

*SigGen Engine User's Guide –Version 1.0*

**Copyright**

© 1997 TDT. All rights reserved.

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

**Licenses and Trademarks**

Microsoft, MS-DOS, Windows and Windows 95 are registered trademarks of Microsoft Corporation.  
Printed in U.S.A.





# Contents

## Preface

## Organization of the Manual

*SigGen Engine Fundamentals*

*Illustrative Examples*

*Software Reference*

*SigGen User's Guide*

## ***Part 1 SigGen Engine Fundamentals***

---

### ***Chapter 1 Introduction***

***What is SigGen Engine? 1-1***

***SigGen Engine Capabilities 1-1***

SigGen Signal Generation 1-1

SigGen Variables 1-1

SigGen Index 1-1

Peripheral Control 1-1

***Who Can Use SigGen Engine? 1-2***

***How To Use SigGen Engine 1-2***

***Before You Begin 1-2***

What you need 1-2

***Installing SigGen Engine 1-3***

Requirements 1-3

Installation 1-3

SigGen engine includes the following source code files. 1-3

Example Files 1-4

### ***Chapter 2 Programming With SigGen Engine***

***Overview 2-1***

***Basic SigGen Engine Functions 2-2***

Building a SigGen Signal 2-2

Using SigGen Schedules 2-2

**Example 2-1 Playing a SigGen Signal 2-3**

**Advanced Features of SigGen Engine 2-8**

Normalization Files 2-8

Controlling Attenuation 2-8

Triggering with the TG6 2-8

Controlling Peripheral Setups 2-9

Dynamic Variables 2-9

Prompted Variables and Files 2-10

Combination Variables 2-10

**SigGen Data Structures 2-11**

SigType 2-11

SegType 2-11

CmpType 2-11

VarType 2-11

MaskType 2-11

ParType 2-11

---

## **Part 2 Illustrative Examples**

---

### **Chapter 3 Quick Start Examples**

Example 3-1: Playing Two SigGen Signals 3-1

Example 3-2: Temporal Integration 3-1

**Example 3-1 Playing Two SigGen Signals 3-2**

Hardware Setup 3-2

Example Program Listing 3-6

**Example 3-2 Temporal Integration 3-12**

Hardware Setup 3-12

### **Chapter 4 Applications**

**Application 4-1 Single-Unit Neurophysiology 4-1**

Overview 4-1

Features 4-1

**Hardware Configurations 4-2**

Program Operation 4-5

Display 4-6

Data Output 4-7

Trigger Rate 4-8

### ***Programming Notes 4-9***

## ***Part 3 Software Reference***

---

### ***Chapter 5 SigGen Engine Functions***

#### ***File Handling Routines 5-1***

LoadSig ( \*Sig, \*fname ); 5-1

SaveSig ( \*Sig, \*fname ); 5-1

LoadNorm ( \*Sig ); 5-1

#### ***Signal Preparation and Generation 5-2***

InitSig ( \*Sig ); 5-2

CalcSig ( \*TSig ); 5-2

InitSigStuff ( ); 5-2

DoNewSig ( \*Sig ); 5-3

ChangeSigNpts ( \*Sig ); 5-3

CalcSeg ( \*TSig, sn ); 5-3

GetNewSeg ( \*Sig ); 5-3

KillSeg ( \*Sig, sn ); 5-4

GenGate ( \*k, n, factor, rf, srate ); 5-4

CheckBoundry ( \*Sig ); 5-4

FormFName ( sss, \*Sig, \*Cmp ) 5-5

ReadResponse ( \*fname, srate, id1, id2 ); 5-5

#### ***Handling Variables 5-6***

PullVars ( \*Sig ); 5-6

GetVal ( \*Sig, p ); 5-6

ReadVarVal ( \*Sig, p ); 5-7

InitVars ( \*Sig ); 5-7

LoadSched ( \*VT ); 5-7

CalcSteps ( \*V ); 5-7

#### ***Prompted Variables 5-8***

PromptVars1 ( \*Sig ); 5-8

PromptVars2 ( \*Sig ); 5-8

#### ***Controlling Peripherals 5-9***

SetAtten ( \*Sig, attadj ); 5-9

SetMasker ( \*Sig, mn, addadj, sumode ); 5-9

MaskerOff ( \*Sig, mn ); 5-10

SetTG6 ( \*Sig, sn, oodur, reps, os3, dur3, os6, dur6 ); 5-10

**External Callback Functions 5-11**

ErrHand ( s1[], s2[] ); 5-11

AskForVal ( \*v, \*prompt, \*units, defv, minv, maxv ); 5-11

AskForFile ( \*sss, \*prompt ); 5-12

GetDynVar ( \*VT, vn ); 5-12

InitDynVar ( \*VT, vn ); 5-12

CE ( mark[] ); 5-12

**Chapter 6 SigGen Data Structures**

SigType: Signal Structure 6-1

VarType: Variable Structure 6-3

SegType: Segment Structure 6-5

CmpType: Component Structure 6-6

MaskType: Peripheral Structure 6-7

ParType: Parameter Structure 6-8

# Preface

SigGen Engine allows you to harness the power of SigGen in your own C applications. SigGen is the most complete signal design package available today. Many complex auditory stimuli can be generated with ease using SigGen. SigGen Engine is the heart of the SigGen Solutions applications BioSig and PsychoSig.

While BioSig and PsychoSig are designed to be flexible for many applications, there will always be a need for custom applications. SigGen Engine was designed to bridge the need between ease of stimulus design and presentation with the need for custom applications. Using SigGen Engine, you can change the stimulus or stimulus schedule you are using, without reprogramming. Just simply create a new stimulus in SigGen.

Before using SigGen Engine you must be familiar with SigGen. It is also assumed that you are familiar with the AP2, APOS, and the basic operation of TDT System II hardware, as well as the C programming environment.

If you are new to System II, you may find it helpful to begin by familiarizing yourself with System II programming basics. Basic programming information and a variety of examples can be found in the document “Signal Processing Applications” in the System II manual.



# Organization of the Manual

The *SigGen Engine User's Guide* is divided into three parts:

- Part 1      SigGen Engine Fundamentals
- Part 2      Illustrative Examples
- Part 3      Software Reference

## ***SigGen Engine Fundamentals***

Part 1, SigGen Engine Fundamentals presents an overview of SigGen Engine and explains how SigGen signals are generated and stored. Part 1 guides you through the process of loading and generating a SigGen signal file.

## ***Illustrative Examples***

In Part 2, Illustrative Examples, SigGen signals are used in sample programs. These programs show how signals are loaded and played, how peripherals are controlled, and how to take advantage of the SigGen stimulus schedule.

## ***Software Reference***

Part 3, the Software Reference, is an indispensable resource for programming with SigGen Engine. It describes the syntax and use of every SigGen Engine function.

## ***SigGen User's Guide***

You will also find the SigGen manual to be an indispensable aid in using SigGen Engine. If you need help in designing a stimulus, stimulus schedule, or in controlling peripherals, refer to the SigGen User's Guide.



---

---

***Part***  
***1*** | ***SigGen Engine***  
***Fundamentals***



# ***Chapter 1* Introduction**

## ***What is SigGen Engine?***

SigGen Engine is designed to implement the signals and peripheral controls specified in SigGen signal files in your C applications. Simple function calls are provided that allow you to generate SigGen stimuli and to control peripheral setups. SigGen Engine is designed to be used with C compilers.

## ***SigGen Engine Capabilities***

SigGen Engine allows you to access and control the vast amount of information contained in SigGen files. Every feature of a SigGen signal can be used by SigGen Engine.

### **SigGen Signal Generation**

SigGen Engine allows you to recreate the signals saved into a signal file and pop them onto the DAMA with one command. Once your program is written, you won't have to rewrite code to change the stimulus. Simply create a new stimulus using SigGen.

### **SigGen Variables**

SigGen Engine provides full access to all variables defined in the SigGen signal file, including dynamic variables. Variable values are automatically calculated by SigGen Engine.

### **SigGen Index**

SigGen Engine allows simple control of the SigGen Index to allow you to step through the schedule defined in SigGen. SigGen Engine automatically handles the calculation of variables and uses them in stimulus generation and peripheral control.

### **Peripheral Control**

SigGen Engine provides function calls for complete peripheral control, including the waveform generator, programmable filter, cosine switch, programmable attenuator, and timing generator.

## Who Can Use SigGen Engine?

Scientists who need to generate simple to complex auditory stimuli in custom applications. Users include:

- Auditory Scientists
- Neurophysiologists
- Speech Scientists

## How To Use SigGen Engine

1. Design your stimulus using SigGen.
2. Use SigGen Engine to call SigGen files from your C application.

## Before You Begin

### What you need

*See Digital Signal Processing Applications in the System II manual*

*See SigGen User's Guide*

*See System II Manual*

- **Signal Processing**  
A basic knowledge of signal processing is necessary. You should understand the parameters necessary for specification of a signal in the time and frequency domains.
- **SigGen Concepts**  
You should understand how to generate a signal in SigGen, how to use variables, and how the SigGen Index works.
- **C Programming**  
SigGen Engine is provided as C source code. You need to be proficient in C programming to take advantage of the SigGen Engine function calls. You should also be familiar with programming TDT hardware.

# Installing SigGen Engine

## Requirements

In order to use SigGen Engine you must have the following:

- TDT's AP2 Array Processor
- TDT's XBUS hardware (required for playing back SigGen signals)
- SigGen Software and Microsoft Windows to generate SigGen signal files

## Installation

Insert the SigGen Engine disk in your floppy drive and run the install program. Follow the instructions of the installation program.

### From DOS:

Type `a:\install`

### From Windows95:

Double click on *install.exe* in the 'a' drive.

### SigGen engine includes the following source code files.

<i>SigEngin.cpp</i>	SigGen Engine
<i>SigEngin.h</i>	SigGen Engine Header File
<i>SigECall.cpp</i>	SigGen Engine Callback Routines
<i>Fcomplex.obj</i>	Complex filter generation (called by SigGen Engine)
<i>Filt-gen.obj</i>	PF1 filter generation (called by SigGen Engine)
<i>Egavga.bgi</i>	Graphics driver for example programs

**Example Files****Example 2-1 Playing a SigGen Signal**

<i>Exam21.cpp</i>	SigGen Engine Example Program Source Code
<i>Exam21.exe</i>	SigGen Engine Example Program, executable file
<i>Exam21.sig</i>	Example SigGen Signal File

**Example 3-1 Playing Two SigGen Signals**

<i>Exam31.cpp</i>	SigGen Engine Example Program Source Code
<i>Exam31.h</i>	Header file for exam31.cpp (empty)
<i>Plot.c</i>	TDT plot routines used by Exam31.cpp
<i>Plot.h</i>	TDT plot routines header file
<i>Sig1.sig</i>	Example SigGen Signal File
<i>Sig2.sig</i>	Example SigGen Signal File
<i>Exam31.exe</i>	SigGen Engine Example Program, executable file
<i>Egavga.bgi</i>	Graphics driver for example

**Example 3-2 Temporal Integration**

<i>Exam32.cpp</i>	SigGen Engine Example Program Source Code
<i>Exam32.h</i>	Header file for Exam32.cpp (empty)
<i>SigECall.cpp</i>	Modified SigGen Engine Callback Routines to Utilize Dynamic Variables
<i>Plot.c</i>	TDT plot routines used by Exam32.cpp
<i>Plot.h</i>	TDT plot routines header file
<i>Ti.sig</i>	Example SigGen Signal File
<i>Exam32.exe</i>	SigGen Engine Example Program, executable file
<i>Egavga.bgi</i>	Graphics driver for example

**Application 4-1 Single-Unit Neurophysiology**

<i>neuron.cpp</i>	SigGen Engine Application Program Source Code
<i>neuron.h</i>	Header file for neuron.cpp (empty)
<i>ShowSchd.cpp</i>	Function to print out SigGen stimulus schedule
<i>neurplot.c</i>	Modified plot.cpp for handling multiple plot types
<i>neurplot.h</i>	Header file for neurplot.c
<i>neuron.exe</i>	SigGen Engine Application Program Compiled Executable
<i>sig1.sig</i>	Sample SigGen Signal File
<i>sig2.sig</i>	Sample SigGen Signal File
<i>spikes.sig</i>	SigGen Signal File to Simulate Neural Spikes
<i>Egavga.bgi</i>	Graphics driver for example programs

# **Chapter 2 Programming With SigGen Engine**

SigGen Engine makes it possible to design a variety applications written in C. Such applications will include standard APOS and XBDRV function calls and user code designed to implement any special processing you desire. This section explains only the use of SigGen Engine calls. Other calls are described in detail in the APOS and XBDRV Software Reference Manuals.

*Key SigGen Engine functions are explained here.*

Many of the SigGen Engine function calls are made internally within SigGen Engine. While you may make use of all of these function calls, only a subset of the functions are necessary for loading a SigGen file, initializing play, and controlling peripherals.

## **Overview**

SigGen signals are not stored as data, but as the components and segments necessary to generate the signal using standard APOS functions. You must understand the SigGen data structures to understand SigGen Engine. You will find that many of these data structures correspond directly to dialog boxes in SigGen. SigGen Engine provides function calls for loading SigGen signals and constructing the actual signal from the component and segment structures.

## Basic SigGen Engine Functions

### Building a SigGen Signal

To load and use a SigGen signal file requires only three calls to SigGen Engine:

1. `LoadSig(*Sig, *fname)` loads the file *fname* into the SigGen signal structure *Sig*.
2. `InitSig(*Sig)` allocates and zeroes a DAMA buffer for receiving the signal. The DAMA buffer that the signal will be generated in is stored in *Sig->id*.
3. `CalcSig(*Sig)` generates the signal. `CalcSig()` calls `CalcSeg()` to generate the signal segments. `CalcSeg()` uses the data in the `SegType`, `CmpType`, and `VarType` structures to generate the signal. `CalcSig()` combines the segments on the stack and pops the resultant signal into DAMA buffer *Sig->id*.

Now you are ready to play the signal from the DAMA buffer.

### Using SigGen Schedules

*SigGen schedules are illustrated in Example 2-1*

One of the most powerful features of SigGen is the use of schedules and variables. SigGen Engine provides access to these schedules through the *SigType.si* variable, which specifies the current SigGen index (SGI). By changing the SGI and calling `CalcSig()`, you can easily generate the appropriate signal for that SGI. All variable incrementing and calculation is done for you by SigGen Engine. To step through a simple schedule, simply increment the SGI and call `CalcSig()` to generate the new signal for that SGI. When you reach termination control, `CalcSig()` returns the value 2.

Note that the SigGen Index (*SigType.si*) begins at 0, not 1.

## **Example 2-1 Playing a SigGen Signal**

Example 2-1 demonstrates the simplicity and power of SigGen Engine. This example illustrates how to load a SigGen file and use the SigGen index to step through the stimulus schedule. A sample SigGen file is provided for running the program, but feel free to experiment using your own SigGen files.

### **Hardware Requirements:**

D/A module, AP2

### **Compiling Requirements**

This program is supplied as complete C source code as well as a compiled executable. To compile this program yourself you will need to include the following files in your Project:

- exam21.cpp
- apos.c
- xdrv.c
- SigEngin.cpp
- Filt\_gen.c (used by SigGen Engine)

## SigGen Signal File

Use *SigGen* to view *exam21.sig* and its stimulus schedule

This example program includes one sample SigGen file, *exam21.sig*. This file plays a 200 ms tone. The frequency of the tone is determined by the variable *frequency*, which steps from 100 to 10,000 Hz. There are 100 SGIs in this file.

## Example Program

The complete listing of the example program is at the end of this section. Key portions of the program that utilize SigGen Engine function calls have been duplicated here for extended explanation. SigGen Engine variables and functions are shown in **bold**.

### Step 1 Declare SigGen Signal Structures

```
sigType *sig1;
```

Declare SigGen signal structures and variable structures.

### Step 2 Load SigGen Signals

```
if(!LoadSig(sig1, FileIn1))
{
    strcpy(strng, FileIn1);
    ErrHand("Error in opening
    signal file", strng);
}
```

**LoadSig()** loads the SigGen signal file *FileIn1* into the structure *Sig1*. **LoadSig()** sets the signal values to default, and loads the normalization files if they are specified.

### Step 3 Initialize Signals and DAMA Buffers

```
InitSig(sig1);
```

**InitSig()** prepares a zero-value DAMA buffer for the signal. The DAMA buffer id is stored in *Sig1->id*.

### Step 4 Setup D/A

```
DAClear(1);
DAmode(1, DAc1);
DAnpts(1, sig1->npts);
DAstrate(1, sig1->strate);
if(dadev==DA3_CODE)
    DA3setslew(1, AUTOSLEW);
```

This block of code sets up the D/A for playing the SigGen signal. Autodetecting calls allow different D/A modules to be used with this program. The D/A is set to convert the number of points in the signal (*Sig1->npts*), with the sampling rate of the signal (*Sig1->strate*).

If the DA3 is present, the slew rate is set.

## Step 5 Calculate & Play Signal for All SGI's

```

CalcSig(Sig1);
do

{
  play(Sig1->id);
  DAarm(1);
  DAgo(1);
  do{ }while(DAstatus(1)==2);
  gotoxy(10,5);
  printf("SGI %d", Sig1->si);
  Sig1->si++;
  boundaryflag = CalcSig(Sig1);
}while(boundaryflag!=TERMINATION
);

```

CalcSig() generates the signal on the stack based on the current SigGen index (SGI).

The signal is then played out from the DA.

When the signal is finished playing, the current SGI (*sig1->si*) is printed to the screen.

The SGI is then incremented (*sig1->si++*), and the signal recalculated based on the new SGI.

CalcSig() returns 1 when signal calculation is complete, and returns 2 when the SGI has reached TERMINATION.

Loop until reach termination.

### Complete Listing of Example 2-1

```

/*****
SigGen Engine Example Program 2-1

This program:
1. Reads in a SigGen file.
2. Calculates the signal.
3. Steps through SGIs.
4. Plays signal.
*****/
#include <stdlib.h>
#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <dos.h>

extern "C"
{
#include "apos.h"
#include "xbdrv.h"
}

#include "SigEngin.h"

#define TERMINATION 2

void main()
{
char strng[100], FileIn1[100], c;
int boundaryflag; //pflag flag for termination control
SigType *Sig1; //SigGen signal structures

//Initialize AP2
if(!apinit(APa))
if(!apinit(APb))
{
printf("\n\nCan't Initialize AP2!\n\n");
exit(0);
}
if(!XBlinit(USE_DOS))
{
printf("\n\nCan't Initialize XBUS!\n\n");
exit(0);
}

// Get SigGen file names

printf("\n Enter File Name of SigGen Signal: ");
gets(FileIn1);

// Load and Prepare Signals
if(!LoadSig(Sig1, FileIn1)) //Load SigGen signal 1
{
strcpy(strng, FileIn1);
ErrHand("Error in opening signal file", strng);
}

// Prepare AP2 for SigGen signal. DAMA buffer Sig1->id
InitSig(Sig1);

// Program D/A device
// Note: Utilizes autodetecting calls so will work with different D/As

DAclear(1); //Clear DA to default settings
DAmode(1, DAC1); //Select DA for 1 channel play
DAnpts(1, Sig1->npts); //Set number of samples to convert based on SigGen signal 1
DAsrate(1, Sig1->srate); //Set sampling rate to sampling rate of SigGen signal 1
if(dadev==DA3_CODE)
DA3setslew(1, AUTOSLEW);

clrscr();

```

```
printf("\n SigGen Engine Example 2.1: Playing '%s'", Sig1->name);
CalcSig(Sig1); // Calculate SigGen signal for first play
do
{
    play(Sig1->id);           // play SigGen signal from DAMA buffer
    DAarm(1);                // arm DA
    DAgo(1);                 // Start conversion from DA
    do{}while(DAstatus(1)==2); // wait to finish playing

    gotoxy(10,5);
    printf("SGI %d", Sig1->si); // Print current SGI

    Sig1->si++;              // increment SigGen index
    boundaryflag = CalcSig(Sig1); // Calculate SigGen signal

}while(boundaryflag!=TERMINATION); //loop while not at termination

DAClear(1);
}
```

## Advanced Features of SigGen Engine

Now that we have seen the power of SigGen Engine in generating and playing a signal, we'll explore the other powerful features of SigGen Engine. These features are used in the examples in Chapters 3 and 4.

### Normalization Files

SigGen Engine automatically loads and implements normalization files. These files can be used to correct for irregular frequency responses of transducers.

Normalization files are loaded when LoadSig() is called. LoadSig() calls LoadNorm(), which allocates DAMA buffers and loads the normalization files. CalcSig() uses normalization files when generating signals in the frequency domain.

### Controlling Attenuation

*Controlling attenuation using peripheral setups is described under Controlling Peripheral Setups.*

Attenuation can be controlled with a programmable attenuator (PA4).

Attenuation levels can be set in two places in SigGen: in the signal dialog box and in the peripheral setup dialog box.

To set the attenuation in the signal dialog box, use the SigGen Engine call SetAtten(\*Sig, attadj). The level of attenuation for the current SGI will be set on the PA4 specified in the SigGen file. If the attenuation is set by a variable, this will be automatically handled for you by SigGen Engine.

The SetAtten() function call also provides a simple method to calibrate the user's system from within your program by allowing them to specify a certain amount of additional attenuation (*attadj*) to add to the PA4.

### Triggering with the TG6

*See Controlling Peripheral Setups for information on using SetTG6 to control triggering of peripheral setups.*

SigGen Engine provides the function call SetTG6(\*Sig, sn, oodur, reps, os3, dur3, os6, dur6) to create timing sequences for all of the TG6 channels, based on arguments in the function call and variables in the SigGen signal. SigGen signals contain information about triggering the D/A and peripheral setups.

In a typical setup the user may have TG6 channel 2 connected to the D/A trigger, and TG6 channel 3 connected to the A/D trigger. The onset delay for triggering the D/A is contained in the SigGen signal structure and is used automatically by SetTG6. The onset delay and pulse duration for the A/D are passed by the variables os3 and dur3. Set these variables in your program to control signal acquisition.

Remember that you can still create your own triggering setups with standard XBDRV calls. SigGen Engine simply provides a standard setup for triggering and simplified implementation of that setup.

## Controlling Peripheral Setups

*For more information see Example 3-1 and the SetMasker function in Chapter 5*

SigGen allows users to specify two peripheral setups. Each setup can control a group of peripheral devices including a programmable filter (PF1), waveform generator (WG1/2), cosine switch (SW2), and programmable attenuator (PA4). The peripheral setups specified in the SigGen file can be implemented in SigGen Engine by calling `SetMasker(*Sig, mn, addadj, sumode)`. This function call sets up all the peripheral devices defined in the SigGen file. It calculates and loads filters into the PF1, loads the WG1/2 with the specified waveform, loads the SW2 with the ramp shape and rise-fall time, and loads the PA4 with attenuation values.

Since two peripheral setups can be specified in SigGen, the setup that is being implemented is passed in the argument *mn* (note: *mn*=0 is for peripheral setup #1, and *mn*=1 is for peripheral setup #2.)

As with `SetAtten()`, additional attenuation can be added to the PA4 through the variable (*addadj*), which can allow a simple method of calibrating the system.

To create TG6 timing sequences to trigger peripheral setups call `SetTG6()`, which will create the timing setup for peripheral setup #1 on TG6 channel 4, and peripheral setup #2 on TG6 channel 5.

## Dynamic Variables

*For more information see Example 3-2*

You can implement dynamic variables by writing your own `GetDynVar()` function. For example, to use a dynamic variable for sound level that depends on user response to detecting or not detecting a sound, you would add code to `GetDynVar()` that calculated level based on user response.

`GetDynVar()` is found in the SigGen Engine Callback functions (`SigECall.cpp`).

## Prompted Variables and Files

If variables or file names are defined as prompted variables in SigGen, you must call the SigGen Engine functions `PromptVars1()` or `PromptVars2()` to prompt for user input. These functions call `AskForVal()` or `AskForFile()` in `SigECall.cpp`, which prompt for user input.

If you want to modify how variables are prompted to suit your purpose, such as prompting in a graphics mode, you can modify `AskForVal()` and `AskForFile()`. They currently use `PRINTF` commands to issue the prompts.

## Combination Variables

SigGen Engine calculates combination variables when it is constructing a signal with `CalcSig()` and when controlling peripherals. However, the value of `Sig->curval` is the value of the variable *before* the combination operation is performed.

To obtain the value of combination variable after the combination operation is performed, call `GetVal()` or `ReadVarVal()`. If the variable is not a combination variable, the current value of the variable is returned. The difference between these two functions is in the control of alternating variables. `GetVal()` will alternate the variable between the min and max values each time `GetVal()` is called. `ReadVarVal()` will not alternate the value of the variable, and simply returns the current value.

## **SigGen Data Structures**

This section gives an overview of the SigGen data structures that are used in signal generation and peripheral control. These data structures are fully defined in Chapter 6.

### **SigType**

The data structure that you will deal with most often is SigType. SigType contains important variables regarding the SigGen signal such as the DAMA buffer in which the signal is stored, the current SigGen index, the signal file name, attenuation for a programmable attenuator, signal duration, sampling rate, and number of points. SigType also contains structures for segments (SegType) and variables (VarType).

### **SegType**

The structure for SegType should be familiar to you from using the Segment dialog box in SigGen. SegType contains variables used for generating segments such as start time, duration, gate type, and generation method. The SegType structure also contains the structure for components (CmpType).

### **CmpType**

The component structure defines how the component is generated (e.g. tone, sweep, click). It also holds an array of parameter values needed to generate the component.

### **VarType**

The VarType data structure defines variables in the SigGen signal. The VarType structure corresponds to the Variable dialog box in SigGen. VarType includes variables generated by one of the stepping functions (such as linear step or log step) and combination variables.

### **MaskType**

The MaskType structure contained within SigType, implements the peripheral setups specified by the user. MaskType contain variables for control of the timing generator (TG6), programmable filter (PF1), waveform generator (WG1/2), cosine switch (SW2), and programmable attenuators (PA4).

### **ParType**

The ParType structure holds either a variable ID number or the value of the parameter, if it is a constant. This structure allows a signal parameter to assume a constant value or to track the value of a variable.



---

---

**Part**  
**2**

***Illustrative Examples***



# ***Chapter 3* Quick Start Examples**

Quick Start examples illustrate important features and concepts of SigGen Engine. Ready-to-use application files are provided for each Quick Start example.

The examples in this section are listed below with the concepts introduced in each example.

## **Example 3-1: Playing Two SigGen Signals**

- How to load and playback SigGen stimuli
- Step through a SigGen stimulus schedule
- Control triggering
- Control peripherals

## **Example 3-2: Temporal Integration**

- Dynamic variables
- Combination variables
- How to modify SigECall.cpp

## Example 3-1 Playing Two SigGen Signals

In this example, two SigGen signal files are loaded and played out two channels of a D/A module. The following steps develop a C program for using these files. The complete example program and example SigGen files are provided with SigGen Engine.

### Hardware Setup

**Minimum Requirements: D/A, TG6, PA4**

#### Connections

Connect channel 2 of the TG6 to the TRIG of the D/A module. Connect the output of D/A channel 1 to the input of a PA4. Connect the output of the PA4 and D/A channel 2 to the left and right inputs, respectively, of a headphone buffer or oscilloscope to monitor the output.

### SigGen Signal Files

*Investigate the SigGen signals (sig1.sig and sig2.sig) and their stimulus structures and schedules using SigGen.*

This example program includes two sample SigGen files. One is a tone and the other a noise burst. The tone signal (*sig1.sig*) has two variables defined, Freq (frequency) and Level (for controlling PA4 attenuation). The noise burst signal (*sig2.sig*) also has two variables defined, CenterFreq (Center Frequency) and Bandwidth (# of octaves bandwidth). A stimulus schedule has been set up that steps through the different levels at each frequency for the tone. This is done once for each bandwidth of the noise signal, which is set up in the noise signal stimulus schedule. The noise signal center frequency is set in the stimulus schedule to be the same as the frequency of the noise signal.

### Example Program

The complete listing of the example program is at the end of this section. Key portions of the program that utilize SigGen Engine function calls have been duplicated here for extended explanation. It is important to familiarize yourself with the SigGen signal structure (SigType), to have full access to the information contained in the SigGen signal files. SigGen Engine variables and functions are shown in **bold**.

#### Step 1 Declare SigGen Signal Structures

```
SigType   *Sig1, *Sig2;
VarType  *VT;
```

Initialize SigGen signal structures and variable structures. Sig1 is the structure for SigGen signal 1.

## Step 2 Load SigGen Signals

```
if(!LoadSig(Sig1,FileIn1))
{strcpy(strng, FileIn1);
 ErrHand("Error in opening
 signal file", strng);}
if(!LoadSig(Sig2,FileIn2))
{strcpy(strng, FileIn2);
 ErrHand("Error in opening
 signal file", strng);}
```

Call the SigGen Engine function LoadSig(). This loads the SigGen signal file *FileIn1* into the structure *Sig1*. LoadSig() sets the variable values to their default values, and loads the normalization files if they are specified.

## Step 3 Initialize Signals and DAMA Buffers

```
InitSig(Sig1);
InitSig(Sig2);
InitSigStuff();
```

InitSig() allocates a stack and generates a zero signal on the stack. A DAMA buffer for the signal is allocated and the DAMA buffer id is stored in *Sig1->id*. The stack is popped to the DAMA.

InitSigStuff() initializes the prompt strings for SigGen signal components.

```
SetMasker(Sig1, 0, 0, 1);
PromptVars1(Sig1);
```

Initialize peripheral setup #1, for *Sig1*.

Ask user for values of any prompted variables in *Sig1*.

## Step 4 Setup D/A

```
DAClear(1);
DAmode(1, DUALDAC);
DAnpts(1, Sig1->npts);
DAsrate(1, Sig1->srate);
if(dadev==DA3_CODE)
DA3setslew(1, AUTOSLEW);
DAmtrig(1);
DAarm(1);
```

This block of code sets up the D/A for playing the SigGen signal. Autodetecting calls allow different D/A modules to be used with this program. Select D/A for two-channel play.

The D/A is set to convert the number of points in the signal (*Sig1->npts*), with the sampling rate of the signal (*Sig1->srate*).

If the DA3 is used, the slew rate is set.

Set DA to play on repeated external triggering.

## Step 5 Setup Triggering

```
TG6dur = 4 * (Sig1->dur);
SetTG6(Sig1, 0, TG6dur, 0, 0,
0.0, 0, 0.0);
```

Set TG6 duration to 4 times SigGen signal 1 duration. This SigGen Engine call initializes the TG6 for a timing sequence from channel 2 for triggering the D/A. The onset delay for the trigger (although set to 0 in the SigGen file sig1.sig) is passed in the pointer to Sig1, and is used in setting the timing of the trigger pulse. No timing sequence is set for channels 3 and 6 by passing duration values of 0.0.

## Step 6 Calculate Signal and Plot

```
pflag1 = CalcSig(Sig1);

Sig1->si++;
dpush(Sig1->npts);
scale(1/3276.8);
tdtplot1(p1); tdtplot2(p1, 2,
START);
dropall();
pflag2 = CalcSig(Sig2);
Sig2->si++;
dpush(Sig2->npts);
scale(1/3276.8);
tdtplot1(p2);
tdtplot2(p2, 2, START);
dropall();
```

CalcSig() generates the signal on the stack based on the current SigGen index (SGI). The signal is then scaled to Volts and popped to the DAMA (*Sig1->id*). CalcSig() returns 1 when signal calculation is complete, and returns 2 when the SGI has reached termination.

Increment SGI for *Sig1*.

Because CalcSig() pops the signal to the DAMA, it is not available on the stack for plotting. However, the signal is still there, and can be “recovered” by reallocating the memory of the signal with the dpush command. Scale the signal and plot it, before clearing the stack.

Calculate Sig2.

## Step 7 Set Attenuation and Peripherals, Play, and Trigger

```
if((pflag1!=2) && (pflag2!=2))
{
    SetAtten(Sig1, 0.0);

    SetMasker(Sig1, 0, 0, 0);

    dplay(Sig1->id,Sig2->id);
    TG6arm(1,0);
    TG6go(1);
}
```

*pflag* = 2 when termination control of the SGIs has been reached. Will only play signals if neither has reached a termination boundary.

SetAtten is a SigGen Engine command that sets the attenuation specified on the PA4. The value for attenuation and the PA4 device number are passed in the pointer to *Sig1*. Additional attenuation is set to 0.0 here, but could be used to calibrate the system. This value is added to the attenuation specified in the SigGen signal.

SetMasker updates current settings from *Sig1* Peripheral Setup #1. 0 dB additional attenuation is added to the PA4.

The APOS dplay() function plays dual DAMA buffers containing SigGen signals.

TG6 APOS commands arm and start conversion of TG6 trigger sequence. The TG6 will then trigger the D/A and play the SigGen signals.

## Step 8 Display Variable Names and Values

```

void ShowSGI(SigType
*Sig1,SigType *Sig2)
{
    int j;
    VarType *VT;
    printf("Signal 1 Variables");
    printf("Signal 2 Variables");
    for (j=0; j<MAXVARS; j++)
    {
        VT=&Sig1->Var[j];
        if (VT->name[0]!='.')
            printf("%-10s=%-.3f",
                (VT->name), (VT->curval));
        VT=&Sig2->Var[j];

        if (VT->name[0]!='.')
            printf("%-10s=%-.3f",
                (VT->name), (VT->curval));
    }
}

```

This section of code prints the variable names and their current values, from the SigGen signal structures.

The names are stored in *Sig1->Var[j]->name*.

The values are stored in *Sig1->Var[j]>curval*.

## Example Program Listing

The following is a complete listing of Example 3-1. This example program is provided with SigGen Engine as a compiled executable file (Exam31.exe) and as source code (Exam31.cpp).

```

/*****
SigGen Engine Example Program

This program:
1. Reads in a SigGen file.
2. Calculates and plots the signal.
3. Plays the signal by triggering with TG6.
4. Steps through SGIs
5. Controls PA4 and other peripherals.
6. Displays values and names of all variables.

*****/
#include <stdlib.h>
#include <math.h>
#include <conio.h>
#include <stdio.h>
#include <string.h>
#include <graphics.h>
#include <dos.h>
#include <ctype.h>

extern "C"
{
    #include "apos.h"
    #include "xbdrv.h"
    #include "plot.h"
}

#include "SigEngin.h"
#include "Exam31.h"

//Definitions for rflag
#define EXITPGM    0
#define ZEROSGI   2
#define SKIPSGI   3
#define BACKSGI   3
#define PLAYSGI   1

void ShowSGI(SigType *Sig1,SigType *Sig2);

void main()
{
    // Regular variables
    char    strng[100], FileIn1[100], FileIn2[100], c;
    int     i, j, pflag1, pflag2; //pflag flags for termination control
    int     rflag; // program control flag
    float   TG6dur;

    FILE    *ftmp;
    plotinfo *pl, *p2, *p3;

    SigType *Sig1, *Sig2; //SigGen signal structures
    VarType *VT; //SigGen variable structure

    if (( Sig1 = (SigType *)malloc(sizeof(SigType)))==NULL)
    {
        printf("Not enough memory to allocate Sig1 \n");
        exit(1);
    }

    if (( Sig2 = (SigType *)malloc(sizeof(SigType)))==NULL)
    {

```

```

    printf("Not enough memory to allocate Sig2 \n");
    exit(1);
}
if (( p1 = (plotinfo *)malloc(sizeof(plotinfo))!=NULL) //plot 1
    {
    printf("Not enough memory to allocate p1 \n");
    exit(1);
}
if (( p2 = (plotinfo *)malloc(sizeof(plotinfo))!=NULL) //plot 2
    {
    printf("Not enough memory to allocate p2 \n");
    exit(1);
}

//Initialize AP2
if(!apinit(APa))
if(!apinit(APb))
{
    printf("\n\nCan't Initialize AP2!\n\n");
    exit(0);
}
if(!XBlnit(USE_DOS))
{
    printf("\n\nCan't Initialize XBUS!\n\n");
    exit(0);
}

if((ftmp = fopen("EGAVGA.BGI", "rb"))!=NULL)
{
    ErrHand("EGAVGA.BGI file not found.",
            " You must have this file in current directory to run this program.");
}
fclose(ftmp);

//Check for TG6, DA and AD
if(XBlnit(TG6_CODE, 1) && XBlnit(DA1_CODE, 1) && XBlnit(DA3_CODE, 1))
{
    ErrHand("Device not found.",
            " You must have a TG6 and a D/A to run this program.");
}

// Get SigGen file names
gotoxy(20, 5);
printf("Enter File Name for SigGen Signal 1: ");
gets(FileIn1);

gotoxy(20, 6);
printf("Enter File Name for SigGen Signal 2: ");
gets(FileIn2);

// Load and Prepare Signals
if(!LoadSig(Sig1, FileIn1)) //Load SigGen signal 1
{
    strcpy(strng, FileIn1);
    ErrHand("Error in opening signal file", strng);
}
if(!LoadSig(Sig2, FileIn2)) //Load SigGen signal 2
{
    strcpy(strng, FileIn2);

    ErrHand("Error in opening signal file", strng);
}

// Prepare AP2 for SigGen signal. Allocates 0 value stack.
// and DAMA buffers the length of the signal.
// DAMA buffers Sig1->id & Sig2->id
InitSig(Sig1);
InitSig(Sig2);
InitSigStuff(); //Initialize prompt strings for SigGen components

//Initialize peripheral setup using Sig1 peripheral setup #1
SetMasker(Sig1, 0, 0, 1);

```

### 3-8 Example 3-1

```
PromptVars1(Sig1); //prompt for variables if used in Sig1
// Program D/A device
// Note: Utilizes autodetecting calls so will work with different
// DA modules
DAclear(1); //Clear DA to default settings
DAmode(1, DUALDAC); //Select DA for 2 channel play
DAnpts(1, Sig1->npts); //Set number of samples to convert based on SigGen signal 1
DAsrate(1, Sig1->srate); //Set sampling rate to sampling rate of SigGen signal 1
if(dadev==DA3_CODE)
    DA3setslew(1, AUTOSLEW);
DAmtrig(1); //Set up DA for multiple triggering
DAarm(1); //arm DA

// Initialize plot info
initplotinfo(p1); //calls plot.c to initialize plot parameters
initplotinfo(p2);
    p2->xx1 = 360;
    p2->yy1 = 40;
    p2->xx2 = 560;
    p2->yy2 = 180;
    p2->xmax = Sig2->dur;
    p1->xmax = Sig1->dur;
gon(); //calls plot.c to turn graphics on

// Program TG6, will use channel 2 to trigger D/A
// Set TG6 duration to 4 times SigGen signal 1 dur
TG6dur = 4 * (Sig1->dur);
// Set TG6 triggering of D/A through channel2 (this would include
// any onset delay that is specified in the SigGen signal.
SetTG6(Sig1, 0, TG6dur, 0, 0, 0.0, 0, 0.0);

do // Initialize play parameters; also loop for restarting at SGI 0
{

    pflag1 = 0; pflag2 = 0; //reset termination control flags
    Sig1->si=0; //set SGI to zero
    Sig2->si=0; //set SGI to zero

    do
    {
        rflag=PLAYSGI; // initialize rflag in case skip SGI

        // Calculate standard and target signals

        pflag1 = CalcSig(Sig1); // Calculate SigGen signal 1
                                // pflag1 = 2 when termination reached
        Sig1->si++; // increment SigGen index
        dpush(Sig1->npts); // push to recover signal on stack for plot
        scale(1/3276.8);
        tdtplot1(p1); //plot signal
        tdtplot2(p1, 2, START);
        dropall(); //clear stack

        pflag2 = CalcSig(Sig2); // Calculate SigGen signal 2
                                // pflag2 = 2 when termination reached
        Sig2->si++; // increment SigGen index
        dpush(Sig2->npts); // push to recover signal on stack for plot
        scale(1/3276.8);
        tdtplot1(p2);
        tdtplot2(p2, 2, START);
        dropall();

        if((pflag1!=2) && (pflag2!=2)) //if have not reached Term. Control
        {
            SetAtten(Sig1, 0.0); // set attenuation on PA-4 specified in
                                // SigGen signal 1

            SetMasker(Sig1, 0, 0, 0); // Update peripheral setup #1, using values in Sig1
                                // 0 dB additional attenuation on PA4
            dplay(Sig1->id,Sig2->id); // play SigGen signals from DAMA buffers

            TG6arm(1,0); //arm TG6
        }
    }
}
```

```

    TG6go(1);                //start TG6 conversion
    ShowSGI(Sig1, Sig2);     // Display SGI index and signal variables
    gotoxy(1, 1);
    printf("SGI: %4d          'b' Back 's' Skip 'r' Restart
'x'eXit", Sig1->si-1);

    do{
        if(kbhit())
        {
            c=getch();
            if(c=='x' || c=='X') //eXit
                rflag = EXITPGM;
            if(c=='r' || c=='R') //Restart at SGI 0
                rflag = ZEROSGI;
            if(c=='s' || c=='S') //Skip this SGI
                rflag = SKIPSGI;
            if(c=='b' || c=='B') //Back one SGI
            {
                Sig1->si=Sig1->si-2; //SGI already incremented
                Sig2->si=Sig2->si-2; //so subtract 2
                if (Sig1->si<0)
                    Sig1->si=0;
                if (Sig2->si<0)
                    Sig2->si=0;
                rflag=BACKSGI;
            }
        }
        }while(rflag==PLAYSGI); //loop until change SGI
    TG6stop(1); //stop triggering
    do{while(DAstatus(1)==2); //wait to finish conversion
    }
    }while((pflag1!=2) && (pflag2!=2) && ((rflag==PLAYSGI) || (rflag==SKIPSGI) ||
(rflag==BACKSGI))); //loop while not at termination or restart or quit
    TG6stop(1);
    }while(rflag==ZEROSGI); //Loop back to beginning if repeat
    goff();
    DAclear(1);
} /* === end of main === */

void ShowSGI(SigType *Sig1,SigType *Sig2) //Print SGI variables and values to screen
{
    int j;
    VarType *VT;
    gotoxy(1,20);
    printf("Signal 1 Variables");
    gotoxy(40,20);
    printf("Signal 2 Variables");
    for (j=0; j<MAXVARS/2; j++)
    {
        VT=&Sig1->Var[j];
        gotoxy(1,21+j);
        if (VT->name[0]!='.') //default no Variable '.....'

        //print variable name and current value
        printf("%-10s=%.3f", (VT->name), (VT->curval));
        VT=&Sig2->Var[j];
        gotoxy(40,21+j);
        if (VT->name[0]!='.')
            printf("%-10s=%.3f", (VT->name), (VT->curval));
    }
    for (j=MAXVARS/2; j<MAXVARS; j++)
    {
        VT=&Sig1->Var[j];
        gotoxy(20,16+j);
        if (VT->name[0]!='.')
            printf("%10s=%.3f", (VT->name), (VT->curval));
        VT=&Sig2->Var[j];
        gotoxy(60,16+j);
        if (VT->name[0]!='.')
            printf("%10s=%.3f", (VT->name), (VT->curval));
    }
}

```



## **Example 3-2 Temporal Integration**

In this example a four-interval forced choice experiment is developed in which a sound is presented monaurally in one of four intervals. The sound level is varied with *Level* as a dynamic variable. The sound is presented in one of four intervals, and the subject responds to which of the intervals he heard the sound. When the subject's response is correct the sound level is decreased 4 dB by increasing the attenuation on a programmable attenuator (PA4). *Level* is defined in the SigGen file as a combination variable.

The SigGen schedule is used to obtain a threshold at each of four different signal durations.

This example program is structurally the same as Example 3-1. Important changes to the program are explained in detail below. The entire source code, a sample SigGen file, and a compiled executable are provided on the installation disks.

### **Hardware Setup**

**Minimum Requirements: D/A, TG6, PA4**

#### **Connections**

Connect channel 2 of the TG6 to the TRIG of the D/A module. Connect the output of D/A channel 1 to the input of a PA4. Connect the output of the PA4 to the left input of a headphone buffer or oscilloscope to monitor the output.

*Investigate the SigGen signal (TI.sig) and its variables and schedule using SigGen.*

```
extern "C"
{
    #include "apos.h"
    #include "xbdrv.h"
    #include "plot.h"
}
#include "SigEngin.h"
#include "Exam31.h"
```

## **Step 1** Declare Global Variables

```
int stepvalu, ntrials;
```

## SigGen Signal Files

This example program includes one sample SigGen file. Two variables are defined: *Duration* and *Level*. *Duration* is calculated using LogStep (base 10) and has its termination specified by Boundary Control. *Level* is defined as a Dynamic Variable, with a default level of 70 dB. It is also defined as a combination variable, as 99.9 dB – *Level*. This value is used to set the attenuation on the PA4 (in the Signal Parameters dialog box, PA4-1 is set to *Level*).

## Example Program

Key portions of the program that utilize SigGen Engine function calls have been duplicated here for extended explanation. SigGen Engine variables and functions are shown in **bold**.

## Include Files

These files must be included in your project to compile this program.

## **Steps 2-5** Load SigGen Signals, Initialize Buffers, Setup D/A, Setup Triggering

Same as Example 3-1, except only one signal is used.

## Step 6 Set Attenuation and Peripherals, Play, and Trigger

```

if((pflag1!=2))
{
  SetMasker(Sig1, 0, MaskerCal,
0);
  play(Sig1->id);
  DAarm(1);
//some code not shown here//

  SetAtten(Sig1, 0.0);

  for (i=1; i<5; i++)
  {
    playtrial[i]=0;
  }
  playn = random(4)+1;
  playtrial[playn]=1;
  for (i=1; i<5; i++)
  {
    delay(100);
    if (playtrial[i])
    {
      TG6arm(1,0);
      TG6go(1);
    }
  }
  delay(Sig1->dur);

```

*pflag* = 2 when termination control of the SGIs has been reached. Will only play signals if neither has reached a termination boundary.

SetMasker will update current settings from *Sig1* Peripheral Setup #1. *MaskerCal* dB additional attenuation is added to the PA4 to illustrate how you could calibrate the system.

SetAtten is a SigGen Engine command that sets the attenuation specified on the PA4. The value for attenuation and the PA4 device number are passed in the pointer to *Sig1*. Additional attenuation is set to 0.0 here, but could be used to calibrate the system. This value is added to the attenuation specified in the SigGen signal.

delay(100); sets inter-stimulus interval.

*playn* is used to select the random trial on which to play the sound, by arming and triggering the TG6. The TG6 will trigger the D/A and play the SigGen signals. The program waits the duration of the signal before continuing.

## Step 7 Get User Response & Set Dynamic Variable

```

c=getch();
//some code not shown//
if(playn==atoi(&c))
{
  stepvalu = -1 * stepsize;
  setcolor(RED);
  sprintf(strng, "CORRECT");
  outtextxy(x, 10*i, strng);
}
else
{
  stepvalu = stepsize;
  setcolor(RED);
  sprintf(strng, "INCORRECT");
  outtextxy(x, 10*i, strng);
}
}
delay(200);

```

If the subject responds correctly, *stepvalu* is set to  $-stepsize$ . *stepsize* is set to 4 dB in this example. On the next trial *level* will be lowered 4 dB (and attenuation increased 4 dB).

Likewise, if the subject responds incorrectly, *stepvalu* is set to *stepsize*. This *stepsize* could be assigned a different value depending on what trial was being run, so that the first few trials had larger increments or decrements in level.

Delay for inter-trial interval.

## Step 8 Update Dynamic Variables

```

j=ShowSGI(Sig1);

//Code Fragment From
ShowSGI/////

if (VT->source==DYNAMIC)
    i = j;
    }
    return(i);
//End of code fragment//

VT=&Sig1->Var[j];
level = VT->curval;
makedamaf(threshbuf, trials,
level);
qpushf(threshbuf);
block(0, trials); reduce();
tdtplot1(p2);
tdtplot3(p2, 3, START);
dropall();

trials++;
if (trials>ntrials)
    rflag = 3;
pflag1 = CalcSig(Sig1);
}while(rflag==1);

InitVars(Sig1);
SGI++;
if (SGI<0)
    SGI=0;
Sig1->si=SGI;
}

```

*j* is the variable ID for the dynamic variable. This is determined in the ShowSGI function by looking at the value of VT->source, and is used to obtain the current value of level. Note, this is the value before the combination operation is performed, and thus represents the true level, not the attenuation applied to the PA4.

The DAMA buffer *threshbuf* holds the threshold data and for plotting.

The signal is recalculated using CalcSig() to update the dynamic variables.

InitVars() reinitializes the variables for the next SGI. Since, InitVars() also sets Sig->si=0, the program must restore the correct SGI.

## Step 9 Modify SigECall.cpp

```

float GetDynVar(VarType *VT, int
vn)
{
    extern int stepvalu;
    VT->curval=VT->curval+
        (stepvalu);
    return(VT->curval);
}

```

GetDynVar() in SigECall.cpp is modified to use the variable *stepvalu* in calculating the current level. GetDynVar() is called when CalcSig() is called in the main program. It is not called directly.

# Chapter 4 Applications

This chapter demonstrates how SigGen Engine can be used to create custom applications and to provide working code for getting started with typical applications.

## Application 4-1 Single-Unit Neurophysiology

### Overview

This Single-Unit Neurophysiology program demonstrates the versatility of SigGen Engine in generating powerful applications. This program is fully functional: it will play one or two SigGen stimuli, measure spike times with an Event Timer (ET1), display neural recordings and stimulus recordings, and save the data to files for further analysis.

The program is supplied as C source code and as a compiled executable. The files include:

*neuron.cpp* Source code  
*neurplot.cpp* modified plot.cpp for different types of plots  
*neuron.exe* Compiled executable  
*sig1.sig* Example SigGen signal file  
*sig2.sig* Example SigGen signal file  
*spikes.sig* SigGen signal file to simulate neural spikes

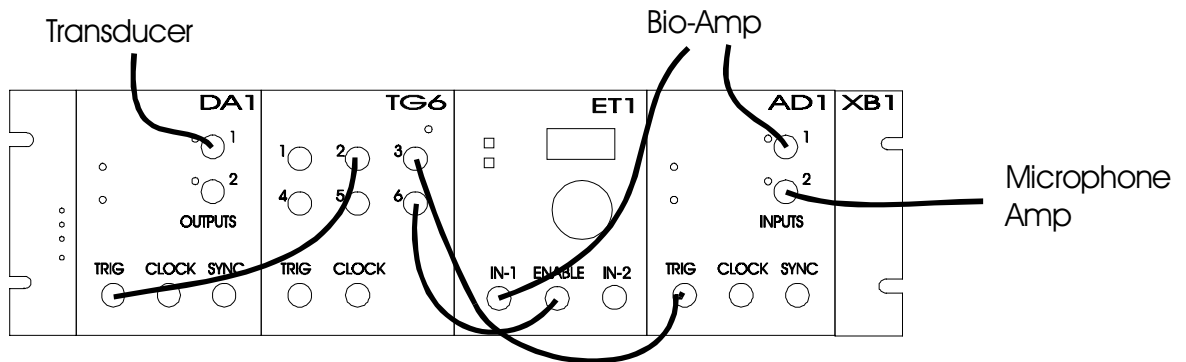
### Features

- Play 1- or 2-channel SigGen signals.
- Record and save stimuli, neural spike trains, spike times.
- Calculate and save Peri-Stimulus Time Histograms (PSTH) and Inter-Spike Interval Histograms (ISIH).
- Display spike raster plots, spike train, and stimulus.
- Overlay of detected spikes on spike train.
- Display PSTH, ISIH, and rate-activity plots.
- Separate SEARCH and RECORD modes.
- Control over number of presentations of each SigGen signal.
- Display and control over SigGen stimulus schedule.
- Control over peripherals specified in SigGen signal.

## Hardware Configurations

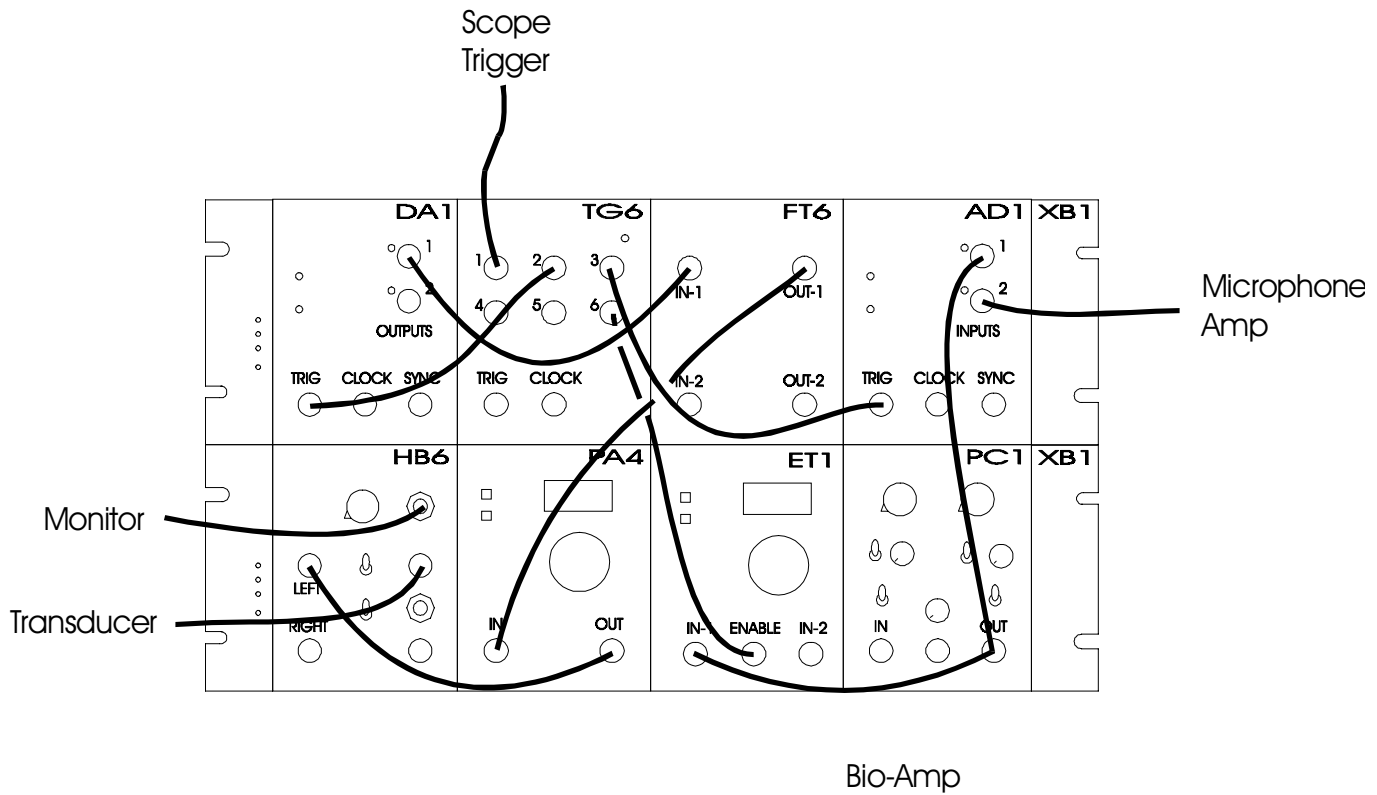
### Minimum System

The minimum setup that is needed to run this program includes a D/A module, an A/D module, a TG6 and an ET1. The basic connections between the Trigger Generator (TG6) and the DA1, AD1, and ET1 are shown here. These connections are the same for all the configurations shown here. A pre-amplifier (Bio-Amp) amplifies the neural recording, which is fed to the ET1 and channel 1 of the AD1. A microphone amplifier amplifies the signal as it is recorded at the subject.



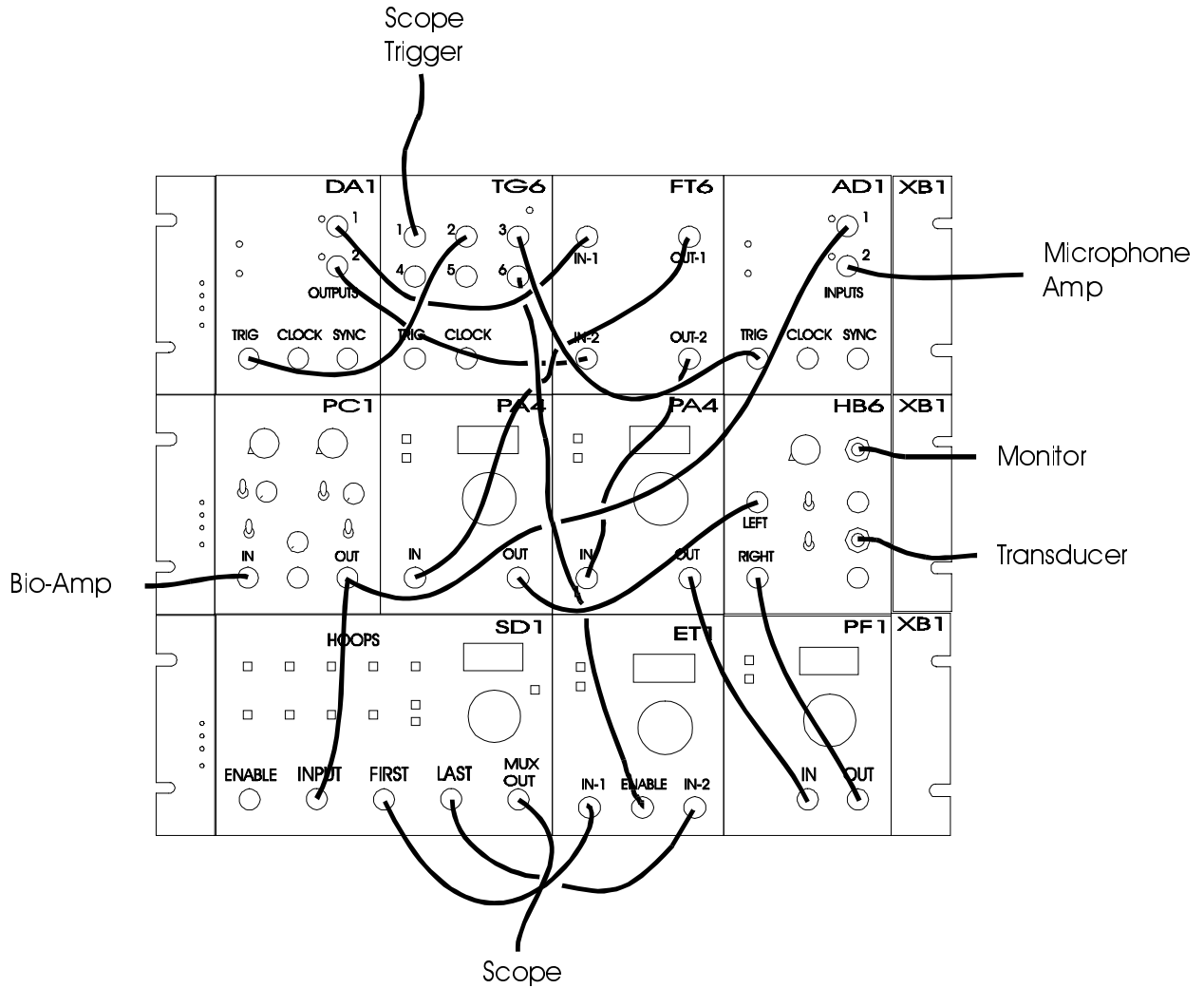
### Basic System

This is a more appropriate setup for gathering neurophysiological data. An anti-aliasing filter (FT6) filters the output of the DA1. A programmable attenuator (PC1) attenuates the signal produced by the DA1. A spike preconditioner (PC1) amplifies and filters the neural signal from the pre-amplifier (Bio-Amp). Adjust the gain on the PC1 to set the level of the spikes to an appropriate level for the ET1. The HB6 drives a transducer.



### Spike Discrimination System

This system is the same as the Basic system, but incorporates a Spike Discriminator (SD1). The SD1 will allow discrimination of different spike shapes from a neuronal signal. A second PA4 is shown attenuating the second channel from the DA1. A PF1, which can be controlled from the SigGen signal, filters the right channel of sound.



## Program Operation

Signal presentation and neural recording are controlled by triggering from the TG6. Channel 2 of the TG6 triggers the DA, and channel 3 triggers the AD. TG6 channel 6 ENABLEs spike detection by the ET1. The neural recording will be made at the same sampling rate and duration as the SigGen signal.

### SigGen Signals

When you first run the program you will be prompted for SigGen file names. One or two SigGen signals can be used as stimuli. To use only one SigGen signal, press ENTER when prompted for the second SigGen signal. The SigGen variables in the *first* SigGen signal will be used to step through the SigGen stimulus schedule and to control peripherals.

Two sample SigGen signal files are included with the example (*sig1.sig* and *sig2.sig*). Another SigGen signal file, *spikes.sig*, is included that can be used to simulate neural spikes by using it as one of the SigGen files, and directing the output of the DA to the ET1.

### Modes

There are three modes of operation. The program starts in the SEARCH mode.

**SEARCH**- Plays stimuli and displays neural spikes and recording, but does not save data.

**PAUSE**- Pause mode is the same as the Search mode, but no stimuli are played. Spike data are displayed, but not saved.

**RECORD**- In the record mode each SigGen index will be played a number of times specified by the user (see *Changing the Number of Trials*), and then advance to the next SGI. Data will be saved only in the Record mode. When you are recording data, RECORD will be displayed in red.

'S' toggles between Search and Record modes.

'P' toggles between Pause and Record modes.

### Controlling the SigGen Index

The SigGen Index can be incremented or decremented by pressing A (Advance) or B (Back). This can be done in any mode.

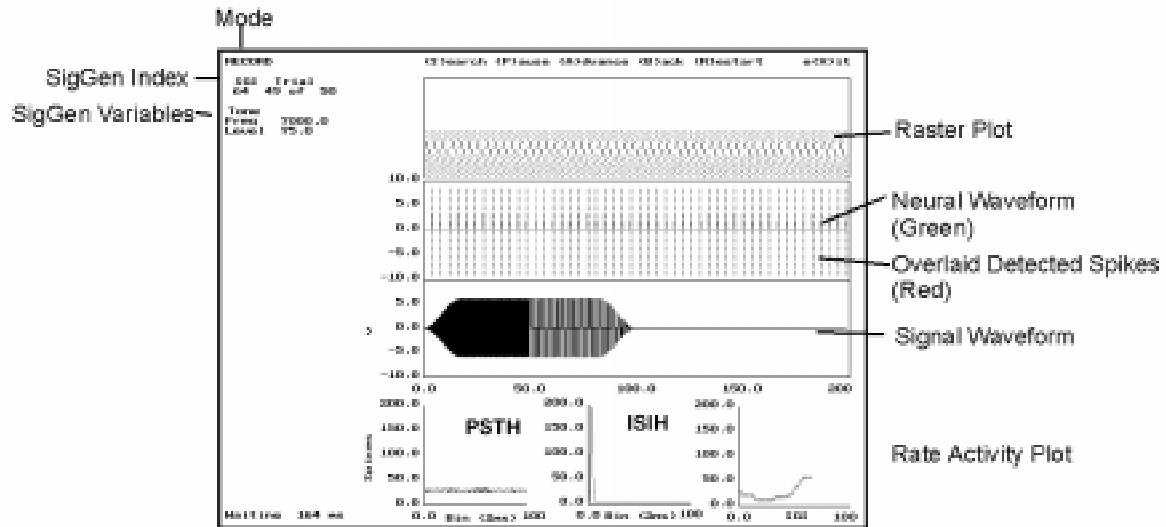
### Displaying the SigGen Schedule

The SigGen stimulus schedule can be displayed by pressing 'L'. To return to the main screen, press 'X'.

### Changing the Number of Trials

Press 'T' when in Search or Pause mode, and then enter the number of trials.

## Display



### Mode

The current mode is displayed in the upper left corner of the screen.

### SGI and Variables

The current SigGen Index and trial are displayed in the upper left along with the current SigGen variable values.

### Raster Plot

Spike times are plotted in the form of a raster plot, in which each dot represents a spike. These data are saved to a file after all trials have been run for a SigGen index, or when the program is exited.

### Neural Waveform

The neural waveform from A/D channel 1 is plotted below the raster plot. Spikes detected by the ET1 are indicated by a red dotted line superimposed on the neural waveform. This will allow you to adjust the level of your spike signal to achieve the desired spike detection.

### Signal Waveform

The signal waveform from A/D channel 2 can be used to record the signal at the subject or directly from the D/A output.

### Peri-Stimulus Time Histogram (PSTH)

A cumulative PSTH is generated for each SGI and plotted below the signal waveform. The bin width of the PSTH can be modified by pressing 'N' from SEARCH or PAUSE modes. This PSTH is optionally saved to an ASCII file.

**Inter-Spike Interval Histogram (ISIH)**

A cumulative ISIH is also calculated for each SGI and plotted next to the PSTH. The ISIH bin width can be specified separately from the PSTH bin width by pressing 'N'. The ISIH data can be saved to a separate file or saved to the same file as the PSTH data (see below *Data Output*).

**Rate Activity Plot**

A rate activity plot is generated and plotted next to the PSTH plot. The rate activity data is the average number of spikes for all trials at each SGI. This data is not saved to a file, but can easily be calculated from the raw ET-1 data.

**Data Output****Spike Data, PSTH, ISIH**

The *Neuron* program always saves the spike data from the ET1 to a file. Data for PST and ISI histograms can be optionally saved in other files. These files can be easily imported into spreadsheets for further analysis and plotting. To change the path or filenames for these files press 'F' in the Search or Pause modes, and then enter new filenames. If the file already exists, the data will be appended to the end of the file with a new header line. To save the PST and ISI data to the same file, give them the same filename. If you do not wish to save these files, press ENTER when prompted for filenames.

**Waveforms**

The neural waveforms and signal waveforms can be saved in either ASCII or binary format. By default these files are not saved. To save them, press 'F' from the Search or Pause mode. Neural and signal waveforms are saved to separate files. Enter a prefix of five or fewer letters to be used for the filename (note: use different prefixes for neural and signal waveforms). If you do not wish to save these files, press ENTER when prompted for a prefix.

When neural waveforms and/or signal waveforms are saved, each waveform is saved to a separate file. The file name convention is as follows:

(file prefix)(SGI).(trial).

Thus, if you had a prefix of 'SubA' for a neural waveform and had two trials for each of three SigGen Indices, the files that would be generated would be:

SubA0 . 0	(SGI 0, Trial 0)
SubA0 . 1	(SGI 0, Trial 1)
SubA1 . 0	(SGI 1, Trial 0)
SubA1 . 1	(SGI 1, Trial 1)
SubA2 . 0	(SGI 2, Trial 0)
SubA2 . 1	(SGI 2, Trial 1)

### ASCII Files

Saving ASCII files will limit how fast you can present signals (see *Trigger Rate* below).

No header information is saved to ASCII files.

ASCII file data are in units of Volts for easy analysis and plotting.

You must know the sample rate at which the signals were sampled to determine the appropriate time axis on which to plot these files. By default, the same sampling rate that was used in the SigGen signal file is used in for sampling the neural and signal waveforms.

### Binary Files

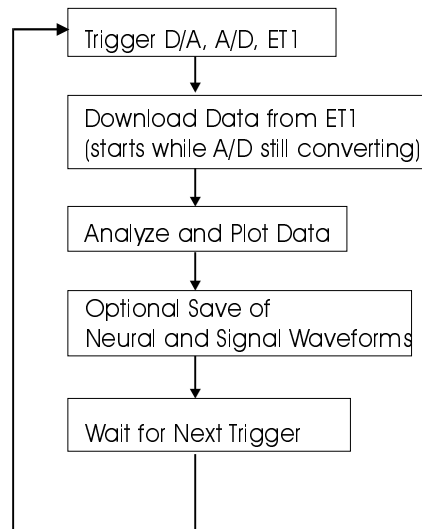
Binary files take much less disk space than ASCII files. Thus, binary files can be saved much more quickly than ASCII files. However, binary files are more difficult to read into spreadsheets. You could write an analysis program that uses standard APOS functions to read in and analyze the binary files. No header information is saved to the binary files.

The binary data are not scaled to Volts (to scale them to Volts multiply by (1/3276.8)).

## Trigger Rate

The TG6 triggers stimulus presentation, stimulus recording, and ET-1 timing.

The trigger period (1/trigger rate) must be longer than the stimulus period. The default trigger period is set to five times the stimulus duration. To set the trigger rate, press 'G' in Search or Pause modes, and enter a new trigger rate (in msec). The program flows as follows:



It takes time to download data from the ET1 and generate the plots on the screen. Thus, you will lose data if the trigger rate is too fast, such that the next trigger goes out before all of the data has been downloaded, processed, saved and plotted. If this happens, a warning will be displayed on the screen and the program will revert to SEARCH mode if you were in RECORD mode.

The program also displays an approximate amount of time the computer waits before the next trigger is received (accurate to only about 50 ms). This can be used to determine the most optimal trigger rate. The fastest trigger rate possible will vary depending on how many spikes are detected and whether the neural and signal waveforms are being saved.

## Programming Notes

While the structure of Application 4-1 is similar to that of the previous examples, there are some differences that involve SigGen Engine. These differences will be explained below in order to help you with your own modifications to this program and in writing your own programs.

SigGen Engine variables and functions are shown in **bold**.

### TG6 Triggering

```
SetTG6(Sig1, 0, TG6dur, ntrials,
0, 1.0, 0, neuraldur);
```

#### Multiple Triggering

To provide precise timing of signal presentations, the TG6 is used with multiple triggering, with the number of repetitions set to the number of trials (*ntrials*) for each SGI.

### ET1 Triggering

Channel 6 enables ET1 event timing. This is set by adding *neuraldur* to the SetTG6 command. The ET1 thus records data over the same period as the signal is played.

```
TG6new(1, 1, TG6dur, 0);
TG6high(1, 1, 0, 1.0, 0x4);
//chan 3
TG6high(1, 1, 0, neuraldur,
0x20); //chan 6
```

### SetTG6()

While SigGen Engine provides SetTG6() to ease the configuration of triggering setups, it may not cover all situations. In this program, a separate triggering sequence (sequence 1) was created for the PAUSE mode in which neural signals are recorded, but no sound is played. This was done by triggering channels 3 and 6, but not channel 2 (D/A).

### SigGen Stimulus Schedule

```
for (i=0; i<45; i++)
{
    Sig->si=SGI;
    PullVars(Sig);
//code not shown here
}
```

A separate routine (Showschd.cpp) displays all SigGen indices and their corresponding variable values (45 per page). This routine uses PullVars() to get variable values. This avoids recalculating the signal for each SGI. In previous examples, only the current variable values were displayed, and these were calculated when CalcSig() was called.

## DAMA Buffers

```
trash();
dropall();
ETldrop(1);
neuralbuf=_allot16(neuralnpts);
sigbuf=_allot16(neuralnpts);
```

```
InitSig(Sig1);
Sig1->si=SIGI;
pflag1 = CalcSig(Sig1);
if (pflag1 == 2)
    break;
if (playmode==3)
{
    InitSig(Sig2);
    Sig2->si=SIGI;
    CalcSig(Sig2);
}
```

```
LoadNorm(Sig1);
if (playmode==3)
    LoadNorm(Sig2);
```

DAMA buffers are used to store spike data. All DAMA buffers are cleared with the `trash()` command after every SigGen Index has been completed. The program reallocates DAMA buffers for signals, recordings, and normalization files after the `trash()` command.

The recording buffers are reallocated using the APOS `_allot16()` command with the new DAMA buffer id.

`InitSig()` is called to reallocate a DAMA buffer for the SigGen signal. It is not necessary to call `LoadSig()`, because the signal structures are still loaded. `CalcSig()` calculates the signal and pops it to the DAMA.

In the other examples, normalization files are loaded when `LoadSig()` is called (which calls `LoadNorm()`). However, to reload them after clearing the DAMA buffers, it is only necessary to call `LoadNorm()`.

---

---

***Part***  
***3***

***Software Reference***



# Chapter 5 SigGen Engine Functions

## File Handling Routines

These routines load and save SigGen signal files.

### LoadSig ( \*Sig, \*fname );

<b>Prototype</b>	int LoadSig( SigType* Sig, const char *fname);
<b>Operation</b>	Loads a SigGen signal from a file. Calls InitVars() and LoadNorm() after loading signal.
<b>Arguments</b>	*Sig pointer to SigGen signal into which to load *fname pointer to DOS file name
<b>Returns</b>	0 if cannot load file or normalization file (if applicable) 1 otherwise
<b>Called by</b>	
<b>Comments</b>	Is able to load previous revisions of SigGen files and fills in default values for parameters that aren't saved in those files.

### SaveSig ( \*Sig, \*fname );

<b>Prototype</b>	int SaveSig( SigType* Sig, const char *fname);
<b>Operation</b>	Saves SigGen signal
<b>Arguments</b>	*Sig pointer to SigGen signal *fname pointer to DOS file name
<b>Returns</b>	0 if can not open file for saving 1 otherwise
<b>Called by</b>	

### LoadNorm ( \*Sig );

<b>Prototype</b>	int LoadNorm(SigType* Sig);
<b>Operation</b>	Loads normalization file if normflag is set. Allocates DAMA buffers (16k floats) for magnitude and phase normalization.
<b>Arguments</b>	*Sig pointer to SigGen signal
<b>Returns</b>	result of ReadResponse()
<b>Called by</b>	LoadSig()

## Signal Preparation and Generation

Signal preparation and generation is primarily performed by calling the `InitSig()` and `CalcSig()` functions. The other functions listed below are used by these functions to generate the signals. You could use them if you wanted to create a SigGen-like program that allowed signal manipulation.

### InitSig ( \*Sig );

<b>Prototype</b>	<code>void InitSig( SigType *Sig);</code>
<b>Operation</b>	Prepares the AP2 for the SigGen signal by allocating zero value stack and DAMA buffers the length of the signal.
<b>Arguments</b>	<code>*Sig</code> pointer to SigGen signal
<b>Returns</b>	n/a
<b>Called by</b>	
<b>Comments</b>	Must have enough AP2 memory for signal. Will generate error if not enough memory.

### CalcSig ( \*TSig );

<b>Prototype</b>	<code>int CalcSig(SigType *TSig);</code>
<b>Operation</b>	Builds signal on stack by combining segments as specified by their application method ( <code>combcodes=add, multiply, subtract, etc</code> ). Scales signal and pops to DAMA buffer number ( <code>Tsig-&gt;id</code> ).
<b>Arguments</b>	<code>*TSig</code> pointer to SigGen signal
<b>Returns</b>	0 error building signal 1 OK 2 termination condition reached (no signal built)
<b>Called by</b>	

### InitSigStuff ( );

<b>Prototype</b>	<code>void InitSigStuff();</code>
<b>Operation</b>	Initializes global prompt strings for SigGen signal components.
<b>Arguments</b>	
<b>Returns</b>	n/a
<b>Called by</b>	
<b>Comments</b>	Used by SigGen to build user interface.

## DoNewSig ( \*Sig );

<b>Prototype</b>	void DoNewSig(SigType *Sig);
<b>Operation</b>	Creates the default SigGen signal.
<b>Arguments</b>	*Sig      pointer to SigGen signal
<b>Returns</b>	n/a
<b>Called by</b>	

## ChangeSigNpts ( \*Sig );

<b>Prototype</b>	void ChangeSigNpts(SigType *Sig);
<b>Operation</b>	Deallocates the signal DAMA buffer for the given signal, if it exists, then reallocates it with the current signal length.
<b>Arguments</b>	*Sig      pointer to SigGen signal
<b>Returns</b>	n/a
<b>Called by</b>	
<b>Comments</b>	Call this if you want to change the length of the signal after it has already been initialized.

## CalcSeg ( \*TSig, sn );

<b>Prototype</b>	void CalcSeg(SigType *TSig, int sn);
<b>Operation</b>	Generates specified segment by adding components together, using specified gating, and applying normalization if a frequency-domain component.
<b>Arguments</b>	*TSig      pointer to SigGen signal sn          segment number
<b>Returns</b>	n/a
<b>Called by</b>	CalcSig()

## GetNewSeg ( \*Sig );

<b>Prototype</b>	int GetNewSeg(SigType *Sig);
<b>Operation</b>	Creates a new segment with default values. If successful, increments total number of active segments.
<b>Arguments</b>	*Sig      pointer to SigGen signal
<b>Returns</b>	number of new segment, or -1 if maximum number of segments already in use
<b>Called by</b>	DoNewSig()

**KillSeg ( \*Sig, sn );**

<b>Prototype</b>	void KillSeg(SigType *Sig, int sn);
<b>Operation</b>	Deletes segment and decrements total number of active segments.
<b>Arguments</b>	sn            segment number
<b>Returns</b>	n/a
<b>Called by</b>	

**GenGate ( \*k, n, factor, rf, srate );**

<b>Prototype</b>	int GenGate(float *k, int n, float <i>factor</i> , float <i>rf</i> , float <i>srate</i> );
<b>Operation</b>	Creates a Blackman window on top of the stack
<b>Arguments</b>	*k            pointer to coefficients n            order of window <i>factor</i> <i>rf</i> rise/fall time of gate (ms) <i>srate</i> sample rate ( $\mu$ s)
<b>Returns</b>	
<b>Called by</b>	CalcSeg()

**CheckBoundry ( \*Sig ) ;**

<b>Prototype</b>	int CheckBoundry(SigType *Sig);
<b>Operation</b>	Returns "true" if any variable that uses Boundary Control has exceeded its min/max limits. See fflag and PullVars().
<b>Arguments</b>	*Sig        pointer to SigGen signal
<b>Returns</b>	1            if limits exceeded 0            if still within bounds
<b>Called by</b>	PullVars()

## FormFName ( *sss*, \*Sig, \*Cmp )

<b>Prototype</b>	void FormFName(char <i>sss</i> [], SigType* <i>Sig</i> , CmpType* <i>Cmp</i> );
<b>Operation</b>	If the specified component is of type FileF or File16, Returns the name of the raw binary file. Appends "000" to file name if constant, or the integer value of the variable it is set to.
<b>Arguments</b>	<i>sss</i> returned file name * <i>Sig</i> pointer to SigGen signal * <i>Cmp</i> pointer to component number
<b>Returns</b>	file name in <i>sss</i>
<b>Called by</b>	CalcSeg()
<b>Comments</b>	File name is returned only for File[16 F]_[T F] component types.

## ReadResponse ( \*fname, *srate*, *id1*, *id2* );

<b>Prototype</b>	int ReadResponse(char * <i>fname</i> , float <i>srate</i> , int <i>id1</i> , int <i>id2</i> );
<b>Operation</b>	Loads a normalization file to DAMA buffers <i>id1</i> and <i>id2</i> .
<b>Arguments</b>	* <i>fname</i> pointer to file name of text file <i>srate</i> D/A sample period (us) <i>id1</i> DAMA buffer for magnitude normalization <i>id2</i> DAMA buffer for phase normalization; set to zero if phase normalization not used
<b>Returns</b>	0                unable to open file 1                otherwise
<b>Called by</b>	LoadNorm()
<b>Comments</b>	Requires enough AP2 memory for two NORM_SIZE (16k) floating-point buffers, plus one more if <i>id2</i> !=0. Ignores blank lines and lines starting with zero or a space.

## Handling Variables

Variable handling is automatically performed by other SigGen Engine functions, such as LoadSig() and CalcSig() by calling the following variable handling routines.

### PullVars ( \*Sig );

<b>Prototype</b>	void PullVars(SigType *Sig);
<b>Operation</b>	Calculates and sets the current value for all variables. Sets the variable's <i>fflag</i> to 1 if exceeds min/max, 0 otherwise. Sets the signal's <i>allfin</i> flag to value returned by CheckBoundry().
<b>Arguments</b>	*Sig pointer to SigGen signal
<b>Returns</b>	n/a
<b>Called by</b>	CalcSig()
<b>Comments</b>	Ignores variables named "." For illegal variable types, <i>curval</i> is set to -0.9999

### GetVal ( \*Sig, p );

<b>Prototype</b>	float GetVal(SigType* Sig, ParType p);
<b>Operation</b>	Determines if the given parameter is a constant or tracked off a SigGen variable, and returns the appropriate value of that parameter. If it is tracked off a SigGen variable and the variable is a combination variable, the combination value is returned. If a variable is defined as an alternating variable, GetVal() will alternately return the MIN or MAX value each time it is called.
<b>Arguments</b>	*Sig pointer to SigGen signal <i>p</i> parameter
<b>Returns</b>	value of the given parameter
<b>Called by</b>	FormFName(), CalcSeg(), SetAtten(), SetTG6(), SetMasker()

## ReadVarVal ( \*Sig, p );

<b>Prototype</b>	float ReadVarVal(SigType* <i>Sig</i> , ParType <i>p</i> );
<b>Operation</b>	Same as GetVal(), but does not implement alternating variables.
<b>Arguments</b>	<i>*Sig</i> pointer to SigGen signal <i>p</i> parameter
<b>Returns</b>	value of the given parameter
<b>Called by</b>	

## InitVars ( \*Sig );

<b>Prototype</b>	void InitVars(SigType* <i>Sig</i> );
<b>Operation</b>	Sets the current value of each SigGen variable to its default value and sets <i>numitems</i> to number of steps, if applicable. Calls external callback InitDynVar() if variable is dynamic.
<b>Arguments</b>	<i>*Sig</i> pointer to SigGen signal
<b>Returns</b>	n/a
<b>Called by</b>	LoadSig()

## LoadSched ( \*VT );

<b>Prototype</b>	int LoadSched( VarType* <i>VT</i> );
<b>Operation</b>	Loads a list of values for the specified variable.
<b>Arguments</b>	<i>*VT</i> pointer to SigGen variable
<b>Returns</b>	0            unable to open schedule or error in schedule file. 1            successful
<b>Called by</b>	InitVars(), PromptVars1()

## CalcSteps ( \*V );

<b>Prototype</b>	int CalcSteps(VarType * <i>V</i> );
<b>Operation</b>	Returns number of steps required to run through all the values of a SigGen variable.
<b>Arguments</b>	<i>*V</i> pointer to SigGen variable
<b>Returns</b>	9999      for  step size  < 1e <sup>-5</sup> # steps    for variable types of LINSTEP and LOGSTEP[2 10] 0            for all other variable types
<b>Called by</b>	InitVars()

## Prompted Variables

Prompted variables are a variable type that can be selected in SigGen. They allow the user to request that the application prompt the operator for a value of a variable at run-time. In SigGen the user can select either Level 1 or Level 2 prompted variables. You must call `PromptVars1()` and `PromptVar2()` to prompt for user input of prompted variables. These functions call `AskForVal()` and `AskForFile()` in `SigECall.cpp`. You can customize how variables are prompted by changing the `AskForVal()` or `AskForFile()` functions.

### PromptVars1 ( \*Sig );

<b>Prototype</b>	<code>void PromptVars1(SigType* Sig);</code>
<b>Operation</b>	Prompts for input of all variables defined as Level One. If no value is provided by the user, uses default value. If returned value is out of range, uses applicable min/max value. Sets current value of SigGen variable to result.
<b>Arguments</b>	<i>*Sig</i> pointer to SigGen signal
<b>Returns</b>	n/a
<b>Called by</b>	
<b>Comments</b>	Uses external callback <code>AskForVal()</code> .

### PromptVars2 ( \*Sig );

<b>Prototype</b>	<code>void PromptVars2(SigType* Sig);</code>
<b>Operation</b>	Same as <code>PromptVars1()</code> , for Level Two (PMPT_EACH) variables.
<b>Arguments</b>	<i>*Sig</i> pointer to SigGen signal
<b>Returns</b>	n/a
<b>Called by</b>	
<b>Comments</b>	Uses external callback <code>AskForVal()</code> .

## Controlling Peripherals

SigGen Engine simplifies control of peripheral modules. These modules can be used to produce continuous, band-limited noise, can attenuate and gate signals, and can control timing of stimulus presentation. The following are the SigGen Engine commands that are used to control peripheral modules.

### SetAtten ( \*Sig, attadj );

<b>Prototype</b>	float SetAtten(SigType* <i>Sig</i> , float <i>attadj</i> );
<b>Operation</b>	Sets attenuation for PA4 defined in SigGen's Setup Signal dialog
<b>Arguments</b>	<i>*Sig</i> pointer to SigGen signal <i>attadj</i> amount to add to existing attenuation
<b>Returns</b>	Sig->atten if the specified PA4 is in the XBUS rack, 0.0 otherwise
<b>Called by</b>	
<b>Comments</b>	Performs range checking to keep attenuation sent to PA4 between 0.0 and 99.9 dB.

### SetMasker ( \*Sig, mn, addadj, sumode );

<b>Prototype</b>	int SetMasker(SigType* <i>Sig</i> , int <i>mn</i> , float <i>addadj</i> , int <i>sumode</i> );
<b>Operation</b>	Configures the peripheral devices based on the specified setup
<b>Arguments</b>	<i>*Sig</i> SigGen record <i>mn</i> masker number [0 or 1]; aka Setup #1 or Setup #2 from SigGen peripheral setup <i>addadj</i> amount to add to PA4 attenuation (in dB) <i>sumode</i> update mode: SM_UPDATE=0 for updating maskers SM_INIT=1 for first-time initialization (clears peripherals) SM_CAL=2 for calibration (issues software triggers for the WG1/2 and SW2)
<b>Returns</b>	1 if an error occurs (including illegal PF1 frequency values, errors configuring peripherals) 0 if no errors occur
<b>Called by</b>	
<b>Comments</b>	Will not generate any error if devices are not present that are requested in setup.

## MaskerOff ( \*Sig, mn );

<b>Prototype</b>	void MaskerOff(SigType* <i>Sig</i> , int <i>mn</i> );
<b>Operation</b>	Turns off WG1/2 if they are enabled in peripheral setup <i>mn</i> and they are present in the XBUS rack.
<b>Arguments</b>	<i>*Sig</i> pointer to SigGen signal <i>mn</i> Peripheral Setup # [0 or 1]
<b>Returns</b>	n/a
<b>Called by</b>	

## SetTG6 ( \*Sig, sn, oodur, reps, os3, dur3, os6, dur6 );

<b>Prototype</b>	int SetTG6(SigType* <i>Sig</i> , int <i>sn</i> , float <i>oodur</i> , int <i>reps</i> , float <i>os3</i> , <i>dur3</i> , <i>os6</i> , <i>dur6</i> );
<b>Operation</b>	Defines a timing sequence for all six channels of the TG6.
<b>Arguments</b>	<i>*Sig</i> pointer to SigGen signal <i>sn</i> TG6 sequence number (0..7) <i>oodur</i> overall duration of TG6 timing sequence, in ms <i>reps</i> number of times to to play sequence after trigger; if 0, will play sequence continuously until TG6stop() is called <i>os3/6</i> onset delay of chan 3 & 6, in ms <i>dur3/6</i> duration of chan 3 & 6 pulses, in ms; if <= 0.0, no timing sequence is set for channel
<b>Returns</b>	0 if successful 1 if XBUS error occurs
<b>Called by</b>	
<b>Comments</b>	Resolution of timing is 1 ms

TG6 is configured in continuous mode (plays the timing sequence *reps* times after triggering)

Channel assignments are as follows:

- 1 1 ms pulse at beginning of sequence (can be used for oscilloscope trigger).
- 2 1 ms pulse at end of onset delay (for D/A trigger). Onset delay comes from Sig->osdel.
- 3 controlled by *os3* and *dur3* arguments.
- 4 Peripheral Setup 1 timing signal, if enabled. Uses Sig->mask[0] values.
- 5 Peripheral Setup 2 timing signal, if enabled. Uses Sig->mask[1] values.
- 6 controlled by *os6* and *dur6* arguments.

Note: BioSig uses channel 3 for A/D triggering.

## External Callback Functions

SigGen Engine requires that you define some functions to accept user input, handle errors, and control dynamic variables. These “external callback” functions allow you to control how SigGen Engine interacts with the user. These callback functions are defined in SigECall.cpp, and are designed for DOS applications.

### ErrHand ( s1[], s2[] );

<b>Prototype</b>	void ErrHand(char <i>s1</i> [], char <i>s2</i> []);
<b>Operation</b>	Print error messages <i>s1</i> and <i>s2</i> . Halt program run.
<b>Arguments</b>	<i>s1</i> error message string <i>s2</i> error message string
<b>Returns</b>	n/a
<b>Called by</b>	Used by most SigGen functions for general error handling.

### AskForVal (\*v, \*prompt, \*units, defv, minv, maxv);

<b>Prototype</b>	int AskForVal(float* <i>v</i> , char* <i>prompt</i> , char * <i>units</i> , float <i>defv</i> , float <i>minv</i> , float <i>maxv</i> );
<b>Operation</b>	Prompts user input for value, and check against acceptable range.
<b>Arguments</b>	* <i>v</i> pointer to current value or value entered by user * <i>prompt</i> pointer to string to screen to prompt user input * <i>units</i> pointer to units string <i>defv</i> default value <i>minv</i> minimum value <i>maxv</i> maximum value
<b>Returns</b>	1        always
<b>Called by</b>	PromptVars1(), PromptVars2()

## AskForFile ( \*sss, \*prompt );

<b>Prototype</b>	int AskForFile(char* <i>sss</i> , char* <i>prompt</i> );
<b>Operation</b>	Prompt user to enter file name
<b>Arguments</b>	<i>*sss</i> not used <i>*prompt</i> pointer to prompt string for user input
<b>Returns</b>	0 if does not include extension (searches for '.') 1 if includes extension
<b>Called by</b>	PromptVars1()

## GetDynVar ( \*VT, vn );

<b>Prototype</b>	float GetDynVar(VarType * <i>VT</i> , int <i>vn</i> );
<b>Operation</b>	Return current value of dynamic variable <i>VT</i> .
<b>Arguments</b>	<i>*VT</i> pointer to VarType Structure <i>vn</i> variable id number
<b>Returns</b>	default value ( <i>VT</i> ->defval)
<b>Called by</b>	PullVars()
<b>Comments</b>	Modify this code when you want to use a dynamic variable. This function is called via PullVars() from CalcSig() for every dynamic variable in the signal.

## InitDynVar ( \*VT, vn );

<b>Prototype</b>	void InitDynVar(VarType * <i>VT</i> , int <i>vn</i> );
<b>Operation</b>	Loads dynamic variable with default value
<b>Arguments</b>	<i>*VT</i> pointer to VarType Structure <i>vn</i> variable id number
<b>Returns</b>	n/a
<b>Called by</b>	InitVars();

## CE ( mark[] );

<b>Prototype</b>	int CE(char <i>mark</i> []);
<b>Operation</b>	If XBUS or AP2 error, calls error handler with error message
<b>Arguments</b>	<i>mark</i> [] not used
<b>Returns</b>	0 if no errors
<b>Called by</b>	InitSig(), ChangeSigNpts(), CalcSeg(), CalcSig(), GenGate(), SetTG6(), SetMasker(), LoadNorm()

# Chapter 6 SigGen Data Structures

These data structures are used by SigGen Engine to load and process the information in the SigGen signal file. Default values are shown in brackets [default].

## SigType: Signal Structure

SigType contains all the information needed to build a SigGen signal. This includes the DAMA buffer in which the signal is stored, the current SigGen index, the signal file name, attenuation for a programmable attenuator, the signal duration, sampling rate, and number of points.

Default values are initialized by SigGen.

### SigType Structure

```
struct SigType
{
int      id;                DAMA buffer ID
int      si;                current SigGen index (SGI) [0]
int      mpflag;           (not used in SigGen Engine)
int      maxsi;            max SGI (not calculated in SigGen Engine)
char     allfin;           result of CheckBoundry(); 0=no boundaries reached, 1=at
                           least one variable reached boundary

char     name[30];         signal Name ['.....']
char     fname[94];       signal File Name ['...']
ParType  osdel;            signal onset delay; used by SetTG6 to set 1ms pulse from
                           channel 2 after osdel (D/A start) [0,0.0]

float    dur;              signal duration in ms [100]
float    srate;            sampling period (us) [20.0]
long     npts;             number of points in signal
float    sf;               scaling factor [1] (not used by SigGen Engine)
int      attdev;           PA4 device number for setting Signal attenuation (different
                           from PA4 in peripheral setup) [1]
ParType  atten;           attenuation (dB) for PA4 [0,0.0]
float    callev;           calibration level (used by CalcSeg) [0.0]
float    calvolt;         calibration voltage (used by CalcSeg) [1.0]
int      nsegs;           number of segments [0]
int      dacchan;         DA channel for signal conversion [1]
VarType  Var[MAXVARS];    signal variables
SegType  Seg[MAXSEGS];    signal segments
long     rec_type;        record type for different versions of SigGen files
int      normflag;        use normalization curve=1; don't use=[0]
int      norm_id1;        DAMA buffer number for frequency normalization curve [0]
int      norm_id2;        DAMA buffer number for phase normalization curve [0]
char     normfname[100];  normalization file name ['typical.nrm']
MaskType Mask[2];         Peripheral Setup #1 Mask[0] Peripheral Setup #2 Mask[1]
char     dummy[1000-110-
(2*sizeof(MaskType))];   space reserved for future use
};
```



## VarType: Variable Structure

The VarType data structure contains variables for defining how variables are used in the SigGen signal. The VarType structure should be familiar to SigGen users from the Variable dialog box. VarType includes variables generated by one of the stepping functions (such as linear step or log step) and combination variables.

### Control flag Definitions

```
#define TC_NORMAL      0
#define TC_BOUND      1
#define TC_LOOP       2
```

### Generation Method Definitions

```
#define UNDEFED      0
#define CONSTANT     1
#define LINSTEP      2
#define LOGSTEP2     3
#define LOGSTEP10    4
#define RANDOM       5
#define PRMP_ONE     6
#define PRMP_EACH    7
#define CONST_FILE   8
#define PRMP_FILE    9
#define ALTING      10
#define VALLIST     11
#define DYNAMIC     12
#define LAST_SRC    13
```

### Variable Source Definitions

```
#define _UNDEFED      0x0001
#define _CONSTANT     0x0002
#define _LINSTEP      0x0004
#define _LOGSTEP2     0x0008
#define _LOGSTEP10    0x0010
#define _RANDOM       0x0020
#define _PRMP_ONE     0x0040
#define _PRMP_EACH    0x0080
#define _CONST_FILE   0x0100
#define _PRMP_FILE    0x0200
#define _ALTERNATE    0x0400
#define _VALLIST     0x0800
#define _DYNAMIC     0x1000
```

### Combination Operation Definitions

```
#define _V_P_Vc      0
#define _V_M_Vc      1
#define _Vc_M_V      2
#define _V_T_Vc      3
#define _V_D_Vc      4
#define _Vc_D_V      5
```

**VarType Structure**

```

struct VarType
{
char    name[15];           variable name ['.....']
char    units[5];          variable units [0]
char    prompt[60];        prompt for variable ["Unknown Prompt"]
int     source;            variable source/step method (e.g
                           LINSTEP,LOGSTEP2,VALLIST) [UNDEFED]

char    fname[100];        variable file name[...]
int     numitems;          number of steps of variable in schedule [0]
int     repfact;           repeat factor    [1]
int     skipfact;          skip factor      [1]
int     sgioffset;         SGI offset      [0]
char    fflag;             flag for boundary condition; [0]=within boundary, 1=at boundary
char    termctrl;          termination control method [TC_NORMAL]
float   vlist[MAXFILE];    value list    [0]
int     combvar;           combination variable [0] =none, 1=comb var used with constant
                           stored in combval, >1=comb variable used with another variable
                           with variable id = (combvar-2)

int     combop;            combination operation    [_V_P_Vc] (curval + combval)
float   combval;           constant value for combination operation [0.0]
int     dummy[36];         reserved for future use
float   curval;            current value of variable (note: this does not account for any
                           combination operation) [0.0]
float   defval;            default value of variable [0.0]
float   minv;              minimum value of variable [-1e10]
float   maxv;              maximum value of variable [1.0e10]
float   stepv;             step value for variable [0.0]
};

```

## SegType: Segment Structure

SegType contains variables used for generating segments such as, start time, duration, gate type, and generation method. Defaults [] shown are set in GetNewSeg().

### Generation Method Definitions

```
#define TIME_METH      0
#define FREQ_METH     1
```

### Application Method Definitions

```
#define ADD_CC        0
#define MULT_CC       1
#define SUB_CC        2
#define DIV_CC        3
#define OMX_CC        4
#define XMO_CC        5
```

### Gate Type Definitions

```
#define NONE_SHAPE    0
#define COS2_SHAPE    1
#define HANN_SHAPE    2
#define RAMP_SHAPE    3
#define BLACKMAN_SHAPE 4
```

### SegType Structure

```
struct SegType
{
  int      act           0=segment inactive/unused; 1=active [1]
  char     name[30];    not used
  int      meth;        generation method: [TIME_METH] or FREQ_METH
  int      comocode;    combination code (..._CC) ("application method")[ADD_CC]
  int      gtype;       gate type (..._SHAPE) [COS2_SHAPE]
  int      fftpts;      #pts for fft (radix 2) [4096]
  ParType  amp;         "level" [0,0.0]
  ParType  start,dur;   start time and duration, ms [0,0.0] [0,20.0]
  ParType  gtime;       gate rise/fall time [0,5.0]
  CmpType  Cmp[MAXCMPS]; signal components
};
```

## CmpType: Component Structure

The component structure contains a variable for how the component is generated (e.g. tone, sweep, click).

### Component Constants

#### (Time)

```
#define NOP_T          0
#define TONE_T        1
#define SWEEP_T       2
#define GAUSS_T       3
#define FLAT_T        4
#define CLICK_T       5
#define DC_T          6
#define FILE16_T      7
#define FILEF_T       8
```

### Component Constants

#### (Freq)

```
#define NOP_F          10
#define TONE_F         11
#define BAND_F         12
#define RANGE_F        13
#define CLICK_F        14
#define HARM_F         15
#define DC_F           16
#define FILE16_F       17
#define FILEF_F        18
```

### CmpType Structure

```
struct CmpType
{
  int      code;                component type [NOP_T]
  ParType arg[MAXARGS];        parameters [0,0.0]
  char     txt[30];             ["" ]
  int      sufsrc;              [0]
};
```

## MaskType: Peripheral Structure

The MaskType structure implements the peripheral setups specified by the user. MaskType contain variables for control of the programmable filter (PF1), waveform generator (WG1/2), cosine switch (SW2), and programmable attenuators (PA4).

### Filter Types for WG1/2

```
#define PF_BYPASS      0
#define PF_LOPASS     1
#define PF_HIPASS     2
#define PF_BANDPASS   3
#define PF_BANDREJ    4
#define PF_RANGEPASS  5
#define PF_RANGEREJ   6
```

### MaskType Structure

```
struct MaskType
{
    int      TMenable;           Peripheral Timing Setup Enabled=1, Not Enabled=[0]
    ParType  TMosdel;           TG6 onset delay [0,0.0] (used by SetTG6())
    ParType  TMDur;            TG6 duration [0,10.0] (used by SetTG6())
    int      WGdev;            WG1/2 device number [0]
    int      W Gowu;           owu = "off[mute] while updating" [0]
    int      W Gfreerun;       [1]
    float    W Grftime;        WG rise-fall time [1.0] (see XBDRV WG1/2 manual)
    int      W Gshape;         WG waveform shape [0] (see XBDRV WG1/2 manual)
    ParType  W Gfreq;          WG frequency [0, 1000.0] (see XBDRV WG1/2 manual)
    ParType  W Gphase;         WG phase [0, -1.0] (see XBDRV WG1/2 manual)
    ParType  W Gswrt;          sine wave sweep rate [0, 0.0] (see XBDRV WG1/2 manual)

    int      SWdev;           Cosine Switch (SW2) device number [0]
    int      SWowu;           owu = off [mute] while updating [0]
    int      SWshape;         SW2 gating shape [ONOFF] (see XBDRV SW2 manual)
    ParType  SWrftime;        SW2 rise-fall time [0, 1.0] (see XBDRV SW2 manual)

    int      PFdev;           PF1 device number [0]
    int      PFshape;         PF1 Filter Type [PF_BYPASS] (see XBDRV PF1 manual)
    int      PForder;         filter order used by Filt_gen [8]
    ParType  PFgain;          PF1 gain [0,0.0] (see XBDRV PF1 manual)
    ParType  Pffreq1;         frequencies used by Filt_gen [0, 1000.0]
    ParType  Pffreq2;         [0, 2000] (see XBDRV PF1 manual)

    int      PAdev;           PA4 device number [0]
    int      PAowu;           owu = "off[mute] while updating" [0]
    ParType  PAatt;           PA4 attenuation [0, 0.0]
};
```

## ParType: Parameter Structure

The ParType structure holds either a variable ID number, or the value of the parameter if it is a constant.

### ParType Structure

```
struct ParType
{
int      s;           variable id#, 0 if constant
float    v;           value of parameter, when constant
};
```

# Index

- AskForFile, **5-12**
- AskForVal, **5-11**
- Attenuation, 2-8
- CalcSeg, **5-3**
- CalcSig, 2-5, 3-4, **5-2**
- CalcSteps, 7
- CE, **5-12**
- ChangeSigNpts, **5-3**
- CheckBoundry, **5-4**
- CmpType, **6-6**
- Combination Variables, 2-10
- DoNewSig, **5-3**
- Dynamic Variables, 2-9
- ErrHand, **5-11**
- FormFName, **5-5**
- GenGate, **5-4**
- GetDynVar, 3-13, **5-12**
- GetNewSeg, **5-3**
- GetVal, **6**
- InitDynVar, **5-12**
- InitSig, 2-4, 3-3, **5-2**
- InitSigStuff, 3-3, **5-2**
- InitVars, 7
- KillSeg, **5-4**
- LoadNorm, 4-10, **5-1**
- LoadSched, 7
- LoadSig, 2-4, 3-3, **5-1**
- MaskerOff, **5-10**
- MaskType, **6-7**
- Normalization Files, 2-8
- ParType, **6-8**
- Peripheral Setups, 2-9
- Prompted Variables and Files, 2-10
- PromptVars1, 3-3, **8**
- PromptVars2, **8**
- PullVars, 4-9, **6**
- ReadResponse, **5-5**
- ReadVarVal, 7
- SaveSig, **5-1**
- SegType, **6-5**
- SetAtten, 3-4, **5-9**
- SetMasker, 3-3, 3-4, **5-9**
- SetTG6, 3-3, **5-10**
- SigType, 2-4, 3-2, **6-1**
  - Sig->id, 2-5
  - Sig->npts, 2-4
  - Sig->si, 2-5
  - Sig->srates, 2-4
- Triggering with the TG6, 2-8
- VarType, 3-2, **6-3**
  - VT->curval, 3-5, 3-13
  - VT->name, 3-5
  - VT->source, 3-13

