Functions

Macro functions provide a way to combine commonly used APOS command sequences into a user-defined macro, which is executed by a single call from the user program. APOS macro command sequences are stored and run from the AP2's DAMA. So, after a macro is called, the PC is free to return to the user program while the AP2 runs the APOS macro sequence. The advantages are program simplification, and that the PC can do its own tasks while the AP2 is working away on large arrays. For example, a typical spectrum analyzer routine would involve the following:



Without macros, the PC stands idle until all the APOS procedures in the bracketed section are finished. If the bracketed items are combined into a macro inserted as shown, the AP2 can work on FFT spectrum calculations while the PC is plotting the time data. By the time the PC finishes the time data, the FFT data will be ready (or almost ready) for plotting.

Normally, an APOS call from the user program must be completed before another APOS call can be executed. So, if a long sequence of APOS calls is encountered in a program, the PC can experience considerable idle time while waiting for the AP2 to complete its computations. Macros provide a way to use the PC idle time, thereby effectively providing some multi-tasking ability.

A macro is recorded into an integer DAMA buffer, and typically defined at the beginning of the user's program for use later in the program. If *nstp* is the number of steps in the macro, an integer DAMA buffer of size  $30 \times (2+nstp)$  must be allocated, e.g., for a 10-step macro, a 360 points integer DAMA buffer should be allocated:

```
allot16(MAC_BUF, 360);
```

MAC\_BUF must be less than 100. Each macro step in DAMA has a *label value*, a *fixed-point value*, and a *floating-point value* which serve as a way to jump to a specific step, and pass arguments.

Recording is initiated by the **recordmac** procedure, which tells the AP2 to record all subsequent APOS commands into a designated DAMA buffer until **endmac** is called. The macro can then be called anywhere later in the program using **runmac**. As a programming example, let's record the steps for the FFT spectrum calculations mentioned above to a macro:

```
allot16(FFT_MAC, 300);
recordmac(FFT_MAC);
{
    hann();
    rfft();
    polar();
    drop();
    logten();
    scale(20.0);
}
endmac();
```

This macro is run later in the program by calling:

```
runmac(FFT_MAC);
```

Advanced macro commands to perform looping, incrementing of arguments, and conditional branching are also available. Reference definitions and descriptions are given below for all APOS macro commands. Refer to *Digital Signal Processing Applications Using TDT System II* and software examples on disk for examples on usage and applications of macro commands.

## **recordmac( dbn )**

**Prototype**    void                    recordmac( int *dbn* );  
                   PROCEDURE    recordmac( *dbn* :INTEGER);

**Operation**    Initiates recording of a macro into a specified DAMA buffer. Subsequent APOS calls are not executed, but instead are recorded into *dbn* until **endmac** is called. Note that any non-APOS calls between **recordmac** and **endmac** will not be part of the macro.

**Speed**            NA.

**Limitations**    *dbn* < 100

**Accuracy**        NA.

## **endmac( )**

**Prototype**    void                    endmac( void );  
                   PROCEDURE    endmac;

**Operation**    Ends recording of a macro started with **recordmac**. Some functionality testing of the macro is done, and APOS error messages will be returned if the macro cannot be run.

**Speed**            NA.

**Limitations**    None

**Accuracy**        NA.

## runmac( *dbn* )

**Prototype**    void                    runmac( int *dbn* );  
                   PROCEDURE    runmac( *dbn* :INTEGER);

**Operation**    Executes a macro from DAMA buffer *dbn*. A valid macro command sequence must reside in buffer *dbn*.

**Speed**         NA.

**Limitations**   *dbn* < 100

**Accuracy**     NA.

## whatmac( )

**Prototype**    int                    whatmac( void );  
                   FUNCTION        whatmac :INTEGER;

**Operation**    Called outside of macros to return the macro number (= *dbn* used with **recordmac**) currently running on the AP2.

**Speed**         NA.

**Limitations**   None

**Accuracy**     NA.

## whatstep( )

**Prototype**    int                    whatstep( void );  
                   FUNCTION        whatstep :INTEGER;

**Operation**    Called outside of macros to return the macro step label value currently executing.

**Speed**         NA.

**Limitations**   None

**Accuracy**     NA.

## stopmac( )

**Prototype** void stopmac( void );  
 PROCEDURE stopmac;

**Operation** Called external to macros to interrupt macro execution and return control of the AP2 to the PC program.

**Speed** NA.

**Limitations** None

**Accuracy** NA.

## ls( steplbl )

**Prototype** void ls( int *steplbl* );  
 PROCEDURE ls( *steplbl* :INTEGER);

**Operation** Assigns a specific label value to the next macro step for reference within the macro. The label serves as a position reference for looping, and as a way to use the value returned by the labeled step as an argument in another step.

**Speed** NA.

**Limitations** None

**Accuracy** NA.

## stepval24( macdbn, steplbl )

**Prototype** long stepval24( int *macdbn*, int *steplbl* );  
 FUNCTION stepval24( *macdbn*, *steplbl* :INTEGER):  
 LONGINT;

**Operation** Called outside macro to access a fixed point value returned by the specified step in a macro.

*macdbn* Macro DAMA buffer number.

*steplbl* Macro step label assigned using **ls**.

**Speed** NA.

**Limitations** None

**Accuracy** NA.

## stepvalf( macdbn, steplbl )

**Prototype** float stepvalf( int *macdbn*, int *steplbl* );  
 FUNCTION stepvalf( *macdbn*, *steplbl* :INTEGER): SINGLE;

**Operation** Called outside macro to access a floating-point value returned by the specified step in a macro.  
*macdbn* Macro DAMA buffer number.  
*steplbl* Macro step label (assigned using **ls**).

**Speed** NA.

**Limitations** None

**Accuracy** NA.

## loopn( steplbl, n )

**Prototype** void loopn( int *steplbl*, int *n* );  
 PROCEDURE loopn( *steplbl*, *n* :INTEGER);

**Operation** Causes macro execution to loop back to the specified macro step.  
*steplbl* Macro step label assigned using **ls**.  
*n* Number of loops (excluding the first).

**Speed** NA.

**Limitations** None

**Accuracy** NA.

## jump( steplbl )

**Prototype** void jump( int *steplbl* )  
 PROCEDURE jump( *steplbl* :INTEGER);

**Operation** Causes macro execution to jump to the specified macro step.  
*steplbl* Macro step label (assigned using **ls**).

**Speed** NA.

**Limitations** None

**Accuracy** NA.

## usestepval( arg# )

|                    |                                                                                                                                                                                                                                                                                                                                                       |                                                                      |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------|
| <b>Prototype</b>   | void<br>PROCEDURE                                                                                                                                                                                                                                                                                                                                     | usestepval( int <i>arg#</i> );<br>usestepval( <i>arg#</i> :INTEGER); |
| <b>Operation</b>   | Indirectly assigns an argument value to the next macro procedure, or the macro procedure call following <b>usestepval</b> uses the value returned by a labeled step as one of its arguments--                                                                                                                                                         |                                                                      |
|                    | <ul style="list-style-type: none"> <li>• <i>arg#</i> is the argument position (in the next step) that gets the indirect value. For example, <i>arg2</i> indicates the second argument position in the next step is assigned the indirect value.</li> <li>• The <u>step label</u> of the step that holds the value is used as the argument.</li> </ul> |                                                                      |
| <b>Speed</b>       | NA.                                                                                                                                                                                                                                                                                                                                                   |                                                                      |
| <b>Limitations</b> | None                                                                                                                                                                                                                                                                                                                                                  |                                                                      |
| <b>Accuracy</b>    | NA.                                                                                                                                                                                                                                                                                                                                                   |                                                                      |

## usedamalist( arg# )

|                    |                                                                                                                                                                                                                                                                                                                                                                        |                                                                        |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------|
| <b>Prototype</b>   | void<br>PROCEDURE                                                                                                                                                                                                                                                                                                                                                      | usedamalist( int <i>arg#</i> );<br>usedamalist( <i>arg#</i> :INTEGER); |
| <b>Operation</b>   | Indirectly assigns of an argument value to the next macro procedure, or, the macro procedure call following <b>usedamalist</b> uses the next value in a DAMA buffer list as one of its arguments--                                                                                                                                                                     |                                                                        |
|                    | <ul style="list-style-type: none"> <li>• <i>arg#</i> is the argument position (in the next step) that gets the indirect value.</li> <li>• The <i>dbn</i> of the DAMA list is used as the argument of the procedure call statement.</li> <li>• A pointer increments to the next value in the DAMA list <i>dbn</i> each time a particular list is referenced.</li> </ul> |                                                                        |
| <b>Speed</b>       | NA.                                                                                                                                                                                                                                                                                                                                                                    |                                                                        |
| <b>Limitations</b> | None                                                                                                                                                                                                                                                                                                                                                                   |                                                                        |
| <b>Accuracy</b>    | NA.                                                                                                                                                                                                                                                                                                                                                                    |                                                                        |

## counter24( start, incr )

|                    |                                                                                                                                                                                                                                                                                                                                                                          |
|--------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Prototype</b>   | void                    counter24( long <i>start</i> , long <i>incr</i> );<br>PROCEDURE    counter24( <i>start</i> ; <i>incr</i> :LONGINT);                                                                                                                                                                                                                              |
| <b>Operation</b>   | Sets up a counter whose value can be used as an argument in other procedure calls within the macro. Each time a <b>counter24</b> statement is called, its counter value increments by <i>incr</i> starting from <i>start</i> . The value of the counter is referenced by its step label (assigned using <b>ls</b> immediately preceding the <b>counter24</b> statement). |
| <b>Speed</b>       | NA.                                                                                                                                                                                                                                                                                                                                                                      |
| <b>Limitations</b> | None                                                                                                                                                                                                                                                                                                                                                                     |
| <b>Accuracy</b>    | NA.                                                                                                                                                                                                                                                                                                                                                                      |

## counterf( start, incr )

|                    |                                                                                                                                                                                                                                                                                                                                                                        |
|--------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Prototype</b>   | void                    counterf( float <i>start</i> , float <i>incr</i> );<br>PROCEDURE    counterf( <i>start</i> ; <i>incr</i> :SINGLE);                                                                                                                                                                                                                             |
| <b>Operation</b>   | Sets up a counter whose value can be used as an argument in other procedure calls within the macro. Each time a <b>counterf</b> statement is called, its counter value increments by <i>incr</i> starting from <i>start</i> . The value of the counter is referenced by its step label (assigned using <b>ls</b> immediately preceding the <b>counterf</b> statement). |
| <b>Speed</b>       | NA.                                                                                                                                                                                                                                                                                                                                                                    |
| <b>Limitations</b> | None                                                                                                                                                                                                                                                                                                                                                                   |
| <b>Accuracy</b>    | NA.                                                                                                                                                                                                                                                                                                                                                                    |

## if24( *steplbl1*, *cc*, *steplbl2*, *jmpstep* )

**Prototype**    void            if24( int *steplbl1*, int *cc*, int *steplbl2*,  
                                         int *jmpstep* );  
**PROCEDURE**   if24( *steplbl1*, *cc*, *steplbl2*, *jmpstep*  
                                         :INTEGER);

**Operation**    Compares two fixed-point values referenced by two step labels and jumps to a third step label if the comparison is true.

*steplbl1,2*    Macro step labels that reference values to be compared.

*cc*            Comparison code:

EQ            1

NEQ          2

LT            3

GT            4

LT\_EQ        5

GT\_EQ        6

*jmpstep*      Macro step label to jump to if comparison is true.

**Speed**        NA.

**Limitations** None

**Accuracy**    NA.

## iff( *steplbl1*, *cc*, *steplbl2*, *jmpstep* )

**Prototype**    void            iff( int *steplbl1*, int *cc*, int *steplbl2*,  
                                         int *jmpstep* );  
**PROCEDURE**   iff( *steplbl1*, *cc*, *steplbl2*, *jmpstep*  
                                         :INTEGER);

**Operation**    Compares two floating-point values referenced by two step labels and jumps to a third step label if the comparison is true.

*steplbl1,2*    Macro step labels that reference values to be compared.

|                |                                                    |
|----------------|----------------------------------------------------|
| <i>cc</i>      | Comparison code:                                   |
|                | EQ 1                                               |
|                | NEQ 2                                              |
|                | LT 3                                               |
|                | GT 4                                               |
|                | LT_EQ 5                                            |
|                | GT_EQ 6                                            |
| <i>jmpstep</i> | Macro step label to jump to if comparison is true. |

**Speed** NA.

**Limitations** None

**Accuracy** NA.

## constant( *lval*, *fval* )

**Prototype** void constant( long *lval*, long *fval* );  
PROCEDURE constant( *lval*; *fval* :LONGINT);

**Operation** Used to provide a comparison constant for the **if24** and **iff** macro functions. The step label for this call is referenced as one of the step labels in **if24** or **iff** when comparison against a constant value is required.

*lval* 24-bit fixed-point value used by **if24**

*fval* Floating-point value used by **iff**

**Speed** NA.

**Limitations** None

**Accuracy** NA.

# APOS Function Index

- \_allot16()*, 39
  - \_allotf()*, 39
  - \_fir()*, 75
  - \_iir()*, 72
- A**
- absval()*, 50
  - acosine()*, 62
  - add()*, 49
  - allot16()*, 40
  - allotf()*, 39
  - aloge()*, 53
  - alogten()*, 53
  - ap\_active()*, 4, 20
  - ap\_present()*, 19
  - ap\_select()*, 3, 19
  - apinit()*, 3, 19
  - asine()*, 62
  - atangent()*, 62
  - atantwo()*, 63
  - average()*, 77
- B**
- block()*, 8, 25
- C**
- cadd()*, 67
  - cat()*, 24
  - catn()*, 24
  - cfft()*, 69
  - chgplay()*, 104
  - cift()*, 69
  - cinvt()*, 67
  - cmult()*, 67
  - constant()*, 118
- cosine()*, 61
  - counter24()*, 116
  - counterf()*, 116
  - cumsum()*, 57
- D**
- dama2disk16()*, 42
  - deallot()*, 40
  - decimate()*, 57
  - disk2dama16()*, 41
  - divide()*, 50
  - dplay()*, 89
  - dpush()*, 33
  - drecord()*, 89
  - drop()*, 23
  - dropall()*, 23
  - dupn()*, 24
- E**
- endmac()*, 111
  - extract()*, 26
- F**
- fastrecord() /b*, 88
  - fill()*, 45
  - fir()*, 74
    - with initial conditions*, 75
  - flat()*, 46
  - foldnadd()*, 58
  - freewords()*, 20
- G**
- gauss()*, 46
  - getaddr()*, 41
  - getnarts()*, 59
- H**
- hamm()*, 70
  - hann()*, 70
- I**
- if24()*, 117
  - iff()*, 117
  - iir()*, 72
    - with initial conditions*, 72
  - interpol()*, 58
  - inv()*, 51
- J**
- jump()*, 114
- L**
- loge()*, 52
  - logn()*, 53
  - logten()*, 52
  - loopn()*, 114
  - ls()*, 113
- M**
- maglim()*, 54
  - make()*, 47
  - makedama16()*, 47
  - makedamaf()*, 48
  - maxlim()*, 54
  - maxmag()*, 78
  - maxmag\_()*, 79
  - maxval()*, 77
  - maxval\_()*, 78, 79
  - minlim()*, 54
  - minval()*, 78
  - minval\_()*, 78, 79
  - mplay()*, 99

*mrecord()*, 97*mult()*, 50**N***noblock()*, 8, 25**P***parse()*, 37*pfireall()*, 101*pfireone()*, 100*play()*, 86*chgplay()*, 104*dplay()*, 89*mplay()*, 99*playcount()*, 103*playseg()*, 102*seqplay()*, 93*playcount()*, 103*playseg()*, 102*plotmap()*, 105*plotwith()*, 107*polar()*, 65*pop16()*, 33*popdisk16()*, 34*popdiska()*, 35*popdiskf()*, 35*popf()*, 34*popport16()*, 33*popportf()*, 34*poppush()*, 35*power()*, 52*ppausestat()*, 101*push16()*, 29*pushdisk16()*, 29*pushdiska()*, 31*pushdiskf()*, 31*pushf()*, 30*pushport16()*, 29*pushportf()*, 30**Q***qdup()*, 24*qgauss()*, 46*qpop16()*, 36*qpopf()*, 36*qpoppart16()*, 37*qpoppartf()*, 36*qpush16()*, 32*qpushf()*, 31*qpushpart16()*, 32*qpushpartf()*, 32*qrand()*, 45*qsine()*, 61*qwind()*, 63**R***radd()*, 49*reccount()*, 103*record()*, 87*drecord()*, 89*fastrecord()*, 88*mrecord()*, 97*reccount()*, 103*recseg()*, 102*seqrecord()*, 95*recordmac()*, 111*recseg()*, 102*rect()*, 65*reduce()*, 26*reverse()*, 76*rfft()*, 69*rifft()*, 70*runmac()*, 112**S***scale()*, 51*seed()*, 46*seqplay()*, 93*seqrecord()*, 95*setaddr()*, 41*shift()*, 51*shuf()*, 66*sine()*, 61*split()*, 66*sqroot()*, 51*square()*, 52*stackdepth()*, 21*stepval24()*, 113*stepvalf()*, 114*stopmac()*, 113*subtract()*, 49*sum()*, 77*swap()*, 23**T***tangent()*, 61*tone()*, 47*topsize()*, 20*totop()*, 25*trash()*, 40**U***usedamalist()*, 115*usestepval()*, 115**V***value()*, 45**W***whatis()*, 48*whatmac()*, 112**X***ximag()*, 65*xreal()*, 66

