GT98901

Multi-Function USB Educational Board

GTDEMO Software

User's Guide

Last updated November 24, 2014

MARVIN TEST SOLUTIONS

Safety and Handling

Each product shipped by Marvin Test Solutions is carefully inspected and tested prior to shipping. The shipping box provides protection during shipment, and can be used for storage of both the hardware and the software when they are not in use.

The circuit boards are extremely delicate and require care in handling and installation. Do not remove the boards from their protective plastic coverings or from the shipping box until you are ready to install the boards into your computer.

If a board is removed from the computer for any reason, be sure to store it in its original shipping box. Do not store boards on top of workbenches or other areas where they might be susceptible to damage or exposure to strong electromagnetic or electrostatic fields. Store circuit boards in protective anti-electrostatic wrapping and away from electromagnetic fields.

Be sure to make a single copy of the software CD for installation. Store the original CD in a safe place away from electromagnetic or electrostatic fields. Return compact disks (CD) to their protective case or sleeve and store in the original shipping box or other suitable location.

Warranty

Marvin Test Solutions products are warranted against defects in materials and workmanship for a period of 12 months. Marvin Test Solutions shall repair or replace (at its discretion) any defective product during the stated warranty period. The software warranty includes any revisions or new versions released during the warranty period. Revisions and new versions may be covered by a software support agreement. If you need to return a board, please contact Marvin Test Solutions Customer Technical Services Department via http://www.marvintest.com/magic/ - the Marvin Test Solutions on-line support system.

If You Need Help

Visit our web site at http://www.marvintest.com for more information about Marvin Test Solutions products, services and support options. Our web site contains sections describing support options and application notes, as well as a download area for downloading patches, example, patches and new or revised instrument drivers. To submit a support issue including suggestion, bug report or questions please use the following link: http://www.marvintest.com/magic/

You can also use Marvin Test Solutions technical support phone line (949) 263-2222. This service is available between 8:30 AM and 5:30 PM Pacific Standard Time.

Disclaimer

In no event shall Marvin Test Solutions or any of its representatives be liable for any consequential damages whatsoever (including unlimited damages for loss of business profits, business interruption, loss of business information, or any other losses) arising out of the use of or inability to use this product, even if Marvin Test Solutions has been advised of the possibility for such damages.

Copyright

Trademarks

ATEasy®, CalEasy, DIOEasy®, DtifEasy, WaveEasy	Marvin Test Solutions, Inc. (Prior name, Geotest – Marvin Test Systems, Inc)
C++ Builder, Delphi	Embarcadero Technologies Inc.
LabView, LabWindows^{tm}/CVI	National Instruments
Microsoft Developer Studio, Microsoft Visual C++, Microsoft Visual Basic,, .NET, Windows XP, VISTA, Windows 7 and 8	Microsoft Corporation

All other trademarks are the property of their respective owners.

Table of Contents

Chapter 1 - Introduction

Manual Scope and Organization

Manual Scope

This manual provides all the information necessary for installation, operation, and maintenance of the GT98901 multi-function demonstrational board. This manual assumes the reader has a general knowledge of PC based computers, Windows operating systems.

This manual also provides tutorial information for developing in ATEasy, an application development framework for functional test, automated test, and data acquisition. Finally, this manual covers programming information about using the GT98901 driver (referred to in this manual as **GTDEMO**). Therefore, this manual assumes a thorough understanding of Windows application development tools and languages.

Manual Organization

The GT98901 manual is organized in the following manner:

Chapter	**Content**
Chapter 1 – Introduction	Introduces the GT98901 manual. Lists all the supported boards and shows warning conventions used in the manual.
Chapter 2 – Overview	Provides the GT98901 list of features, description of the individual boards, architecture, specifications and the virtual panel description and operation.
Chapter 3 –Installation and Connections	Provides instructions on how to install a GT98901 board and the GTDEMO software.
Chapter 4 – ***ATEasy*** Setup and Installation	Step-by-step directions on how to install and configure ***ATEasy***.
Chapter 5 – Overview of ***ATEasy***	An overview of ***ATEasy*** including the various module files, submodules, and the Integrated Development Environment (IDE) layout, menus, and windows.
Chapter 6 – Your First Project	Describes the relationship of projects and applications, the workspace, creating a project, and project settings.
Chapter 7 – Getting Started With the GTDEMO Driver	Describes the provided ATEasy driver and how to use it within a project to control the various GT98901 subsystems.
Chapter 8 – Variables and Procedures	Explains how to declare variables, naming guidelines and properties of variables, and how to create and use procedures.
Chapter 9 – Drivers and Interfaces	Describes how to create and configure instrument drivers and user interfaces.
Chapter 10 – Commands	Defines ***ATEasy*** commands.
Chapter 11 – Working with Forms	Describes using forms to control programs and instrument operation.
Chapter 12 – Additional Exercises	Contains several step-by-step tutorials and descriptions of various facets of test application development.
Appendix A – Function Reference	Provides a list of the GTDEMO driver functions. Each function description provides syntax, parameters, and any special programming comments.

Chapter	Content
Appendix B – SCPI Function Reference	Provides a list of the SCPI functions that the GT98901 use. Each function description provides syntax, parameters, and any special programming comments.

Conventions Used in this Manual

Symbol Convention	Meaning
⚡	Static Sensitive Electronic Devices. Handle Carefully.
STOP	Warnings that may pose a personal danger to your health. For example, shock hazard.
❗	Cautions where computer components may be damaged if not handled carefully.
TIP	Tips that aid you in your work.

Formatting Convention	Meaning
`Monospaced Text`	Examples of field syntax and programming samples.
Bold type	Words or characters you type as the manual instructs. For example, function or panel names.
Italic type	Specialized terms. Titles of other references and information sources. Placeholders for items you must supply, such as function parameters

Chapter 2 - Overview

Introduction

The GT98901 is an educational demo board designed to help educate engineers learning about automated test equipment (ATE). A test engineer in an electronics-based industry is required to design test processes that ensure that a product or a system completely meets its stated specifications. The unit under test (“UUT”) could be as simple as a resistor or as complex as a space shuttle, with multiple sub-systems that require testing individually and as a complete system.

At the most basic level, ensuring that a device is working properly consists of verifying that it is doing what it claims to do according to a list of stated specifications. A complete test system must both provide stimulus and acquire data. Many test systems require testing multiple physical locations with a single measurement device. To handle this type of operation, the test system requires switching networks to route the signal path from test point to test equipment. Finally, test systems often include some user-interaction elements ranging from an LED indicating PASS or FAIL status to automated test logging and binning.

The GT98901 is a multi-function assembly that is capable of reading and writing analog and digital data through its analog-to-digital and digital-to-analog converter and digital input/output lines. The GT98901 provides basic switching capabilities with its SPDT relay as well as clock reference and visual interaction with the user. Due to its versatile design, the demo board can function as both a UUT or as a test system with another component as the UUT.

Features

The GT98901 educational demo board has the following features:

Four analog-to-digital converters (“ADC”) with 12 bits of resolution and a sampling clock capable of making up to 128 samples at up to 50 kHz.

Two digital-to-analog converters (“DAC”) with 8 bits of resolution that can be programmed as an arbitrary waveform generator with 128 samples per channel output at up to 234 kHz.

Eight digital input/output (“DIO”) channels. Direction of each channel can be configured independently.

One digital channel can be configured as a pulse width modulated (“PWM”) output.

Two single-pole double-throw (“SPDT”) relays.

Two user-controlled light-emitting diodes (“LED”) for visual feedback.

One momentary push button switch.

Three dip switches.

Adjustable reference clock output programmable from 3.6 kHz to 120 MHz.

Applications

Basic Test System operations.

Device Simulation.

ATE and ATEasy training.

Other educational applications.

Board Description

The GT98901 is an educational demo board consisting of a USB-controlled PCB assembly. The board has three ten-bit screw terminals for the analog and digital IO ports, the references, and the relays. For direct user interaction, the GT98901 has three dip switches and a push button. Visual feedback includes a LCD screen and two LEDs. Auditory feedback includes a speaker.

An analog-to-digital converter is a measurement device that analyzes a continuous (analog) voltage level and converts that voltage level to a digital value, which represent the waveform's amplitude. The four ADC channels are located on the right-hand screw terminal and labeled AIN1 through AIN4. These input only ports can be individually sampled via an API command or any of the four analog input ports can be set to digitize via a programmable sampling clock. This burst-mode of digization can sample anywhere between 1 and 128 samples. The ADCs can be used to make voltage measurements or to digitize a continuous waveform, allowing the user to analyze the waveform or store the waveform data on their test system.

A digital-to-analog converter behaves in the opposite manner of the ADC. The DAC will take in a digital code and output a corresponding voltage level. The two DAC ports are also on the right-hand screw terminal with the ADC terminals. Each channel can output an independent static voltage level or can be programmed with an array of values that can be continuously looped. Up to 128 points of data per channel can be stored and an arbitrary waveform can be generated at up to 234 kS/sec.

When an electrical subsystem needs to run synchronously, it does so by using a distributed clock signal. A reference clock is clock signal that oscillates with a high degree of accuracy with respect to it's state or desired nominal value. Reference clocks are often used to synchronize electronic subsystems. The GT98901 provides an adjustable reference clock output, which is programmable from 3.6 kHz to 120 MHz. The port is accessible from the right-hand screw terminal and is labeled CLK.

Storage and transmission of data between computer systems is more efficiently achieved when the data is passed digitally. A digital output channel can output a logical high or logical low digital signal. A digital input channel can read in an external voltage and resolve it to a logical high or logical low value. The digital input/output channel can perform both DI and DO operations. The eight DIO ports of the GT98901 are located on the left-hand screw terminal block and are labeled DO through D7. The digital ports conforms to TTL logic levels: where a logical high is between 0 and 0.8 volts and a logical high is between 2 and 3.3 volts. The digital ports support per channel data direction control and per channel data configuration.

Many of the components on the GT98901 run off a regulated 3.3-volt power supply. This power supply is accessible from the J2 screw terminal block. Since the power is shared between the other components, drawing more than 10 mA of current from the power supply may cause other components to fail.

Pulse Width Modulation (PWM) is a modulation technique that allows you to represent an analog voltage with a variable width digital pulse. In PWM, the duty cycle of a square wave is used to represent an analog voltage amplitude or current, etc. PWMs have both a user-configurable duty cycle and frequency. The frequency is the rate at which the signal is repeated. The duty cycle is the percentage of the square wave that thesignal is ON. The PWM is ON when the digital port is output at a logical high state and OFF when it is outputting a logical low. One digital channel of the GT98901, marked D6, can be configured as a PWM.

A switching component allows the user to change an electrical circuit by creating an open circuit where there was previously a short or vice versa. A single pole, double throw (SPDT) relay is a switching component that consists of a single common (CO) contact and two potential contacts that the CO can alternately be connected to. By default, the CO is shorted to the normally closed (NC) contact, but the operation of relay can be changed to break the connection between NC and CO and make a connection between CO and the normally open (NO) contact. Two single-pole double-throw ("SPDT") relays are installed on the GT98901. The contacts are accessible from the bottom screw terminal block and are labeled 1NC, 1CO, and 1NO for the first relay and 2NC, 2CO, and 2NO for the second relay.

Two user-controlled light-emitting diodes ("LED") are included in the lower right segment of the GT98901 for visual indication of user-defined events. LED2 is orange and LED3 is green.

A push button switch labeled SW1 is also located in the lower right-hand side of the GT98901. It is a momentary switch, which means that when the push button is pressed, a flag is raised in the GT98901's microcontroller. When the push button's flag is queried, it is also cleared. So, querying the push button flag will let you know whether the button has been pushed since the last time the status was queried.

On the lower left-hand side of the board are three dipswitches. The state of each dipswitch can be individually queried from the microcontroller. The dipswitch can be used to toggle user-defined features such as debugging display or error injection.

The speaker on the board can be toggled on and off and emits a 10kHz signal.

The most prominent user notification element on the GT98901 is the LCD display. It can display eight individual lines of 21 ASCII characters each. The user can clear the display and update each line independent from the other lines.

Figure 2-1 shows the GT98901 with its components labeled.

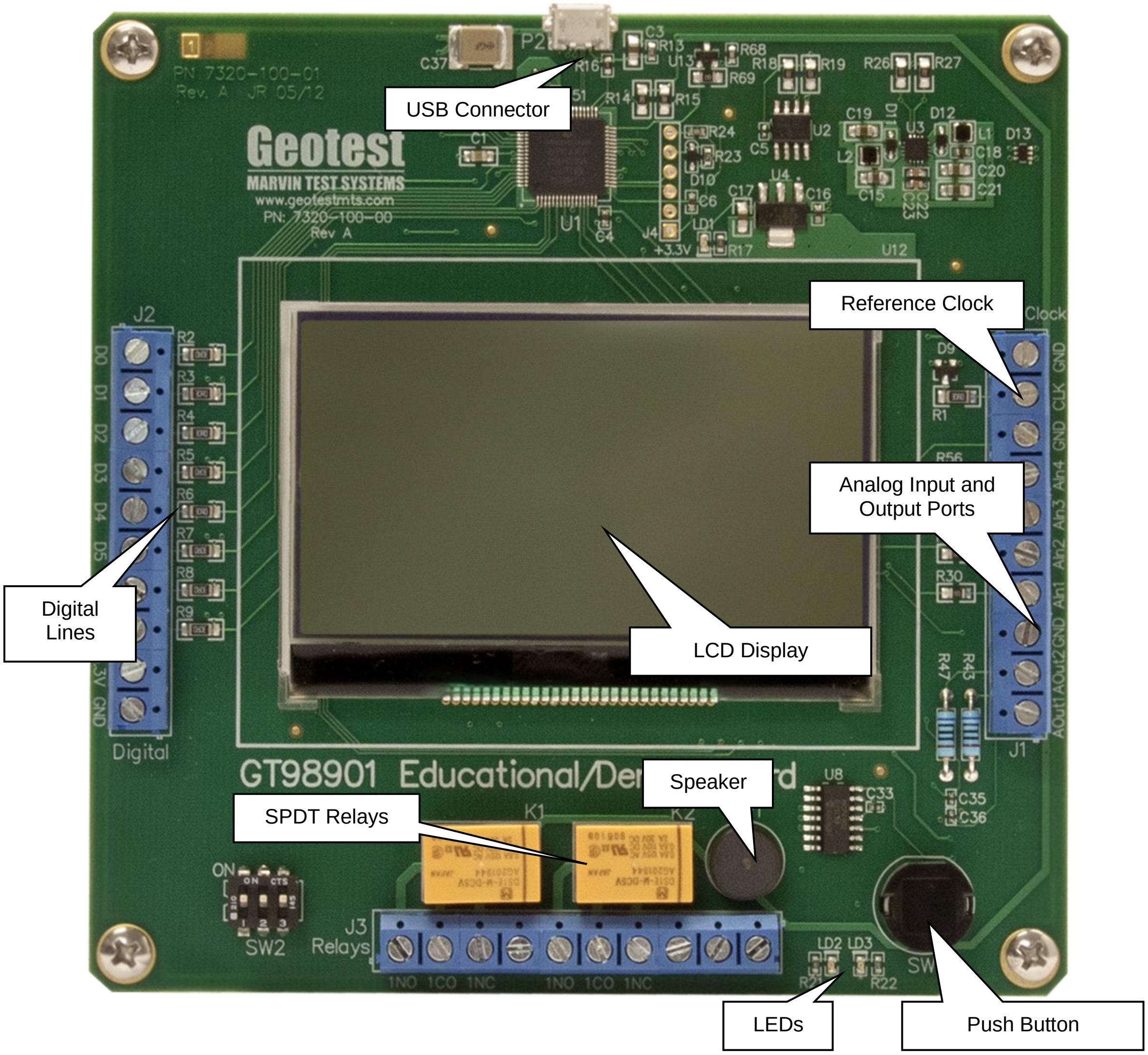


Figure 2-1: GT98901 Board Front View

Specifications

The following table outlines the specifications of the GT98901.

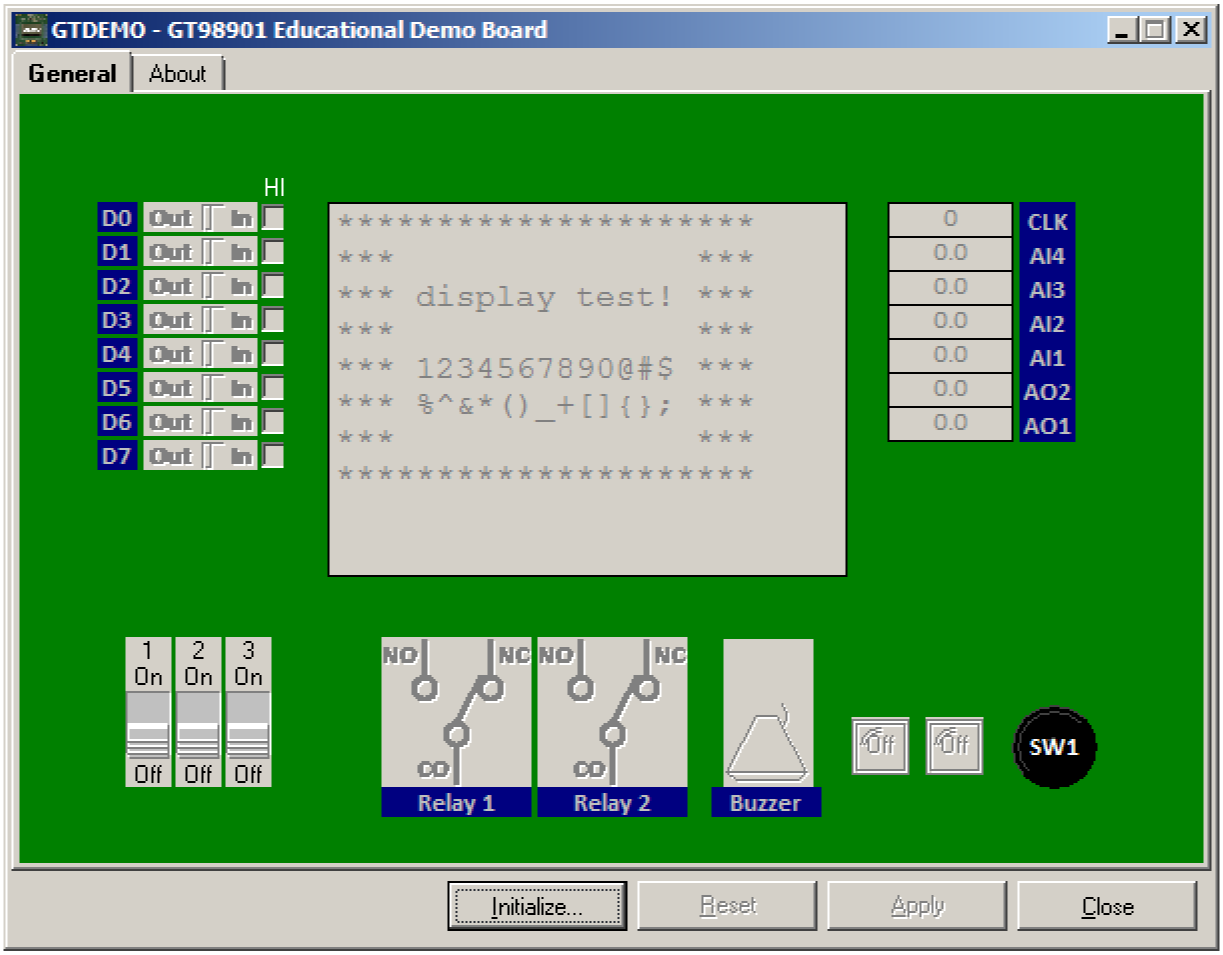
A to D Converter	
Number of Channels	4
Number of Bits	12
Input Range	±10 Volts
Sample Clock	50 kHz
Accuracy	±0.5% of Full Scale
Samples	Programmable from 1 to 128
D to A Converter	
Number of Channels	2
Number of Bits	8
Output Voltage	± 10 V
Accuracy	±0.5% of Full Scale
Sample Clock (Common for Both Channels)	234 kHz/N N = 1 to 65,535
Sample Memory	128 per channel
Digital I/O	
Number of Channels	8, configurable as input or output
Logical Level	LVTTL
PWM Output	One output can be configured as a pulse width modulated output
Miscellaneous I/O Functions	
Relays	(2) SPDT (Form C)
LEDs	2
Push Button	1, momentary, read by microcontroller
Buzzer	One, controlled by microcontroller
Dip Switches	Three, read by microcontroller
Reference Clock Output Level	LVTTL compatible
Reference Clock Frequency	120 MHz divided by 2^N (N is programmable from 0 to 15)

I/O Connections	
USB Port	One
I/O Terminal Connections	Three screw terminal ports supporting: Digital I/O Relay I/O Analog I/O Reference clock output

Virtual Panel Description

The **GTDEMO** includes a virtual panel program, which provides full access to the various configuration settings and operating modes. To understand the front panel operation, it is best to become familiar with the functionality of the board.

To open the virtual panel application, select **GtDemo Panel** from the **Marvin Test Solutions**, **GtDemo** menu under the **Start** menu. The GT98901 virtual panel opens as shown here:



```
*********************
***               ***
*** display test! ***
***               ***
*** 1234567890@#$ ***
*** %^&*()_+[]{}; ***
***               ***
*********************
```

Figure 2-2: GT98901 Virtual Panel (not Initialized)

The function of the panel button controls are shown below:

Initialize Opens the Initialize Dialog (see Initialize Dialog paragraph) in order to initialize the board driver. The current settings of the selected demo board **will not change after calling initialize**. The panel will reflect the current settings of the demo board feature which support readback after the Initialize dialog closes.

Reset - resets the PXI board settings to their default state and clears the reading.

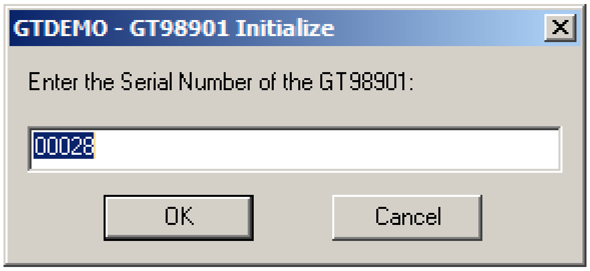
Apply – applies changed settings to the board

Close - closes the panel. Closing the panel **does not affect** the current demo board settings.

Virtual Panel Initialize Dialog

The Initialize dialog initializes the driver for the specified board. The demo board settings **will not change** after initialize is called. Once initialized, the panel will reflect the current settings of the demo board.

The Initialize dialog will prompt the user for the GT98901's serial number. This five-digit number is located on the back of the GT98901's PCB. Pressing the **OK** button will immediately attempt to connect to the board specified by the serial number entered and read its current settings. Pressing the **Cancel** button will return the user to the main panel screen without attempting a connection to a new device.



GTDEMO - GT98901 Initialize

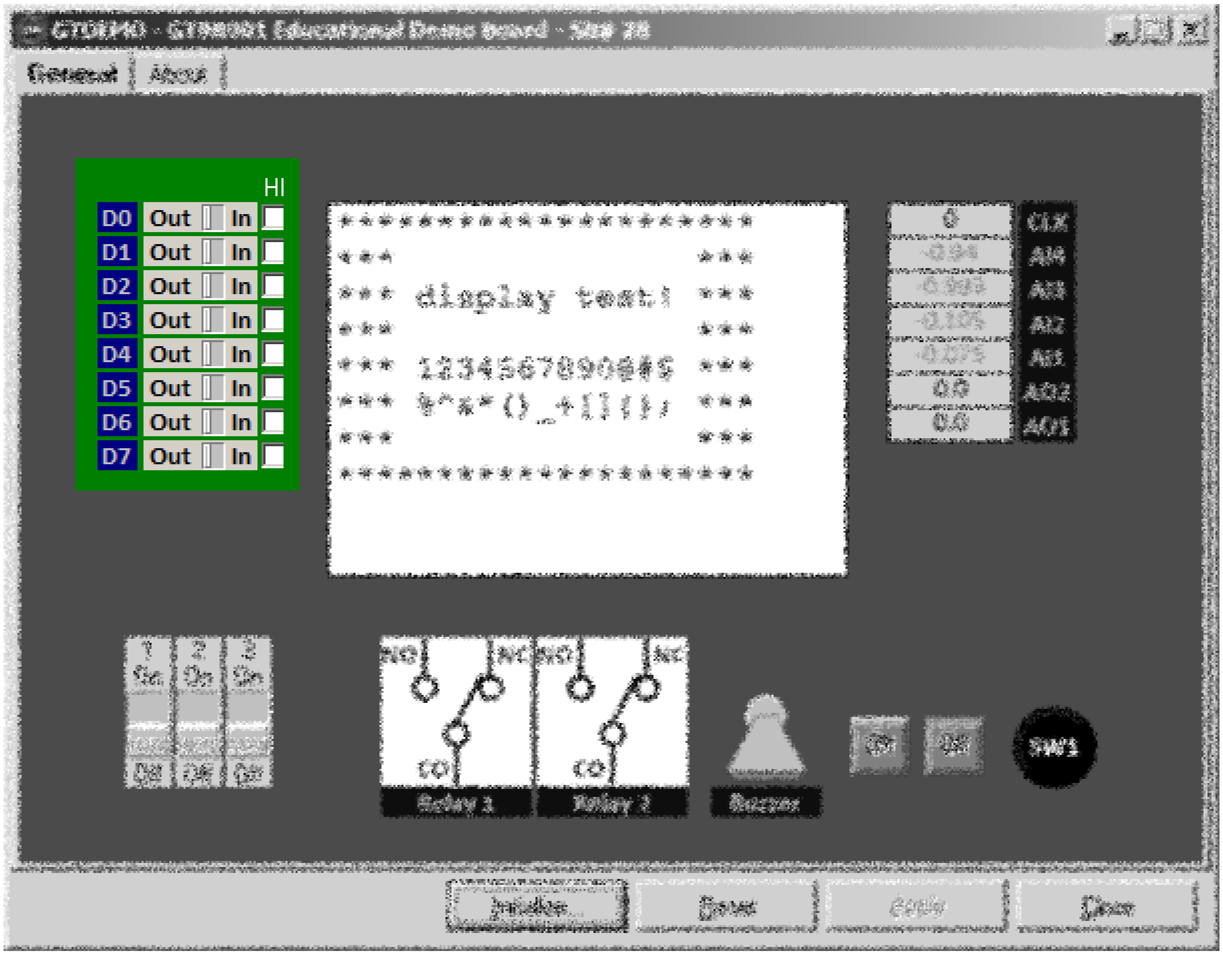
Enter the Serial Number of the GT98901:

00028

Figure 2-3: Initialize Dialog Box

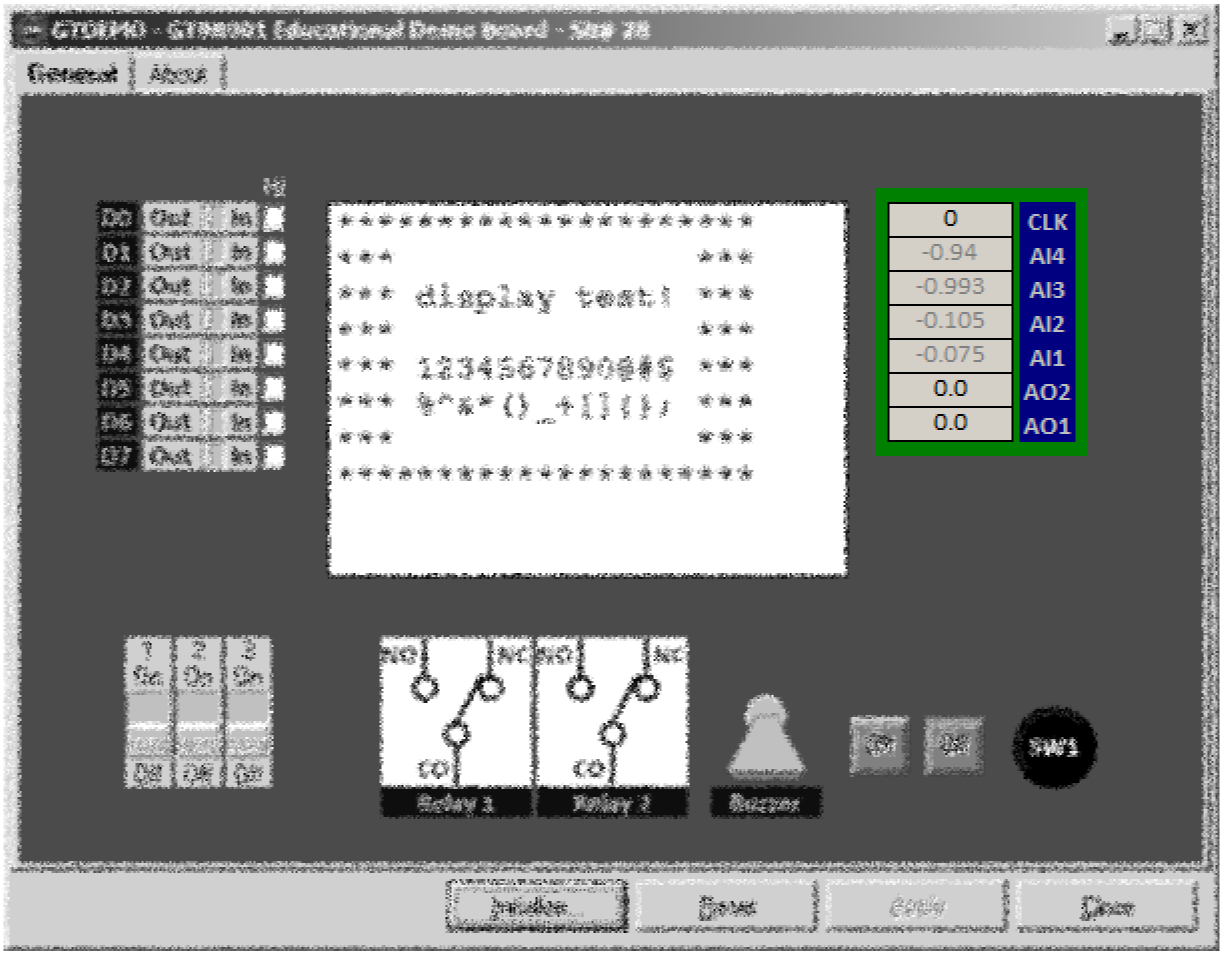
Virtual Panel Digital Functions

After initialization, you can interact with all of the GT98901's subsystems. Highlighted below are the connections that are located on the J2 (left) screw terminal block, which includes all eight digital ports:



Virtual Panel Analog Functions and Reference Clock

After initialization, you can interact with all of the GT98901's subsystems. Highlighted below are the connections that are located on the J1 (right) screw terminal block including the analog input, the analog output and the reference clock connections.



```
*********************
***               ***
*** display test! ***
***               ***
*** 1234567890@#$ ***
*** %^&*()_+[]{}; ***
***               ***
*********************
```

CLK	0
AI4	-0.94
AI3	-0.993
AI2	-0.105
AI1	-0.075
AO2	0.0
AO1	0.0

Figure 2-4: GT98901 Virtual Panel (Analog Functions Highlighted)

The following controls are highlighted:

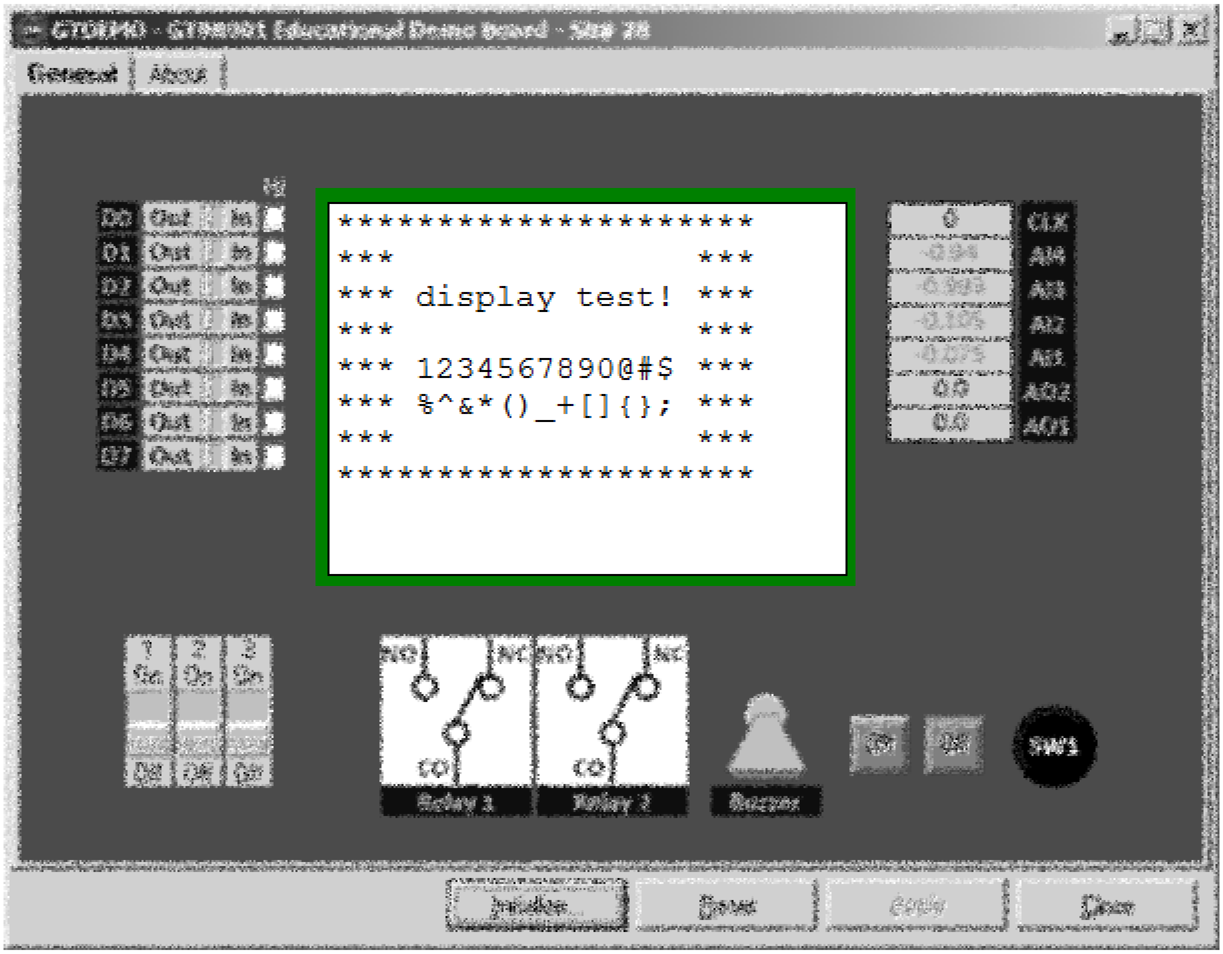
CLK: Allows you to set the speed of the reference in MHz. Allowable values are 0.0036 to 120. If the value you input is out of range, the panel will automatically set itself to the closest allowable value. After making a change to this field, the **Apply** button must be pressed to update the board.

AI1, AI2, AI3, AI4: These four button are read-only and will update automatically every half-second with the current voltage read through these analog input ports.

AO1, AO2: Allow you to set the static voltage that will be produced on the specified analog output channel. Allowable values are 0 to 10 volts. If the value is out of range, the panel will automatically set itself to the closest allowable value. After making a change to either of these fields, the **Apply** button must be pressed to update the board.

Virtual Panel LCD Display Functions

After initialization, you can interact with all of the GT98901's subsystems. Highlighted below is the LCD display portion:



```
*********************
***               ***
*** display test! ***
***               ***
*** 1234567890@#$ ***
*** %^&*()_+[]{}; ***
***               ***
*********************
```

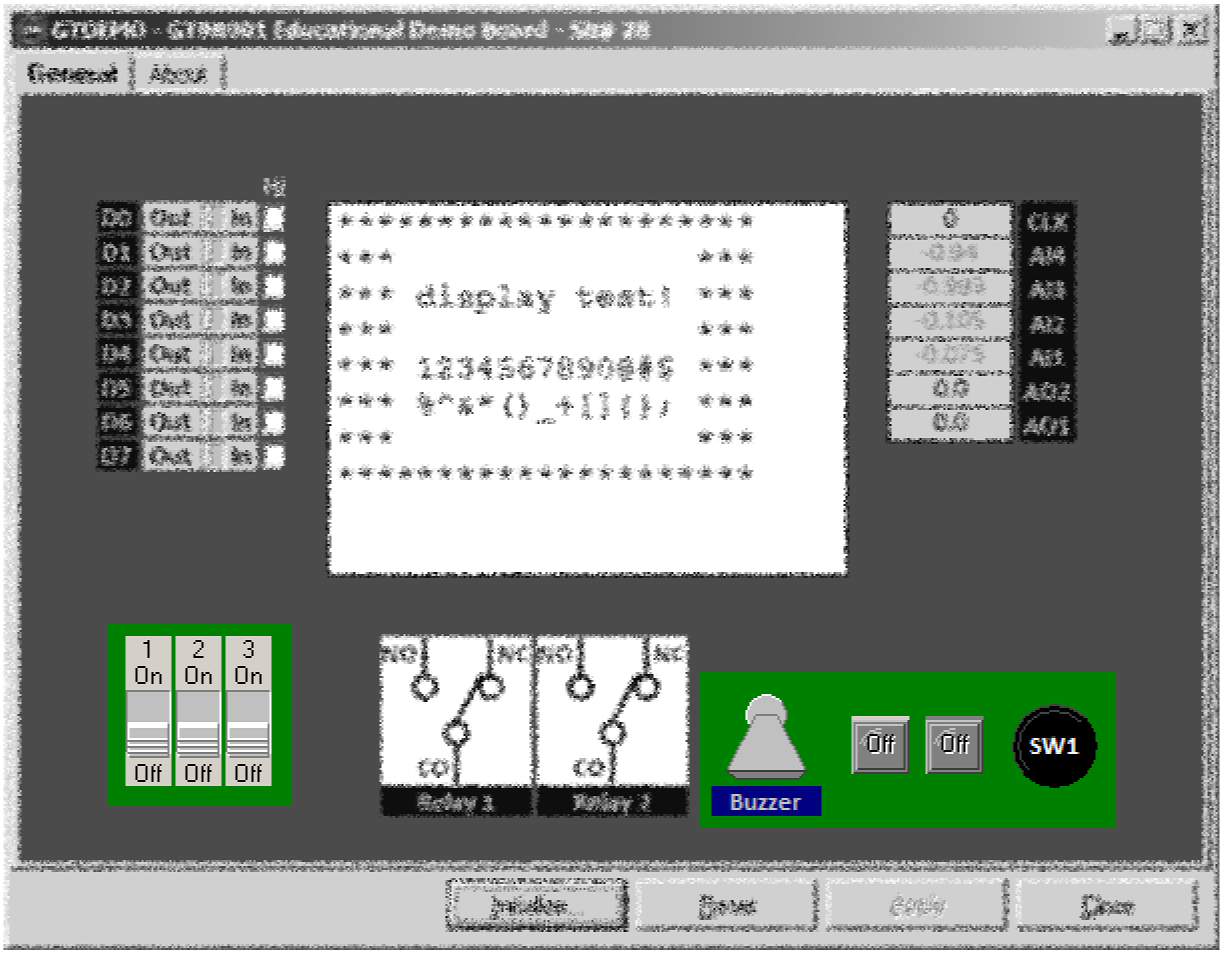
Figure 2-5: GT98901 Virtual Panel (LCD Display Highlighted)

The following controls are highlighted:

The **LCD display** on the GT98901 supports eight lines of 21 ASCII character per line. When the panel initializes, if shows every line completely filled with numbers as depicted in the Figure above. After making changes to this field, the **Apply** button must be pressed to update the board.

Virtual Panel Switch Functions

After initialization, you can interact with all of the GT98901's subsystems. Highlighted below are the switch controls:



```
***********************
***                 ***
***  display test!  ***
***                 ***
***  1234567890@#$  ***
***  %^&*()_+[]{}|  ***
***                 ***
***********************
```

Figure 2-6: GT98901 Virtual Panel (Switches Highlighted)

The following controls are highlighted:

On the left hand side, there are three dipswitch indicators. Every half-second, the state of the dipswitches is read from the microcontroller and these indicators are updated.

Buzzer is the control for the speaker. Toggling the switch will turn the speaker on/off.

The two square lamps between **Buzzer** and **SW1** are the green and orange indicator LEDs. Toggling these buttons will turn the selected LED on/off.

SW1 is the momentary push button indicator. This indicator will turn red if the push button has been depressed within the last half-second.

Virtual Panel Relay Functions

After initialization, you can interact with all of the GT98901's subsystems. Highlighted below are the relay controls, which are accessible through the J3 (bottom) screw terminal block that is below the LCD display on the physical board:

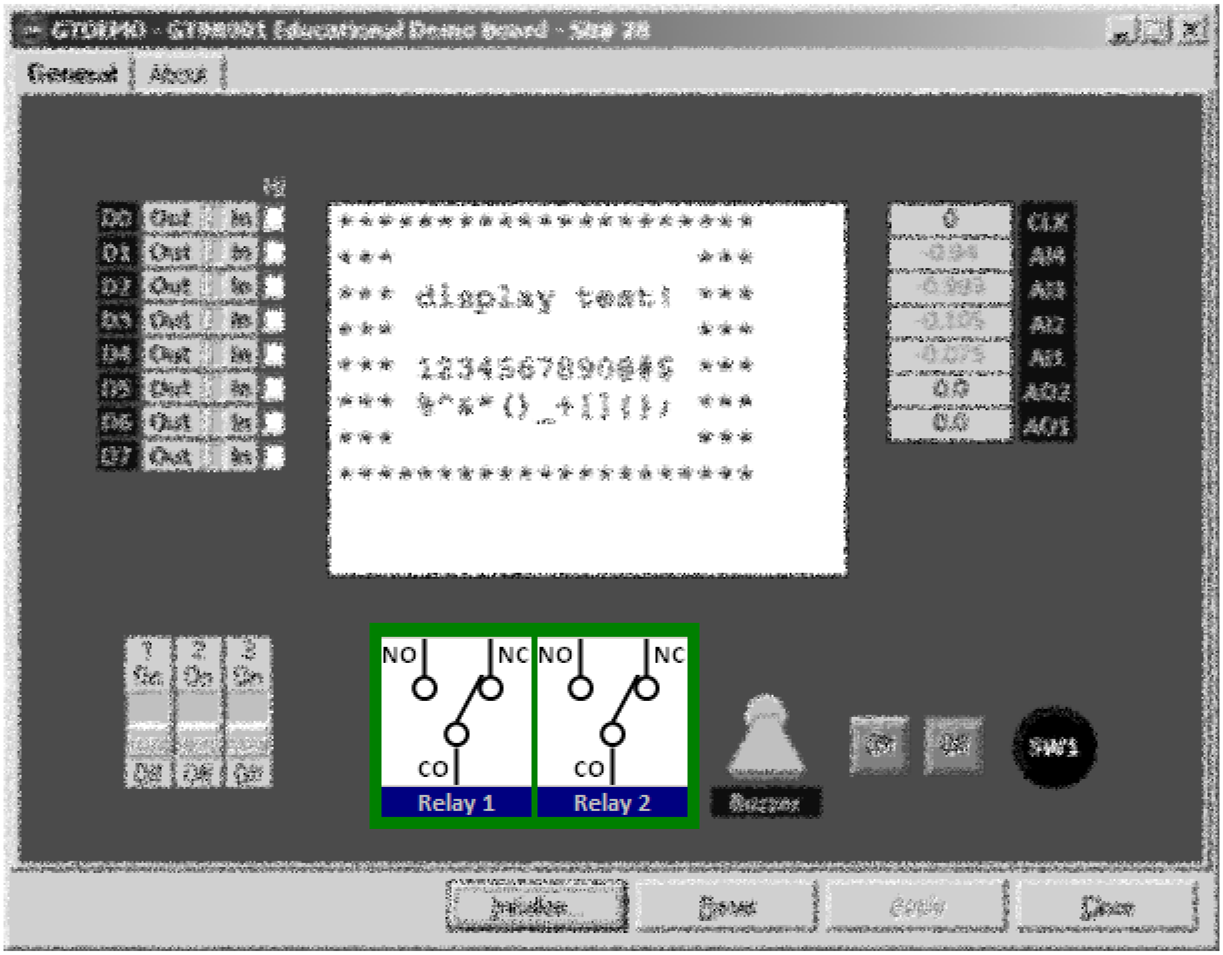


Figure 2-7: GT98901 Virtual Panel (Relays Highlighted)

The following controls are highlighed:

Relay 1 and **Relay 2** function is the same manner. The diagram above the relay label is a clickable control. Clicking it will change the position of the relay. Upon power-up, the default relay position is to create a path from CO to NC.

Chapter 3 - Installation and Connections

Getting Started

This section includes general hardware installation procedures for the GT98901 board and installation instructions for the GT98901 - GTDEMO software. Before proceeding, please refer to the appropriate chapter to become familiar with the board being installed.

To Find Information on…	**Refer to…**
GTDEMO Software Installation	This Chapter
Hardware/Board Installation	This Chapter
Function Reference	Appendix A
SCPI Function Reference	Appendix B

Packing List

All GT98901 boards have the same basic packing list, which includes:

1. GT98901 Board
2. USB to Micro-USB Interface cable
3. Wire and Accessory Kit
4. Disk with GTDEMO Software and Manual

Unpacking and Inspection

After removing the board from the shipping carton:

Caution - Static sensitive devices are present. Ground yourself to discharge static.

1. Remove the board from the static bag by handling only the metal portions.
2. Be sure to check the contents of the shipping carton to verify that all of the items found in it match the packing list.
3. Inspect the board for possible damage. If there is any sign of damage, return the board immediately. Please refer to the warranty information at the beginning of the manual.

System Requirements

The GT98901 board is designed to connect to any computer system that has a USB port. The software is compatible with any computer system running Windows XP or newer (32-bit or 64 bit) that was released prior to the release date of this software.

Installation of the GTDEMO Software

Before connectting the board to a USB port, it is recommended to install the software as described in this section:

1. Insert the Marvin Test Solutions CD-ROM and locate the **GTDEMO.EXE** setup program. If you computer's Auto Run is configured, when inserting the CD a browser will show several options, select the Marvin Test Solutions Files option, then locate the setup file. If Auto Run is not configured you can open the Windows explorer and locate the setup files (usually located under \Files\Setup folder). You can also download the file from Marvin Test Solutions web site (www.marvintest.com).
2. Run the setup and follow the instruction on the Setup screen to install the software.

   Note: You may be required to restart the setup after logging-in as a user with Administrator privileges. This is required in-order to upgrade your system with newer Windows components and to install the HW kernel-mode device drivers that are required by the GTDEMO driver to access resources on your board.
3. The first setup screen to appear is the Welcome screen. Click **Next** to continue.
4. Enter the folder where the software is to be installed. Either click **Browse** to set up a new folder, or click **Next** to accept the default entry of `C:\Program Files[ (x86)]\MarvinTestSolutions\GTDEMO`.
5. Select the type of Setup you wish and click **Next.** You can choose between **Typical**, **Run-Time** and **Custom** setups. **Typical** setup type installs all files. **Run-Time** setup type will install only the files required for controlling the board either from its driver or from its virtual panel. **Custom** setup type lets you select from the available components.

The program will now start its installation. During the installation, Setup may upgrade some of the Windows shared components and files. The Setup may ask you to reboot after it complete if some of the components it replaced where used by another application during the installation – do so before attempting to use the software.

You can now continue with the installation to install the board. After the board installation is complete, you can test your installation by starting a panel program that let you control the board interactively. The panel program can be started by selecting it from the Start, Programs, GTDEMO menu located in the Windows Taskbar.

Overview of the GTDEMO Software

Once the software installed, the following tools and software components are available:

GTDEMO Panel – use to configure, control and display the board settings.

GTDEMO driver - A DLL based function library (GTDEMO.DLL, located in the Windows System folder) used to program and control the board.

Programming files and examples – interface files and libraries for various programming tools, see later in this chapter for a complete list of files, programming languages and development tools supported by the driver.

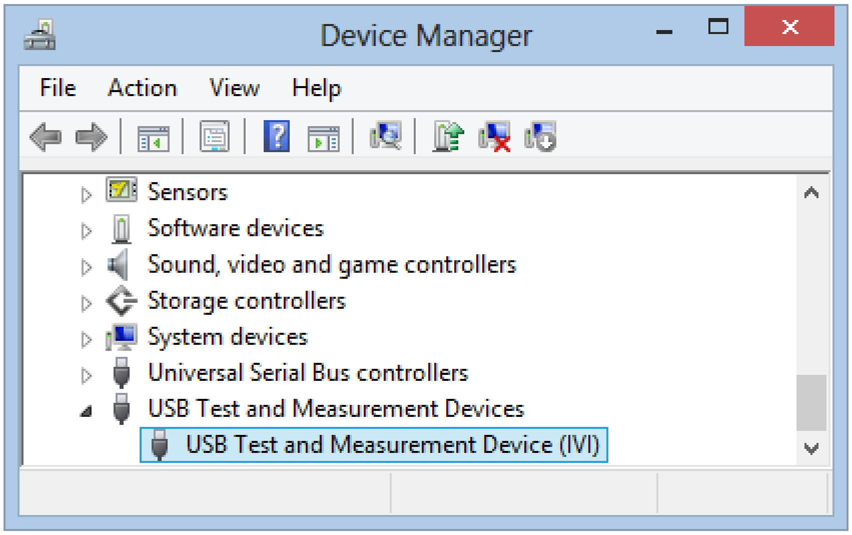
Documentation – On-Line help and User's Guide.

Plug & Play Driver Installation

Plug & Play operating systems such as Windows notifies the user that a new board was found using the **New Hardware Found** wizard after restarting the system or plugging in the new board.

The Windows Device Manager (open from the System applet from the Windows Control Panel) must display the proper board name **USB Teast and Measuremnent Device (IVI)** or **USBTMC** the Group **USB Test and Measurements Devices** before continuing to use the board software (no Yellow warning icon shown next to device). If the device is displayed with an error, you can select it, press delete, and then press F5 to rescan the system again and to start the New Hardware Found wizard.

If the device is not found or is found under a different name, then you will have to reinstall the device driver using the New Hardware Found Wizard. When it opens, select **Browse my computer for driver software** and then select **Let me pick from a list of device drivers on my computer**. This should list all potential device drivers that could fit this board. Select **USBTMC** from the list.



- ▷ Sensors
- ▷ Software devices
- ▷ Sound, video and game controllers
- ▷ Storage controllers
- ▷ System devices
- ▷ Universal Serial Bus controllers
- ◢ USB Test and Measurement Devices
  - USB Test and Measurement Device (IVI)

Figure 3-1: Device Manager Showing USBTMC Device

Connectors

The GT98901 has three screw terminals, colored blue, on the left, right and bottom of the board.

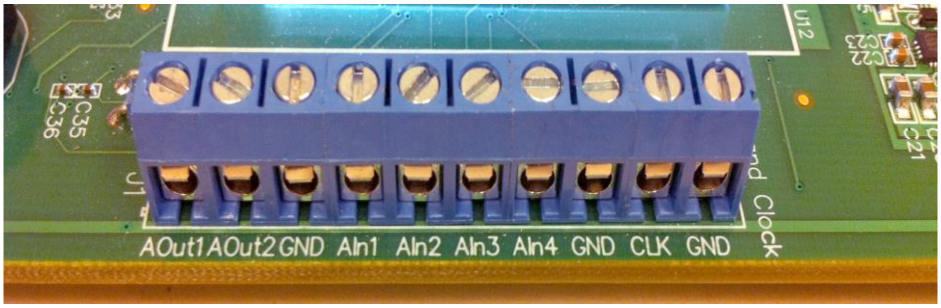


Figure 3-2: J1 Analog and Clock

Pin Designation	Signal
AOut1	Analog Output Channel 1
AOut2	Analog Output Channel 2
GND	GT98901 Common Ground
AIn1	Analog Input Channel 1
AIn2	Analog Input Channel 2
AIn3	Analog Input Channel 3
AIn4	Analog Input Channel 4
GND	GT98901 Common Ground
CLK	Programmable Reference Clock
GND	GT98901 Common Ground

Table 3-1: Left Screw Terminal Connectors

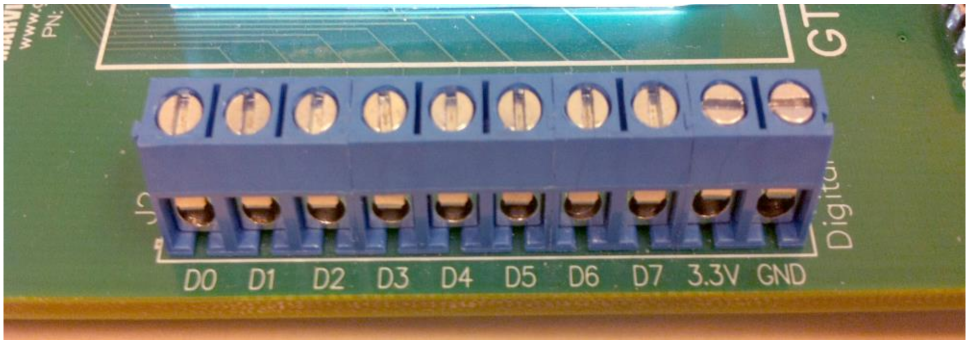


Figure 3-3: J2 Digital

Pin Designation	Signal
D0	Digital Input/Output Channel 0
D1	Digital Input/Output Channel 1
D2	Digital Input/Output Channel 2
D3	Digital Input/Output Channel 3
D4	Digital Input/Output Channel 4
D5	Digital Input/Output Channel 5
D6	Digital Input/Output Channel 6
D7	Digital Input/Output Channel 7
3.3V	3.3V Supply Voltage
GND	GT98901 Common Ground

Table 3-2: J2 Screw Terminal (Left) Connectors

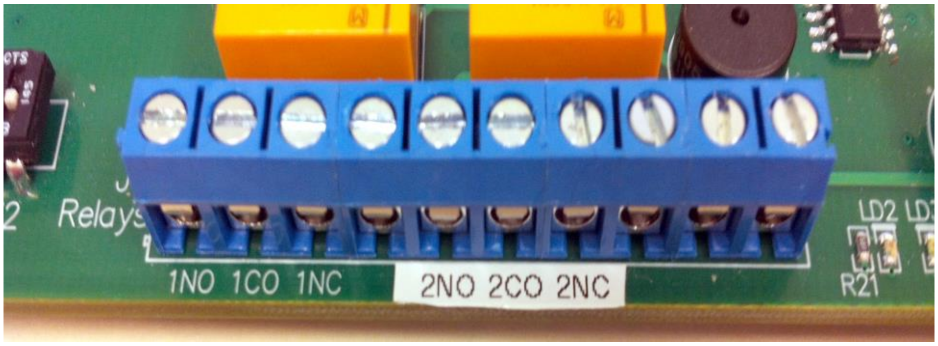


Figure 3-4: J3 Relays

Pin Designation	Signal
1NO	Relay 1's Normally Open Contact
1CO	Relay 1's Common Contact
1NC	Relay 1's Normally Closed Contact
2NO	Relay 2's Normally Open Contact
2CO	Relay 2's Common Contact
2NC	Relay 2's Normally Closed Contact

Table 3-3: J3 Screw Terminal (Bottom) Connectors

Installation Folders

The GTDEMO driver files are installed in the default directory **`C:\Program Files[ (x86)]\MarvinTestSolutions\GTDEMO`.** You can change the default `GTDEMO` directory to one of your choosing at the time of installation.

During the installation, GTDEMO Setup creates and copies files to the following directories:

Name	**Purpose / Contents**
…\MarvinTestSolutions\GTDEMO	The GTDEMO directory. Contains panel programs, programming libraries, interface files and examples, on-line help files and other documentation.
…\ MarvinTestSolutions\HW	HW device driver. Provide access to your board hardware resources such as memory, IO ports and PCI board configuration. See the README.TXT located in this directory for more information.
…\ATEasy\Drivers	ATEasy drivers directory. GTDEMO Driver and example are copied to this directory only if ***ATEasy*** is installed to your machine.
Windows System Folders	Windows System directory. Contains the GTDEMO.DLL driver HW driver shared files and some upgraded system components, such as the HTML help viewer, etc.

GTDEMO Driver Files Description

The Setup program copies the GTDEMO driver, a panel executable, the GTDEMO help file, the README.TXT file, and driver samples. The following is a brief description of each installation file:

Driver File and Virtual Panel

GTDEMO.DLL - 32 bit Windows DLLs for 32 bit applications running under Windows.

GTDEMOPANEL.EXE – An instrument front panel program for all GTDEMO supported boards.

Interface Files

The following GTDEMO interface files are used to support the various development tools:

GTDEMO.drv - ATEasy driver File for GT98901 Virtual Panel Program.

On-line Help and Manual

GTDEMOUG.PDF – Online, printable version of the GTDEMO User's Guide in Adobe Acrobat format. To view or print the file you must have the reader installed. If not, you can download the Adobe Acrobat reader (free) from http://www.adobe.com.

ReadMe File

README.TXT – Contains important last minute information not available when the manual was printed. This text file covers topics such as a list of files required for installation, additional technical notes, and corrections to the GTDEMO manuals. You can view and/or print this file using the Windows NOTEPAD.EXE or any other text file editors.

Example Exercises

The completed tutorials from Chapter 12 of this manual are available in the Exercises directory of the GTDEMO software package. All of the exercises are grouped into one ATEasy workspace that is located in the aforementioned directory.

ATEasy driver and examples:

Exercises.wsp - example workspace

Exercise*X*.sys - example system where *X* correlates to an exercise in Chapter 12

Exercise*X*.prg - example program where *X* correlates to an exercise in Chapter 12

Exercise*X*.prj - example project where *X* correlates to an exercise in Chapter 12

Setup Maintenance Program

You can run the Setup again after GTDEMO has been installed from the original disk or from the Windows Control Panel – Add Remove Programs applet. Setup will be in the Maintenance mode when running for the second time. The Maintenance window show below allows you to modify the current GTDEMO installation. The following options are available in Maintenance mode:

Modify. When you want to add or remove GTDEMO components.

Repair. When you have corrupted files and need to reinstall.

Remove. When you want to completely remove GTDEMO.

Select one of the options and click **Next**.

Follow the instruction on the screen until Setup is complete.

Chapter 4 - ATEasy Setup and Installation

About ATEasy Setup and Installation

This chapter provides information on installing and configuring ***ATEasy***. It supplies software and hardware installation information, setup requirements and options, and registration information.

The README file located on the installation disk provides the latest information about ***ATEasy***, as well as special, late breaking installation notes. Please refer to this file before running ***ATEasy***. The Windows NOTEPAD or WORDPAD programs can be used to view and print this file.

This chapter covers the following topics:

Topic	Content
Hardware & Software Requirements	Minimum and recommended requirements.
Installing ***ATEasy***	Procedure for installing ***ATEasy***.
Installation Directories	Where ***ATEasy*** places files during installation.
Installation Types	Deciding which ***ATEasy*** components to install.
Installing Hardware Interfaces	Where to get information on installing and configuring interface boards.
Windows NT Installation	Special considerations when installing ***ATEasy*** under Windows NT.
Setup Maintenance Program	Adding or removing individual ***ATEasy*** components, installing a new version of ***ATEasy*** or reinstalling ***ATEasy*** when files are corrupted, and removing ***ATEasy*** entirely.
Registration and Support	How to register and obtain technical support for ***ATEasy.***

Hardware and Software Requirements

ATEasy is a 32-bit Microsoft Windows application program designed and tested for Windows operating systems. You must have Windows installed on your computer prior to installing ***ATEasy***.

To install ***ATEasy*** you need the following minimum configuration:

32 or 64 bit Windows including Windows XP, Windows Server 2003, or 2008, Windows VISTA, Windows 7, Windows 8 or newer.

Installing *ATEasy*

You can install ***ATEasy*** by following these steps, or you can use the silent or automated Installation by using command line options - see below Silent (Automated) Setup:

▼ Follow these steps to install ***ATEasy***:

Insert the ***ATEasy*** CD-ROM disk in the CD-ROM drive. The Setup program runs automatically if your drive is set up to auto play.

If Setup does *not* run automatically, select **Run** from the Start menu and when prompted, type:

[*drive letter*]:\AExplorer

Where [*drive letter*] is the drive letter assigned to your CD-ROM drive. As an example, type "D**:\AExplorer**" when your CD-ROM is assigned to drive letter "D".

Note: Internet Explorer (IE) 6.0 or above is required to install and to use ***ATEasy***.

Note: When installing under Windows 2000, you may be required to restart the setup after logging-in as a user with Administrator privileges. This is required in order to upgrade your system with newer Windows components and to install and a kernel-mode device driver (HW.SYS) required allowing ATEasy application access hardware devices on your computer

A window showing several options will be displayed. Select ***ATEasy*** Software and select Install ***ATEasy*** to start ***ATEasy*** setup program.

The first screen to appear is the Welcome screen. Click **Next** to continue.

The next screen is the License Agreement. When you finish reading it, click **Yes** to continue (answering "No" exits the Setup program).

Enter your name and company name. Click **Next** to continue.

Enter the folder where ***ATEasy*** should be installed. Either click **Browse** to set up a new folder, or click **Next** to accept the default entry of **`C:\Program Files[ (x86)]\ATEasy`**. For more information on the Installation Directories, see page 25.

Select the type of Setup you wish and click **Next.** For more information on the types of installation available, see page 25.

Select the Program folder where the icons and shortcuts for ***ATEasy*** are to be stored. Click **Next** when finished.

The program will now start its installation. During the installation of ***ATEasy***, Setup may upgrade some of the Windows shared components and files.

If prompted, restart Windows. Setup may ask you to reboot after it complete if some of the components it replaced where used by another application during the installation – do so before attempting to run ***ATEasy***.

You can now start ***ATEasy*** by double-clicking the ***ATEasy*** icon on the desktop ATEasy , or by selecting ***ATEasy*** from the **Start**, **Programs,** *ATEasy* menu.

Installation Directories

ATEasy files are installed in the default folder **`C:\Program Files\ATEasy`** or **`C:\Program Files (x86)\ATEasy`** on 64 bit Windows. You can change the default ***ATEasy*** root folder to one of your choosing at the time of installation.

During the installation, ***ATEasy*** Setup creates and copies files to the following directories:

Name	Purpose / Contents
…\ATEasy	The ***ATEasy*** root folder. Contains the ***ATEasy*** program files required to develop ***ATEasy*** applications.
…\ATEasy\Drivers	Driver files: instrument drivers and other drivers.
…\ATEasy\Examples	Programming examples. The example file that you build throughout this guide, MyProject.prj, is also located here along with all of its associated files.
…\ATEasy\Help	Help files.
…\ATEasy\Images	Image files, including icons and bitmaps that can be used with the application toolbars, menus and buttons you create.
…\Windows\System	Windows System folder (…\Windows\System32 when running Windows 2000 or newer). Contains the ***ATEasy*** run-time system and some instruments drivers DLLs. These files are required in order to run ***ATEasy*** applications.

Installing Hardware Interfaces

The installation of interface boards (for example, GPIB or VXI) is not required for the installation and operation of ***ATEasy***. Refer to "*Chapter 8 – Drivers and Interfaces"* for additional information on installing and configuring hardware interfaces.

Installation Types

The Setup program allows you to select one of the following types of installations.

- **Compact** – uses minimal hard disk space, but includes all components required to run and develop an ***ATEasy*** application
- **Custom** – allows you to control which optional components are installed
- **Full** – installs all ***ATEasy*** components
- **License Server** – installs the ***ATEasy*** license server used to provide ***ATEasy*** licenses to other computer running on the same network.
- **Run-Time** – installs the components required to run ***ATEasy***, not including the Integrated Development Environment (IDE), which is used to develop ***ATEasy*** applications
- **Typical** – (default) installs the most commonly used ***ATEasy*** components used to develop and run an ***ATEasy*** application

Installation details are shown in the following table:

Components Installed	Compact	Custom	Full	License Server	Run-Time	Typical
Run-Time	Yes	Yes	Yes	No	Yes	Yes
Program	Yes	Yes	Yes	No	No	Yes
Help	Yes	Yes (opt)	Yes	No	No	Yes
Drivers	No	Yes (opt)	Yes	No	No	Yes
Examples	No	Yes (opt)	Yes	No	No	Yes
License Server	No	Yes (opt)	No	Yes	No	No
ATEasy 2.x Migration	No	No (opt)	Yes	No	No	No

The License Server component is used to setup a network computer to serve as a license server for other computers running that are connected to the network. See the ***ATEasy*** On-Line Books for more information.

Windows HW Device Driver Manual Installation

During ***ATEasy*** installation, a special device driver called HW, must be installed and started before ***ATEasy*** can be used. Under Windows 2000 or newer before installing the HW driver, you must be logged on as a user with administrator privileges.

The ***ATEasy*** Setup program normally installs the driver and starts it automatically. However, if the current user is not logged-in as an administrator, the driver installation fails. This section explains how to install the driver manually when the Setup program fails to do so.

▼ To manually install the kernel mode driver, perform the following:

1. Login as an administrator (applicable only for Windows 2000 or newer).
2. Open a Command prompt window.
3. Change folder to the installation destination folder for the device driver HW by using the CD command. For example:

   ```
   CD \Program Files[ (x86)]\MarvinTestSolutions\HW
   ```
4. At the command prompt, type the following command:

   ```
   HWSETUP -vdd install start
   ```

1. If the current working folder is different from the folder where the HW driver resides, you may specify your own custom path. For example:

   ```
   HWSETUP -vdd install=a: start
   ```

The Setup program installs the driver as a service. The service can be started or stopped from the Windows Device Manager, which can be opened from the Computer Management application. The -vdd switch can be removed from the command if support for 16-bit drivers is not required (only the 32 or 64 bit DLL will be used in this case).

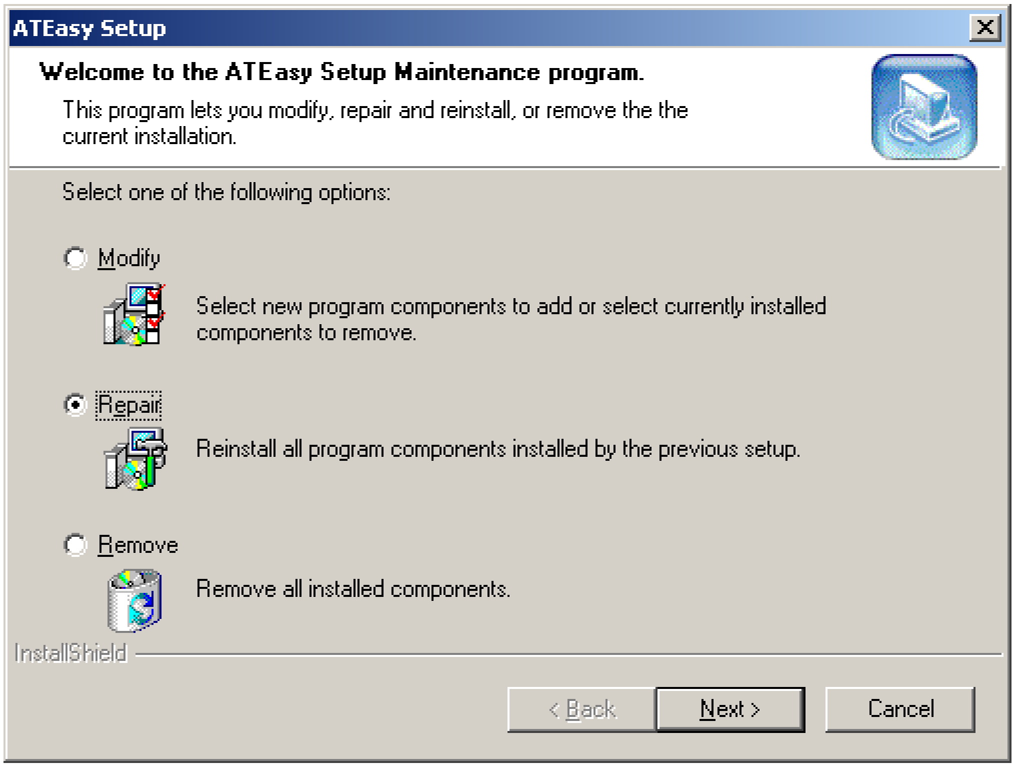
The Setup program HWSETUP.EXE, the device driver HW.SYS/HW64.SYS/HW.VXD and HWVDD.DLL files may be distributed with ***ATEasy*** applications. Additional HWSETUP.EXE command line options are available. To display these options, type **HWSETUP** without command line options.

Setup Maintenance Program

If you run Setup again after ***ATEasy*** has been installed, Setup opens in the Maintenance mode. The Setup Maintenance Program allows you to modify the current ***ATEasy*** installation. You can run Setup in Maintenance mode for the following reasons:

- When you want to add or remove ***ATEasy*** components.
- When you have corrupt files and need to reinstall.
- When you want to completely remove ***ATEasy***.

The Maintenance mode screen is shown below. Select one of the screen options and then click **Next**.



ATEasy Setup

Welcome to the ATEasy Setup Maintenance program.

This program lets you modify, repair and reinstall, or remove the the current installation.

Select one of the following options:

○ Modify

Select new program components to add or select currently installed components to remove.

● Repair

Reinstall all program components installed by the previous setup.

○ Remove

Remove all installed components.

The Maintenance Mode screen options are described further below:

Maintenance Option	Description
Modify	Use to add or remove individual ***ATEasy*** software components.
Repair	Use to reinstall ***ATEasy*** when you have corrupt files or to upgrade and install a new version of ***ATEasy***. Repair refreshes and recopies current files that are corrupt and upgrades the files if necessary.
Remove	Use to completely remove ***ATEasy*** and all its components. Also removes ***ATEasy*** from the Windows Registry and the Startup menu.

License, Registration, and Support

To use ***ATEasy*** you must purchase a license from Marvin Test Solutions. Three types of licenses are available:

Single License

Network License

Hardware Key (USB or LPT versions)

If you do not have a license, you can activate a 30 days trial version of the ***ATEasy*** software. The trial license contains full ***ATEasy*** functionality for 30 days. You are allowed one 30 days trial period on the computer on which you install ***ATEasy***.

A license can be set up from the ***ATEasy*** **License Setup** dialog box. This dialog is displayed either when starting ***ATEasy*** when no license is installed or from the **About *ATEasy*** menu item under the **Help** menu used when you want to change the license.

Users who purchased a subscription plan must register to activate the plan. A subscription plan entitles you to receive free upgrades and unlimited customer support. If you did not purchase the subscription plan, you may register as well to receive free ***ATEasy*** newsletter, product service packs, updated drivers and examples.

See the ***ATEasy*** on-line books for more information about how to register the product and setup a license.

Our Web site (www.marvintest.com) contains sections describing support options, application notes and knowledge base articles, downloading upgrades, examples, instrument drivers, and submitting support questions for ***ATEasy*** registered users

.

Chapter 5 - Overview of ATEasy

About the Overview

This chapter provides general information regarding ***ATEasy***. It provides an overview of ***ATEasy*** including Automated Test Equipment (ATE) test systems, the structure of ***ATEasy***, the development process, and introduces the ***ATEasy*** Integrated Development Environment (IDE).

Use the table below to learn more about overview topics:

Topic	Description
What is ***ATEasy***?	Provides an overview of ***ATEasy***.
Automated Test System	Explains the automated test system modules.
Workspace, Applications and Modules	Introduces the component parts of an ***ATEasy*** application.
The Project	Explains the structure of an ***ATEasy*** project.
Sub modules	Describes the sub modules serving as containers for objects such as forms and procedures.
The Program Module	Describes the Program module and the program tests.
Tasks and Tests	Describes the program Tests submodule and program Tasks and Tests.
The System Module	Describes the System module.
Commands	Describes Commands statements.
Driver Module	Describes the Driver module.
The Integrated Development Environment	Introduces the ***ATEasy*** Integrated Development Environment (IDE) used to develop ***ATEasy*** applications.

What is *ATEasy*?

ATEasy is a test executive and a software development environment for Test and Measurements (T&M) applications. It contains all the tools required to develop test applications for Automated Test Equipment (ATE) systems and for instrument control applications. The purpose of the ATE system is to perform testing on one or more electronic products called Units Under Test (UUTs) such as components, boards, assemblies, etc. A typical ATE system consists of a computer/controller, several test and measurement instruments and a test application designed to control the system instruments in order to test the UUT.

Running under Microsoft Windows XP or newer, ***ATEasy*** provides a familiar graphical user interface (GUI) combined with the flexibility of an object oriented programming environment. Users of Microsoft Visual Basic or Visual C++ will feel right at home.

Supporting any instrument, regardless of its interface, ***ATEasy*** develops an ATE application in a single integrated environment. With specialized features designed for testing and instrument control applications, ***ATEasy*** can also be used for data acquisition, process control, lab applications, calibration, and for any application requiring instrument control. ***ATEasy*** supports many instrument interfaces including VXI, GPIB (IEEE-488), RS-232/422, PC boards, PXI, and LXI (TCP/IP).

ATEasy's Integrated Development Environment (IDE) is object oriented and data-driven. Editing tools are automatically selected by ***ATEasy*** according to the type of object to be created or modified. This feature simplifies programming, as you merely click on an object and ***ATEasy*** automatically selects the appropriate tool.

ATEasy's IDE includes tools for creating instrument drivers, user interface, tests, documentation, test executives, report generation and anything else you need to create T&M applications – all with point and click and drag and drop ease.

ATEasy contains a high-level programming language enabling test engineers, electronics engineers, and programmers to develop and integrate applications of any scale – small to large, simple to complex. The ***ATEasy*** programming language allows user-defined statements to be used along with flow control, procedures, variables, and other common items found in most programming languages. The ***ATEasy*** programming language is flexible and powerful, yet easy-to-use and self-documenting.

Professional programmers will appreciate ***ATEasy***'s programming language offering DLL calling, C header file importing for DLL functions prototype, OLE/COM/ActiveX controls support, .NET Assemblies, LabView® VIs (Virtual Instruments) or their libraries (LLB), function panel instrument driver files (used mostly by LabWindows/CVI®, multi-threading, exception handling and many more software components and standards for developing complex applications in a truly open system architecture. ***ATEasy***'s programming language also contains many built-in programming elements to simplify programming, allowing a non-programmer to easily use ***ATEasy*** to develop an application.

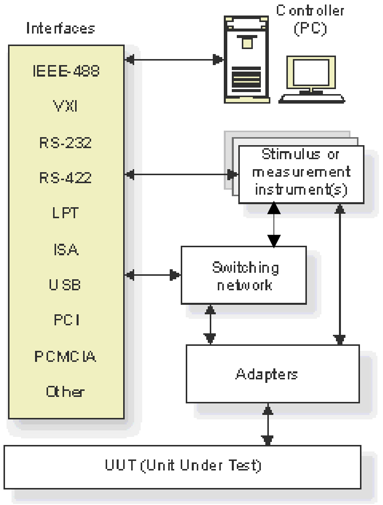
The unique design of ***ATEasy*** provides a structured and integrated framework for developing reusable components and modules you can easily maintain and debug. These components can be reused from application to application reducing the time and effort of developing new, and maintaining existing, applications. The developer is given a framework especially designed especially for a T&M application. The framework contains pre-defined components designed for interfaces (such as GPIB), instruments control and drivers, system configuration, test requirement documents and test executives.

In addition, the ***ATEasy*** IDE provides a Rapid Application Development (RAD) environment. This provides a way to write, run and debug applications in very short cycles as required by instrument-based applications. The ***ATEasy*** IDE is an object-oriented environment, making the editing of common tasks or objects displayed in the IDE very similar to other object-oriented environments. The similar functionality greatly reduces the learning curve for ***ATEasy***.

With ***ATEasy***, multiple users can edit the same file representing a driver system or a program. Files contain version information that allows keeping track of, and documenting, the changes. In addition, all ***ATEasy*** documents can be saved to a text format allowing comparing and merging of changes between multiple users and tracking changes using version control software in a better way.

Automated Test System

An Automated Test System, also referred to as Automated Test Equipment (ATE), is a collection of instruments under computer control performing automated test functions.



The diagram shows a typical configuration of an ATE system. A computer provides control over test and measurement instruments by using hardware interfaces. The instruments, such as measurement, stimulus, switching, power and digital are connected to a Unit Under Test (UUT) through an adapter.

The most common computer used in ATE applications is the PC. Due to its relatively low cost, computing power, and the availability of hardware interfaces and computer programs, the PC has become the de-facto standard of the test industry.

The PC supports numerous methods called interfaces for controlling test instruments. These interfaces include IEEE-488 (GPIB), VXI, ISA bus, PXI/PCI Bus, LXI/TCP-IP, serial communication such as RS-232/422/485 and more. Software programs such as ***ATEasy*** allow the computer to control test instruments using any of these interfaces.

Test instruments include:

Measurement – instruments measuring electrical characteristics

Stimulus – instruments generating electronic signals

Digital – instruments that read and write digital patterns

Power – instruments using power sources

Switching – instruments routing electrical signals to different points

The *adapter*, also referred to as Interface Test Adapter (ITA), routes the signals from the test system to the Unit Under Test (UUT), which is the target of the ATE.

Under software control, the computer performs test sequences and procedures used to determine if the UUT is performing according to its specifications. Controlling the test instruments, routing signals to various test points in the UUT, and measuring UUT responses achieve this performance determination. ***ATEasy*** provides all the tools required during the development, debugging and integration of test sequences and procedures.

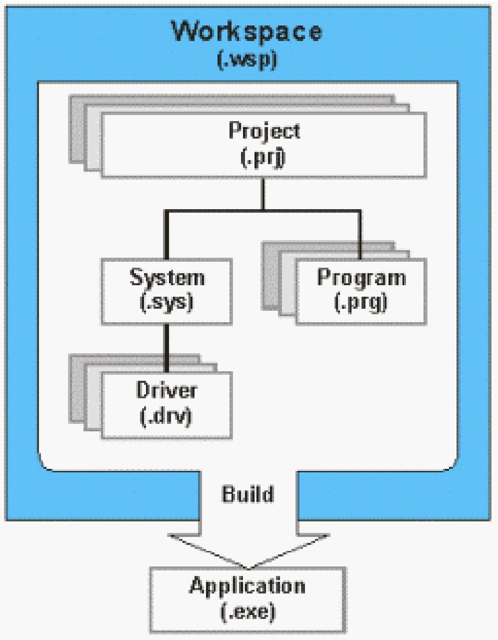
Workspace, Applications and Modules

An ***ATEasy*** application is developed in the Integrated Development Environment (IDE) within a **Workspace** file. A Workspace file is a container holding the programming environment and the last saved layout of the IDE. The Workspace itself is not a part of the application.

ATEasy applications are Windows executable files created from project files containing one or more modules. A typical project file contains a System, one or more Program(s), and one or more Driver(s). The System, Program, and Driver are called ***ATEasy*** modules. Each module contains sub modules, such as Forms, Commands, Procedures and more. Each module is stored in a project file, which may be inserted or moved between projects so it can be reused by any other ***ATEasy*** application.

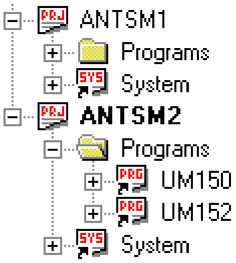
The diagram shows a Workspace, its Project file, Program, System and Driver modules.

The Workspace file and its image as it appears in the IDE contain a list of files or documents and the state of the IDE windows and their content. Only one workspace can be loaded by the IDE at a time. Typically, the workspace file contains a list of one or more projects files loaded by the IDE.



The Project

The Project file contains a list of related module files, called **modules shortcuts**, required to develop and generate an application. The Project becomes an application when it is compiled or built – creating an executable (.EXE) file.



As shown in the diagram, two projects are displayed. The Project ANTSM2 appears in bold to indicate it is the **Active Project**. It contains one **System Shortcut** associated with the hardware configuration of a given test system. In addition, the project also contains **Program Shortcuts**, UM150 and UM152, each associated with a UUT. When a project contains multiple programs, you can select the first program to run. Other programs can be run using the Run statement invoked from the application code.

Only one Project in a workspace can be active. When building, debugging, or running test programs, only the current active project will be used. When you use the **Build** or **Run** commands from the IDE menu, it will build or run only the active project. Once the project is built, the active project is compiled and the result is an executable file that can be run independently of the IDE similar to any Microsoft Windows application.

The relationships among a Project, the System module, and an Application are as follows:

A Project may contain one System module and multiple Program modules.

A Project must have a System, or a Program, or both.

A System may contain one Driver, several Drivers or none.

An Application can be built from a Project that contains Program, or a System, or both.

Submodules

Modules (System, Program, or Driver) contain submodules serving as containers for objects such as forms and procedures. Most submodules are common to all modules and may be used by the system, program, or driver modules. The table below lists and describes the available submodules and their icons as they appear in the ***ATEasy*** development environment:

Submodule	Description
Forms	A *Form* is a window or dialog box comprising part of an application's user interface. Common use of forms is for a virtual instrument's panel and is used to display its status or control its settings. Other uses include a window allowing a user interface to control the test application, save test results, and perform other user interaction. A form can contain a menu bar, toolbar, status bar, and controls. Forms also contain code used to respond to events caused as a result of user action (for example, an OnClick event is called when the user clicks on the control). The ***ATEasy*** internal library contains a large number of ActiveX controls used to display and accept data (for example, AChart control). In addition, you may use third party ActiveX controls.
Commands	*Commands* are user-defined statements extending the ***ATEasy*** programming language. Commands can be associated with procedures. In Drivers, commands can also be associated with an I/O Table used to send or receive data from an instrument.
Procedures	A *Procedure* contains code that can be called by other procedures or test code to perform an action. Procedures allow the code to be modular and re-useable. Procedures have a name used for calls, parameters to get and set data to or from the procedure, and code, which is programming statements used to perform certain actions at run-time.
Events	The *Events* submodule contains pre-defined procedures you can fill in. ***ATEasy*** calls these procedures when an event occurs. Some events are called at initialization and others at the end of a program, system, driver, task, or test. Events are typically used to change the flow control of the application and to customize the test results log. Other events are called when an error or abort occurs, thereby allowing the programmer to decide at run-time what to do upon the occurrence of these events.
Variables	*Variables* are used for storing values. Variables have a name and a type. Types can be one of the ***ATEasy*** basic types including Char, Word, Long, Double, String, Object, Variant, (and more) or any user-defined type such as Structure or Enum. A variable can also be defined as an Array (group of many variables of the same type under one variable), as Public (allowing other modules to use it), or as Constant (cannot change by code). A variable may have initial value.
Types	The *Types* submodule holds user-defined types for a module: Structure, Enum, and Typedef. **Structure** contains fields possessing different types. Structure allows the programmer to group different data typesXXX under one variable. **Enum** contains named integer constants and **Typedef** is used to alias to a different type.
Libraries	A *Library* is an external module containing procedures, classes and other programming elements. ***ATEasy*** can use three kinds of libraries: **Dynamic Link Libraries** (DLLs), which is a file typically with .DLL file extensions containing procedures. **Type Libraries**, which contain classes, procedures, and other programming elements and is based on Microsoft component technology (COM). Type libraries allow you to make use of classes exposed by external libraries or application. Examples of type libraries are ActiveX controls or MS-Excel. .NET assemblies, which contain classes, procedures, and other programming elements and is based on Microsoft .NET component technology.

Submodule	Description
	Unlike DLL where you are required to define (manually or import a C/C++ header file (.h)) the programming elements included in it, a type library or a .NET assembly contains a complete definition of the programming elements exported by the library.
IO Tables	An *I/O Table* is a table of commands to create, send, receive, and decode messages to or from an instrument. I/O Tables can be used to control instruments or processes via message-oriented interfaces such as GPIB, VXI, RS-232, WinSock, and more. Only drivers have this submodule.
Tests	The *Tests* submodule contains a collection of tests, usually grouped under tasks, used to test the UUT. The *Test* contains the code and the requirements of the test. The *task* is a way to group several tests and arrange them in logical order. At run-time when the program runs, each test generates a status: **Pass**, **Fail**, **None**, or **Error**. Only programs have this submodule.
Drivers	*Drivers* is a submodule within a system module containing file shortcuts to all drivers used by the system. The **Driver Shortcut** contains the driver filename, its name, and the driver configuration (for example, interface type or address) as used by the system. The driver itself, when used by the system or program in the project, is typically used to control an instrument by sending and receiving data to/from its interface, such as a GPIB interface.
Misc	*Misc* is a submodule within a project, program, system, and driver module. The Misc subfolder is created by the user and contains sub folders and shortcuts to external files. The Misc folder is used is to store baggage files for your projects or modules such as documentation, dlls, software components and others. You can open /edit/print these files directly from ***ATEasy*** which uses the external Windows application that is associated with the file. For example Microsoft Word will be used to open files with .doc file extensions.

The Program Module

ATEasy test programs are modules containing the necessary tests required to test a Unit Under Test (UUT). Programs follow the guidelines of the Test Requirements Documents (TRD) and therefore are divided into **Tasks** and **Tests**. Program tests and procedures can use the procedures, variables, and other submodules defined in the program.

The program resides under the project **Programs** submodule. Multiple levels of Programs folders can be created in order to organize your project test programs into categories and different UUTs . The program can use **public** symbols, such as procedures or variables, defined in the project system or drivers. Many programs can reside under the same project. The project contains the first program to run. After the program runs, the application can schedule another program to run by using the **Run** statement. Only one program can run at a time. If no program is called, then the application terminates. Every time a program runs, its variables are reset and initialized.

Tasks and Tests

ATEasy allows you to organize a program into Tasks and Tests. A **Task** consists of a group of **Tests** testing the same block or logical unit in the Unit Under Test (UUT). Each Test measures some portion of the UUT and determines if the measurement passed or failed. The results and status measured in the program Tests are normally printed to a Test log which can serve as a Test report.

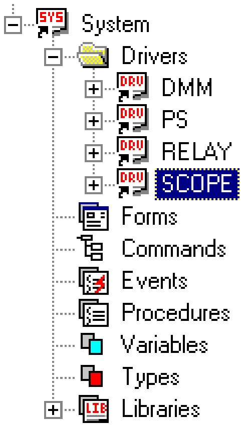
A **Test** contains programming statements (code) generating a single result (typically a measurement), or a status: **Pass, Fail, Error** or **None**. According to the type of Test, ***ATEasy*** can determine at runtime if the measurement is of acceptable value and generate a status. As an example, a Test may contain code applying input power to the UUT and then measuring the current drawn by the UUT.

The Task status is the status of its Tests. For example, if one of the Tests fails in a Task, and another passes under the same Test, then the Task status is Fail. The Task status allows you to immediately determine if the specific UUT block or function is performing as required or if it failed.

Using Tasks and Tests, you can design a test program to match a Test Requirement Document (TRD). ***ATEasy*** is quite flexible when the structure of a Test program needs to be organized after the program is created. Tasks and Tests can be moved, renamed, duplicated, and deleted by a click of a mouse. In addition, during run-time, Tasks and Tests can be called and executed in any order you or the application requires.

The System Module

The **System** is a module containing the hardware configuration of a given test system as well as commands associated with the unique configuration of that system. The System reflects the currently installed instruments and their configurations, such as the instrument interface (for example, GPIB) and its address.



A System may contain zero, one, or more **driver shortcuts** residing under the System's **Drivers** submodule.

The Driver shortcuts contain the name of the Driver (for example, DMM), the Driver file name, and configuration properties such as the device address of its interface.

The System contains sub modules such as Procedures and Variables, which can be used by the project programs if marked as **public**. Unlike Program variables, System variables retain their values throughout the life of the application.

Commands

A **command** is a user-defined statement calling an attached procedure or I/O table. The command statement is used in a test or procedure within a Program, System, or Driver module. Commands offer several advantages:

Once defined, commands appear in cascading menus on the ATEasy Menu bar allowing the user to insert them into the code by selecting them from the menu. The cascading menus organize them into logical groups, such as setup, measurement, and more. This allows the user to locate them faster and eliminates the syntax errors that can appear if you enter commands manually.

Commands simplify programming because you can substitute easy-to-understand, English statements for cryptic procedure names and parameters making the test code more readable and eliminate additional documentation.

Commands can be device-independent. If you code using commands, you can change the driver without having to recode or rewrite your code.

Command examples are shown below:

```
DMM Set Function VAC
MUX Connect BusA (1)
Delay(100)
DMM Measure (TestResult)
```

Commands can be defined under the Program, System, or Driver Commands submodules. The Commands submodule contains the programming statements you designed and created.

System module commands are used to operate the system instruments and reflect the system’s wiring and switching networks. These commands can be called from the project’s programs tests or from the system procedures. In the example below, the command measures volts DC using a DMM (Digital Multimeter) between the unit under test (UUT) points P1 and P13:

```
System Measure VDC P1_P13 (d)
```

This System command can be associated with a System procedure using more than one system instrument (DMM and a SWITCH) as shown here:

```
SWITCH Close (13)
DMM Set Function VDC
DMM Measure (dResult)
SWITCH Open (13)
```

As described, a single System command statement replaced four driver commands resulting in a simpler, modular program in the test using that command.

Driver Module

An ***ATEasy*** **Driver** is a plug-in, reusable module that can export any of its submodules to any other module (Programs, System, and other Drivers) in the project. The Driver is generally used to communicate with the "outside world" such as instruments or other devices. In a project, a driver resides under the **System Drivers** submodule.

When defining a driver, you select the interfaces (for example, GPIB or VXI) the driver supports and their default configuration (such as timeout, terminator and more). Once the driver is inserted into the system, a **driver shortcut** is created. You then select the interface used, and set other configuration attributes such as address.

Unlike a program, the Driver variables retain their values during the life of the application even after a program has finished. For example, if a program invoked a Driver's virtual DMM panel, the virtual panel would still be available after the Program is finished.

Driver commands are high-level statements similar to the statements commonly found in Test Requirement Documents (TRDs). The Driver commands, typically the code used by programs tests, are independent of the actual instrument or the method used to communicate with the instrument (for example, GPIB). A major advantage of this architecture allows ***ATEasy*** users to replace instruments and drivers without the need to modify any of the test program code. This holds true for any instrument or Driver, as long as the Driver commands are designed in the same way for all instruments of the same type.

Like any module, the Driver also contains the Forms submodule. Forms may be used interactively to create virtual panels of the instruments, allowing you to control the instrument interactively without programming.

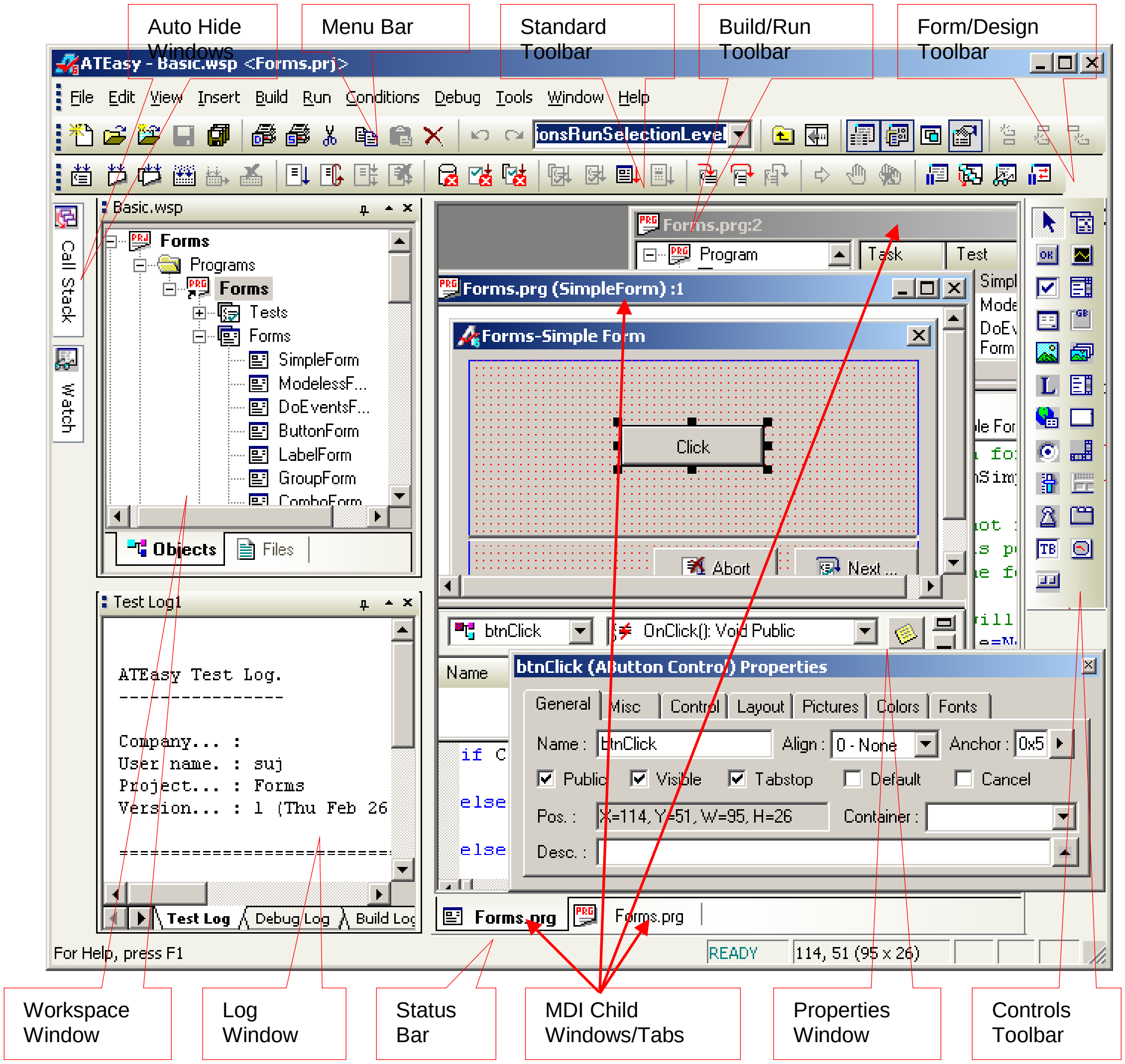
An ***ATEasy*** Driver is a reusable module. Any libraries or programming elements declared as public and used within the Driver are available to all other modules within the project by merely referencing them. The advantage is code duplication is avoided and code reuse is encouraged.

ATEasy drivers can be created by filling the driver sub modules (e.g. inserting an external library and creating driver commands) or by importing a function panel (.fp) driver file that is usually used when programming in LabWindows/CVi®® environment.

The Integrated Development Environment

Developing ***ATEasy*** applications is done using the ***ATEasy*** Integrated Development Environment (IDE). The IDE contains all the tools required to create an application, run, debug, and then build in order to create Windows' executable files.

The following figure shows the main window of the IDE below with callouts to the individual windows.



```
ATEasy Test Log.
----------------

Company... : suj
User name. : suj
Project... : Forms
Version... : 1 (Thu Feb 26

==========================
```

```
if C
else
else
```

Name : btnClick Align : 0 - None Anchor : 0x5

☑ Public ☑ Visible ☑ Tabstop ☐ Default ☐ Cancel

Pos. : X=114, Y=51, W=95, H=26 Container :

Desc. :

For Help, press F1 READY 114, 51 (95 x 26)

The following windows are displayed:

Menu Bar – contains the IDE menus including:

- **File** commands – used for file operation commands such as: Open, Save, and Print. Also used for Microsoft Source Safe connectivity such as Check-In/Out and more.
- **Edit** commands – used for editing operations such as: Undo, Redo, Cut, Copy, Paste, Delete, Find, and Replace.
- **View** commands – used for changing the way you view documents, and to show or hide the workspace built-in windows such as: Workspace, Variables, Properties, Log, and debug windows such as Call Stack/Locals, Watch, Debug, and Monitor. The built-in windows are dock-able, that is, they can be docked to either side of the main window making them always visible.
- **Insert** commands – Used to insert code commands and statements to the code editor and to insert object such as variables, procedures, forms, and more.
- **Build** commands – used to perform syntax checking and to build the application to an executable file.
- **Run** commands – used to start, abort or pause the application, and to perform test level debugging commands such as: Loop On Test and Stop On Failure.
- **Debug** commands – used to perform source level debugging commands such as Step Into, Step Over, Toggle Breakpoint and to debug small portions of your application with commands such as Doit! and Taskit!
- **Tools** commands – used to customize the IDE keyboards commands, menus and toolbars, to set the IDE options such as directories, and to manage users, password, and access rights.
- **Help** commands – provide commands to search and open the on-line help.

The IDE's most common menu commands can also be displayed using the context menu, invoked by using the right mouse button or by using fully customizable keyboard shortcuts.

Toolbars – including the Standard toolbar used for common commands, the Build/Run toolbar used for common build and run commands, the Form Design toolbar used for form layout operations, and the Controls toolbar used for inserting controls to a form.

Status bar – contains multiples panes displaying the status of the application when running (for example, Run or Paused) or other editing properties such as: current line and column, size of the selected control on a form, and more.

The Workspace window – displays the content of the current workspace file in a tree-like view. This window contains two tabs: Objects and Files. The Objects tree view displays all files objects opened by the IDE: project files, modules such as drivers, system or programs, and their submodules such as procedures, and variables. The Files Tab is used to displays the current workspace project and module files without showing the files submodules. The user can perform editing commands on the objects displayed in the tree. Double-clicking on an object/file opens the document view used to display and edit the object/file.

The Properties window – displays the properties of the currently selected object. Clicking on an object in any of the ATEasy windows can set the currently selected object. The properties window contains pages, each of which displays a partial list of the object properties. The user may change the object properties by changing the values displayed in this window.

The Log window – contains three log pages. The Test log displays the test results when the application is running, the Build log displays the build progress and compiler errors, and the Debug log displays trace statement output when the application is running. The test log displays a report automatically generated by

ATEasy when a test program is running. It can display the results in Text format or HTML format, which provides more formatting options.

Debug windows – used to display debugging information about the running application. Includes the following windows:

- **Call Stack/Locals** – displays the variables values of modules variables and procedures variables. When the application pauses, the user can change the values of variables.
- **Watch** – displays expressions, local or global variables that the user input to this window. When the application pauses, the expression is evaluated and its value is displayed in the window.
- **Debug** – used to type programming statements. When the application pauses, the user can run the code to evaluate expressions and to perform certain operations at run-time for debugging purposes.
- **Monitor** – used to display communication data between the application and a device (for example, an instrument) through an ATEasy interface (for example, GPIB).

MDI Child Window showing a Document View window – displays a module and its objects. The window is divided to two panes by a vertical splitter. The optional left pane (not shown here) displays a tree view containing the module submodules and objects. The right pane displays the object being edited, (in this example the SimpleForm is shown), which is selected in the right pane or from the workspace window. Clicking on an object in the tree view causes it to be displayed in the object view. Clicking on the MDI Child Tab can activate MDI child windows.

Auto Hide Windows – the Log, Workspace and Debug windows can be displayed in several display modes: Float displayed a stand alone window anywhere on the desktop, MDI child displayed in the main window as a document view or Docked to the sides of the main window. Changing the display mode for these windows can be done by right click on the caption. When a window docked it can also be set to Auto Hide preserving the main window space by hiding when not needed and displayed when the mouse cursor is above the window. Changing the Auto Hide can be done by clicking on the pin image on the caption of these windows. Docked window can also be expand or collapse by clicking on the down or up arrow on their caption.

Once used, you will find that the IDE is consistent and object-oriented and is geared for rapid application development. This provides you with a tool that is fast, intuitive, and easy-to-use in order to create ***ATEasy*** applications.

Chapter 6 - Your First Project

About Your First Project

This chapter discusses how to create a test application with one program using ***ATEasy's*** Application Wizard. After creating the application, you will learn how to create tests in the program and then how to build, run, and debug the application. You will also add a user interface to the application allowing the user to control the test application with the Test Executive driver supplied with ***ATEasy***. Finally, you will add in the GT98901 and begin making simple measurements with it. Use the table below to learn more about this chapter's topics:

Topic	Description
Starting ***ATEasy***	How to start ***ATEasy***
Application Types	What are the types of ***ATEasy*** projects
Creating an Application	How to create your first application using the Application Wizard
More About the IDE	Key concepts of the ***ATEasy*** IDE
Your First Test Program	How to add tasks and tests to your application
Test Properties	What are the properties of ***ATEasy*** tests
TestStatus and TestResult	Learn about these important internal variables
Running Your First Application	An overview of running an application for the first time
The Log Window	The Log Window and the information it provides
Adding the Test Executive Driver	Demonstrates a way to add user interface to your application that lets the user control the running of test programs, log results, and more.
Using the Test Executive	Describes how to run and use the test executive.
More About Test Executive Driver	Describes other features available in the test executive such as multiple users support and touch panel support.
Building and Executing Your Application	How to build and execute an application after it has been created.

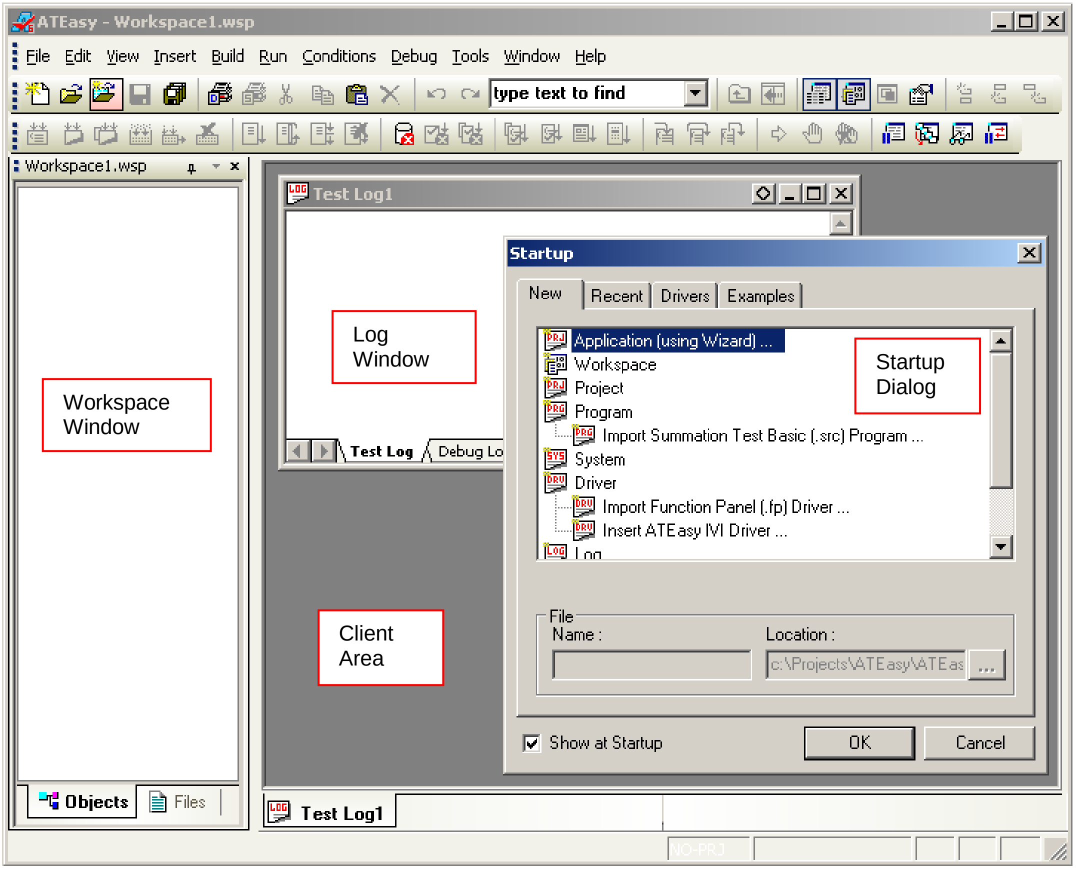
Starting *ATEasy*

Once ***ATEasy*** has been set up, the ***ATEasy*** icon appears on the desktop.

▼ To start ***ATEasy***, follow these steps:

1. Double-click the ***ATEasy*** icon or, select **ATEasy** from the **ATEasy** menu under **Programs** on the **Start** menu.

The first time you start the program, the following screen appears:



This screen represents the ***ATEasy*** Integrated Development Environment (IDE). At first the main window displays two empty windows (**Workspace** and **Log**) and the **Startup** dialog. The **Startup** dialog allows you to create a new application using the application wizard, it also let you open recent, drivers and examples workspace and project files. The **Workspace** window displays objects representing document shortcuts and objects opened in the IDE. The **Client area** is where you will do the majority of your work (adding and editing code, etc.). The **Log** window is displayed on the client area and showing the **Test Log** tab that is used to record **print** statements that your application may have or the default output of a test program. These areas will be covered more fully later in this chapter.

Before creating your first application, you must learn about the application types available when you use the ***ATEasy*** **Application Wizard** to create the application.

Application Types

There are three different types of ***ATEasy*** applications available when you use the Application Wizard to create your application. The following types are available:

Test Application - the most common ATEasy application type. The project has multiple programs and a system. In such projects, each program is used to test a single UUT. You can select the first program to run upon loading. The other programs run when invoking the run statement in your code in most cases, although a form lets the operator select which program to run.
By default the Test Application includes two special drivers. The first driver called TestExec.drv that provides a user interface to your application for running, debugging, and generating test reports for the test programs when they run. The second driver Profile.drv allows you to create and run profiles that enable you to run selected programs, tasks and tests in a custom sequence.

Instrument Panel Application – the project contains a system with one or more drivers, and no programs. Typically, you will create this project to provide a window allowing the user to view or control an instrument settings interactively, or a control panel that is used for process control. As an example, you could have a virtual instrument of a switch matrix displayed while debugging a test program. The virtual instrument would display the status and configuration of the switches while the program is running.

Other Application – a generic project containing any modules you select or create. The user may add a new or existing program, system, and drivers when creating the project.

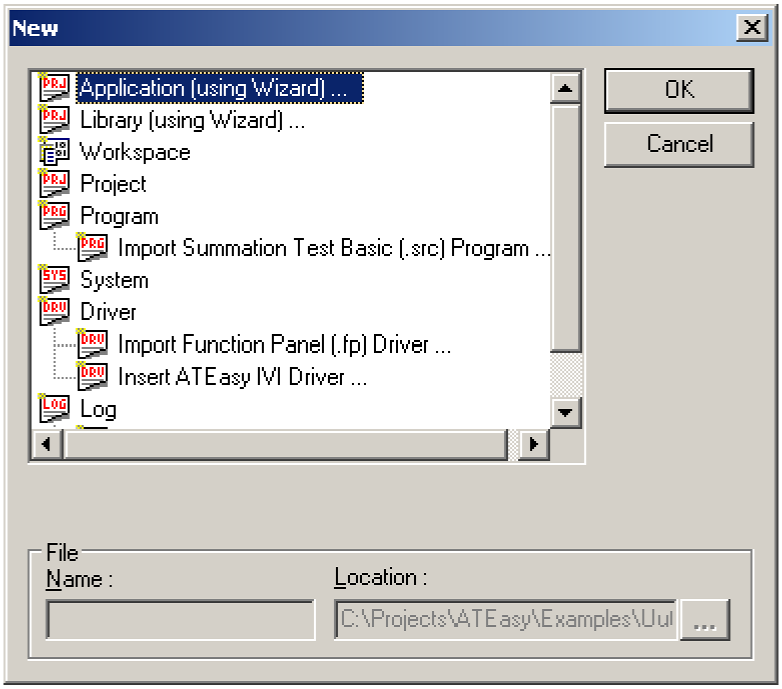
Creating an Application

You may create an application in one of two ways. You may use the ***ATEasy* Application Wizard** or create a new project and then manually insert new or existing programs as well as a system and drivers files to the project. In this section we will use the Application Wizard to get a "jump start" and create your application.

▼ To start the Application Wizard:

1. Select **New** from the **File** menu or click on from the Standard toolbar or just use the **Startup** dialog shown in the previous section.

   The New dialog box displays as shown here:

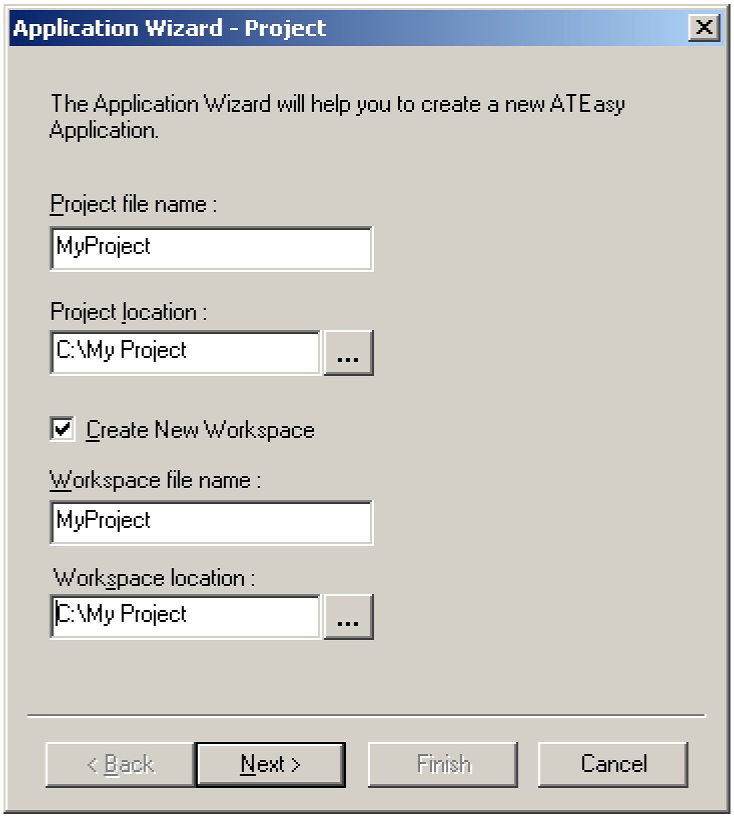

New
Application (using Wizard) ...
Library (using Wizard) ...
Workspace
Project
Program
Import Summation Test Basic (.src) Program ...
System
Driver
Import Function Panel (.fp) Driver ...
Insert ATEasy IVI Driver ...
Log
OK
Cancel
File
Name :
...

Location : C:\Projects\ATEasy\Examples\Uul

2. Select **Application Wizard** from the list and click **OK**. The Application Wizard appears.

▼ To set the Project Name and Location:

3. Type **MyProject** for the project file name and **C:\MyProject** for the project location as shown below. ***ATEasy*** creates the folder if does not exist.



The Application Wizard will help you to create a new ATEasy Application.

Project file name : MyProject

Project location : C:\My Project

☑ Create New Workspace

Workspace file name : MyProject

Workspace location : C:\My Project

Make sure the checkbox is selected for Create New Workspace and type the Workspace name and folder as shown here.

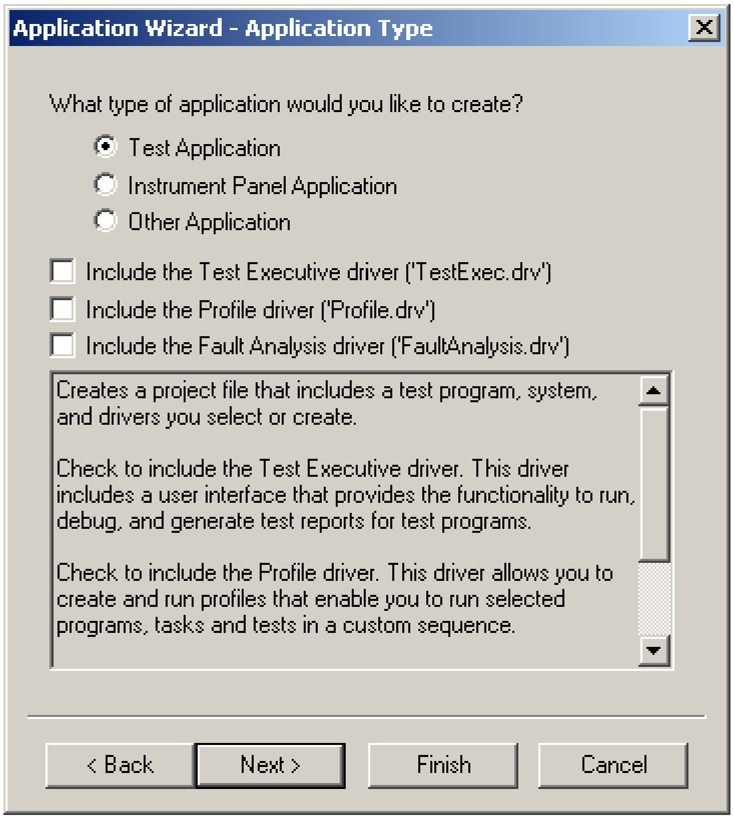
4. Press ENTER or click the **Next** button to continue to the next step.

Note: By default ***ATEasy*** files are created in binary format. ***ATEasy*** files can be saved in the Text format. This allows merging when several users are working on the same the file. It also offers better support for version control software such as Microsoft Visual Source Safe.

▼ To select the Application Type:

5. Select **Test Application** as the project type.

6. Uncheck **Include the Test Executive** driver, the **Profile** driver **and the Fault Analysis driver**. The dialog should look as follows:



Application Wizard - Application Type

What type of application would you like to create?

● Test Application
○ Instrument Panel Application
○ Other Application

□ Include the Test Executive driver ('TestExec.drv')
□ Include the Profile driver ('Profile.drv')
□ Include the Fault Analysis driver ('FaultAnalysis.drv')

Creates a project file that includes a test program, system, and drivers you select or create.

Check to include the Test Executive driver. This driver includes a user interface that provides the functionality to run, debug, and generate test reports for test programs.

Check to include the Profile driver. This driver allows you to create and run profiles that enable you to run selected programs, tasks and tests in a custom sequence.

7. Click **Next** to continue.

We will include the test executive driver later in this chapter.

▼ To create the application:

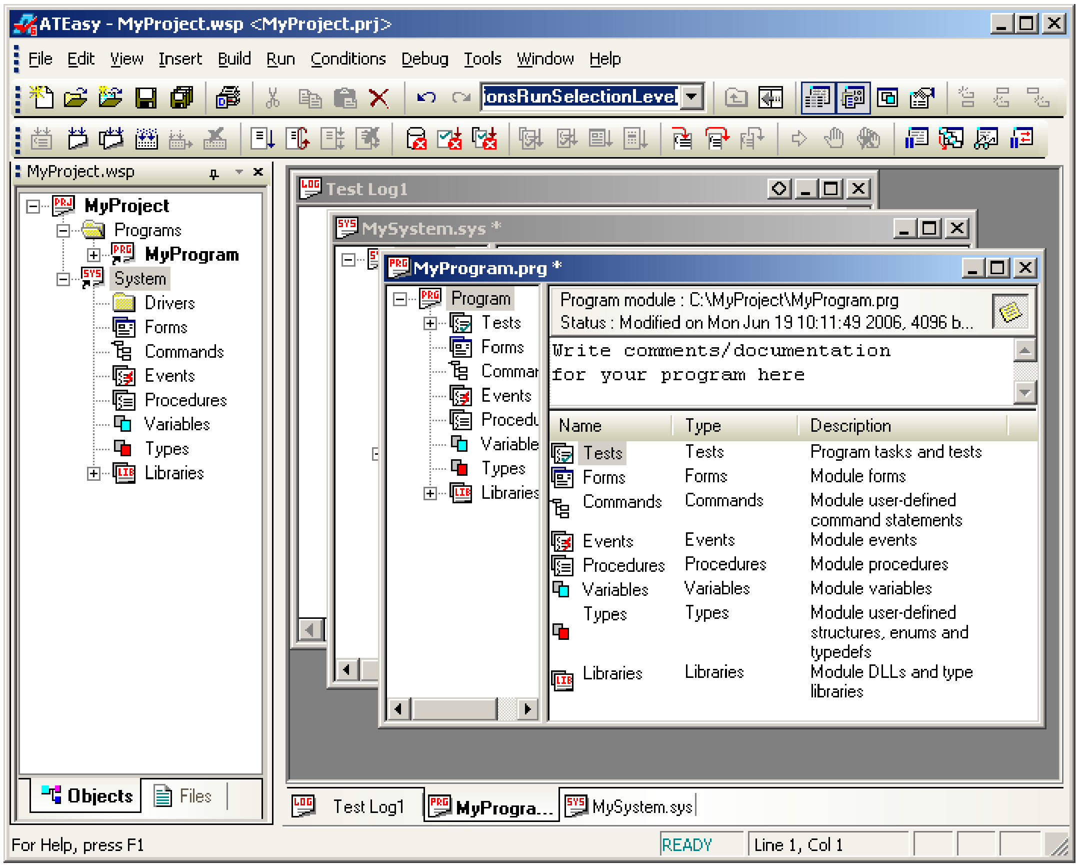
8. The next screen sets the program file name and defaults to the project name and location (C:\MyProject\MyProgram.prg). Rename the Program name to **MyProgram.prg** and click **Next** to continue.

9. The next screen sets the system file name and defaults again to the project name and location (C:\MyProject\MyProject.sys). Rename the System name to **MySystem.sys** and click **Next** to continue.

10. The last screen in the Application Wizard allows you to select drivers. While you are *not* going to add any drivers here, to access a list of drivers, click the new driver icon on the toolbar. The driver list is now available. Click the ellipse button ... to obtain a list of drivers (the default driver location is the ***ATEasy*** Drivers folder, for example, **`C:\Program Files[ (x86)]\ATEasy\Drivers`**).

11. Click **Finish**. The Application Wizard displays a confirming dialog box. This box lists the choices you have made through the Application Wizard. Click **OK** to create the files required for the new application.

After ***ATEasy*** generates the application files, the files are loaded to the IDE and the Workspace window displays the newly created project file and its contents. The two new modules, the program, MyProgram.prg, and new system, MySystem.sys, are displayed in **Document Views** windows. Each window is divided into **tree** and **object** views with a splitter that can be moved to separate the two views.



Program module : C:\MyProject\MyProgram.prg
Status : Modified on Mon Jun 19 10:11:49 2006, 4096 b...

```
Write comments/documentation
for your program here
```

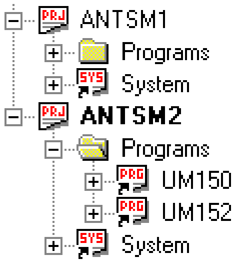
Name	Type	Description
Tests	Tests	Program tasks and tests
Forms	Forms	Module forms
Commands	Commands	Module user-defined command statements
Events	Events	Module events
Procedures	Procedures	Module procedures
Variables	Variables	Module variables
Types	Types	Module user-defined structures, enums and typedefs
Libraries	Libraries	Module DLLs and type libraries

More about the IDE

Before you go any further with building your application, there are a few key concepts to understand about the ***ATEasy*** IDE. These are:

Active Project – the workspace may contain multiple projects. It is important to understand which project is active, since the Build, Run, and Debug commands apply to the active project. The workspace window shows the active project in boldface. You can set the active project by selecting the project to make active and select the **Set Active Project** from the **File** menu.

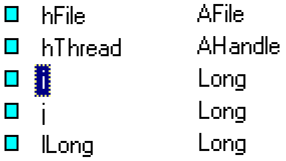
ANTSM1 and ANTM2 are projects: **ANTTM2** is the Active Project.



Selected Object – objects are displayed in the IDE windows with their image representing their type and name. Clicking on an object such as a procedure or a variable will make that object the selected object.

The selected object properties such as type or name are displayed in the **Properties Window**.

As shown here the variable **i** is the selected object.

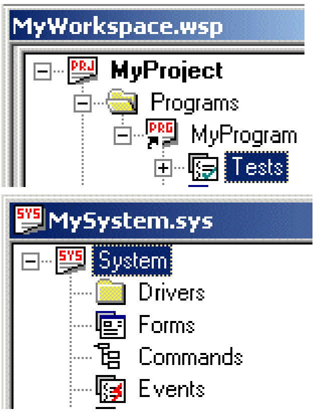


hFile	AFile
hThread	AHandle
i	Long
j	Long
lLong	Long

Edit commands such as **Cut** or **Paste** work only on the selected object.

Active Document – The active document is the document or file where the selected object resides. File Operations such as the **Save** command apply to the active document.

The Active Documents shown here are **MyProgram,** (MyProgram.prg) program, and the system, **MySystem.sys**. Note that even though the workspace document is the workspace file, the active document is the program, since it owns Tests, which is the selected object.

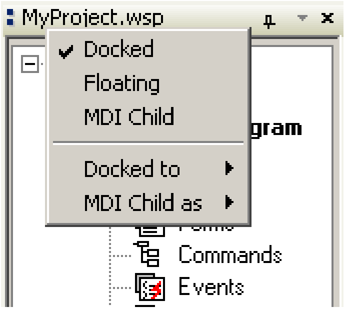


Dockable windows – Dockable windows can be docked to any side of the IDE main window by dragging their title bar close to the border of the main window.

Dockable windows in ***ATEasy*** are the built-in windows: Workspace, Log, Variables, and all the debug windows (for example, Watch window). Dockable windows can be in one of the tree states: Docked, Float, and MDI Child.

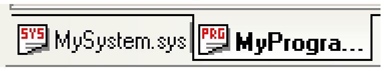
You can see the differences between the states of these windows by right clicking on the diamond button appearing on the dockable window title bar.

The dockable workspace window shown here is in **docked** state.



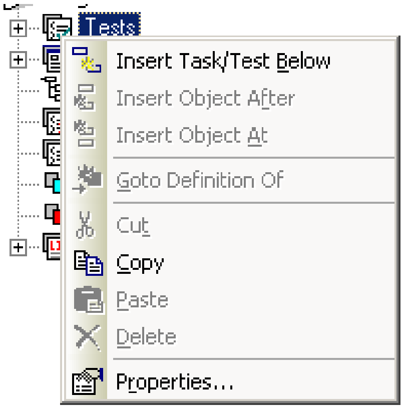
MDI child windows – MDI stands for **M**ultiple **D**ocument **I**nterface. The interface displays multiple documents in the main window. Each document is in its own window, called an MDI Child window. In ***ATEasy***, all the document views as well as the built-in windows (for example, the Log Window) are displayed as MDI Child windows. The windows are displayed in a rectangular area called the **MDI Client area.** It is typically surrounded by docked windows, with the status bar below and toolbar(s) above.

An MDI child window can be activated by clicking on the window or on the Tabs appearing below the MDI client.

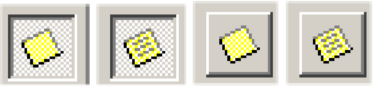


The second Tab referring to a document view displaying Tests in MyProgram.prg is the active MDI child.

Tree views – Both document views and the workspace window contains tree views. The tree view displays objects in a tree control, each with an icon representing its name. Additionally, a plus/minus sign indicates if the object can be expanded or collapsed. You can perform many editing operations on objects via the tree view such as: Rename (F2), Cut, Copy, Paste, Drag and Drop, or other operations from the Edit and Insert menus. Clicking on the object with the right mouse button invokes the object context menu as shown here.



Description View and **Description Button –** Most objects views allow you to enter description text (notes) to describe and document the object.



The button can be pressed or empty (no text entered).

Your First Test Program

Your first program contains one task (Power Tests) and two tests (PS1 and PS2). Define the test requirements by first defining the test type and filling up the test properties. Then, write some code in the test to tell ***ATEasy*** the test result. Consequently, when you run this program you will see actual test results.

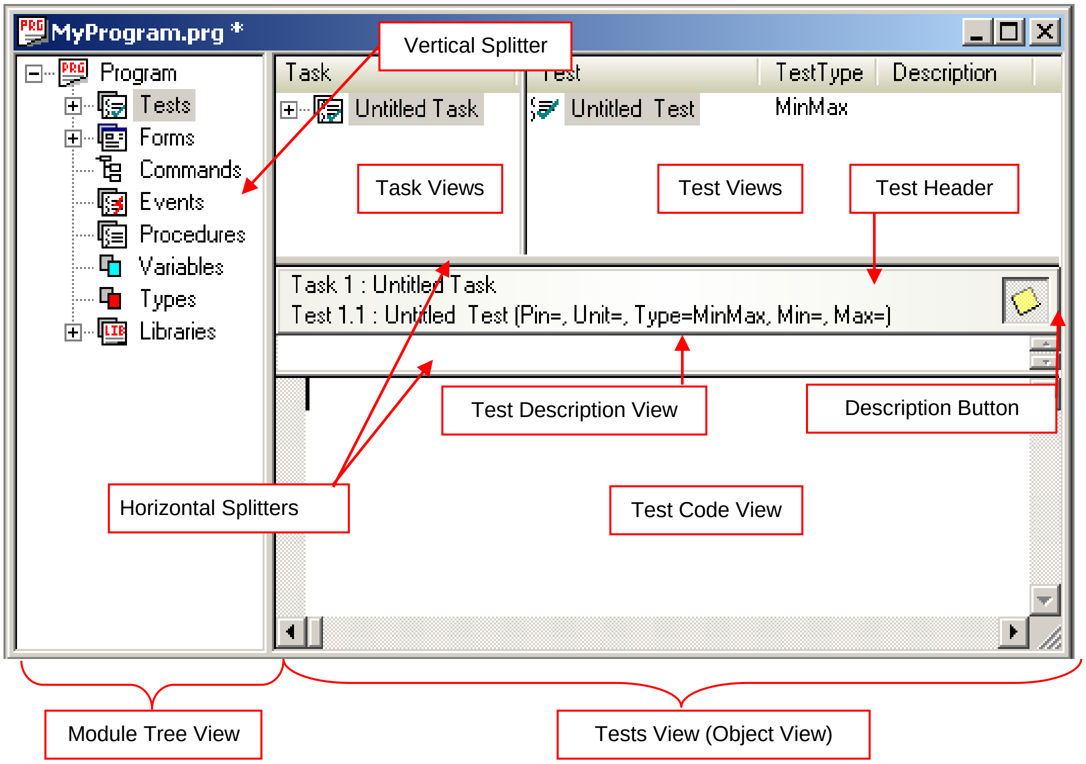
The next step is to add tasks and tests that will be part of your program.

▼ To add a task and tests:

1. Activate the **MyProgram** document view and click on the **Tests** submodule in the tree view. If the document view is not visible, double-click on the **MyProgram** program shortcut.

2. Select **Task/Test Below** from the **Insert** menu or from the standard toolbar. A new task and a test are inserted below the Tests submodule.

The Tests view is shown here:



Task 1 : Untitled Task
Test 1.1 : Untitled Test (Pin=, Unit=, Type=MinMax, Min=, Max=)

The Tests View displayed in the object view displays a split view with two horizontal splitters and one vertical splitter. The following areas are displayed as shown in the previous diagram:

Tasks View – a tree view lists the tasks and tests comprising the program. Selecting a task in this view makes it the Current Task.

Tests View – a list of all tests belonging to the selected task within the Tasks View. Selecting a Test in this view makes it the Current Test.

Test Header – shows the current task name and number, as well as the current test name, type, and properties. The header also contains the Description Button used to show or hide the Test Description View.

Test Description View – a text editor where the current test description is entered.

Test Code View – a text editor where the current test programming code is entered.

▼ To rename the Task and Test:

3. Change the name of the task by clicking on the Untitled Task name and typing **Power Tests**. Note that if you clicked on the task image (icon), the tree view will not be in renaming mode (clicking on the image changes the selected object). You must click on the name to enter editing mode.

Do the same for the Untitled Test and rename it to **PS1**.

▼ To switch to Object View Only:

4. At this point, the document view displays the tree and object at the same time. Since you are planning to work only with the object view, it would be nice to have a larger work area on the screen.

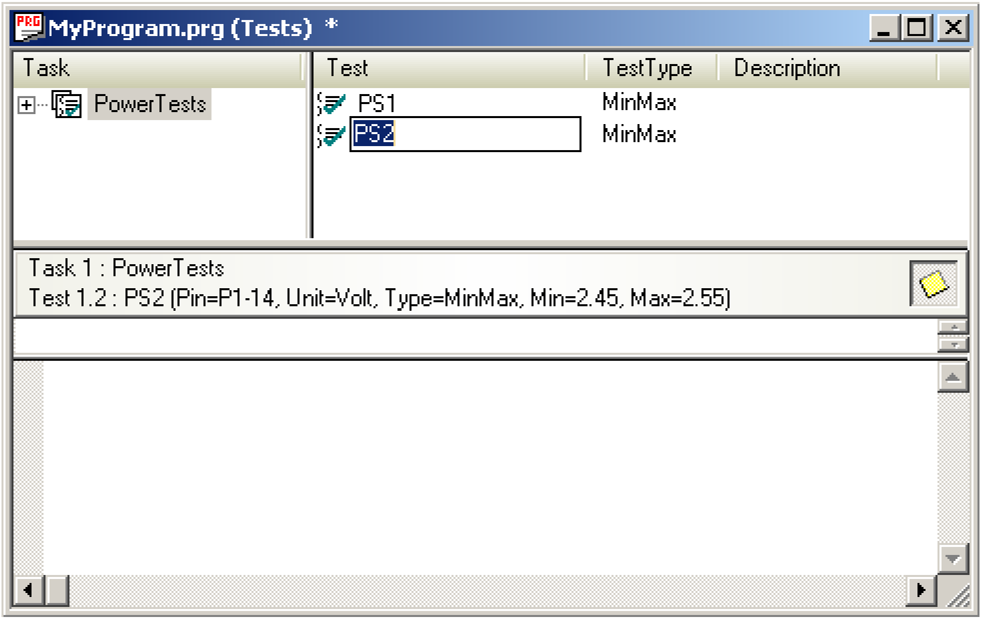
5. Select **Object Only** from the **View** menu. The tree view disappears and the object view is displays on the whole client area of the document view.

▼ To insert the PS2 Test:

6. To add another test, highlight **Test #1** in the Tests View (PS1) and then select **Test After** from the **Insert** menu or from the standard toolbar. A new test is inserted after PS1.

7. Rename the new test from Untitled Test to **PS2**.

The document view should now look as illustrated in the figure below:



Task 1 : PowerTests
Test 1.2 : PS2 (Pin=P1-14, Unit=Volt, Type=MinMax, Min=2.45, Max=2.55)

While you have added the tests, you have not set any test properties. Continue with the next section to learn how to set the test properties.

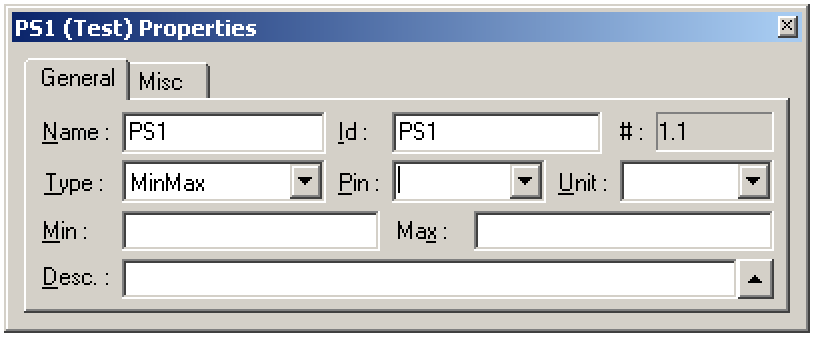
Test Properties

Task and Test, like all other ***ATEasy*** objects, have properties. In the case of a Task, the properties include basic information such as name, ID, and description. However, the Test properties include additional information regarding the test type and other properties used at run-time to determine whether the test passed or failed.

▼ To display the Test Properties:

1. Open the PS1 context menu by clicking the right mouse button on **PS1** and selecting **Properties** . .

The properties window opens as shown here:



Name : PS1
Id : PS1
: 1.1
Type : MinMax

Most of the Test's properties are common to all tests. Some properties however are different from one test type to another. The common properties are:

Property	Description
Name	The Name of the test. Names are not unique and more than one test or task may have the same name. They appear in the test log report when the application is running, and are used by the operator to identify the test.
ID	Test ID is a unique identifier used with programming statements to identify the test uniquely. Typically, they are used with task or test statements that are used to branch to another task or test at run-time.
Type	Test Type. Can be **MinMax**, **Ref2**, **RefX**, **Tolerance**, **Precise**, **String**, and **Other**. The test type sets the requirements determining if the test status is **PASS** or **FAIL** at run-time. Changing the test type changes the available properties in the properties window. Please refer to the next section *Test Status and Test Result* for additional information.
Pin	The Pin where the measurement is being taken. This is a list of user-defined UUT pins. Once you enter a pin name, it is added to the list automatically.
Unit	The Units of measurement. Common units such as "Hz," "Volt," "Ohm," etc. are available and you may add additional Units. Once a new Unit is entered, it is added to the list automatically.
Description	A description of the test for documentation purposes. You can expand the size of this edit box by clicking on the maximize button located on the right of this edit box.
Tag (Misc Page)	Application specific data that is attached to the test and can be used by programmatically by the application.

In our example we will use the **MinMax** test type. In the MinMax test you set the Minimum and Maximum number allowed for the test result so the test will have a **PASS** status. If the test result is higher than the Max value, or lower than the Min value the test status will be **FAIL**

▼ To set the Tests' Properties:

2. Enter the following properties for PS1:

   Pin **P1-13**
   Unit **Volt** (you either type this or select from the combo box list)
   Min **1.25**
   Max **1.35**

3. Select the PS2 test, and enter the following values in PS2 Test Properties:

   Pin **P1-14**
   Unit **Volt**
   Min **2.45**
   Max **2.55**

The next section discusses the properties of the Test Status and Test Result.

Test Status and Test Result

As discussed earlier, the output of the test is a single measurement result. This result should be stored in an ***ATEasy*** built-in internal variable called **TestResult.** Upon completion of a test, ***ATEasy*** automatically evaluates the test status according to the test's properties and the **TestResult** variable value, and assigns a value to another internal variable called **TestStatus**. This value can be any of the following constants: **NONE**, **PASS**, **FAIL**, or **ERR**. The value will be used when printing the test result to the test log report.

TestResult and **TestStatus** are pre-defined internal ***ATEasy*** variables. **TestResult** is defined as the type *Variant*. This type of variable can accept different data types such as Integers, Float, or String. **TestStatus** is an enumerated type (*Enum*).

The following table describes the available test type and their properties:

Test Type	Properties	Evaluation performed by *ATEasy*
MinMax	Min, Max	Analog test where TestResult is compared against Min and Max. TestStatus is PASS if TestResult is between the two.
RefX	Mask, Ref	Digital test (hexadecimal) in which **TestResult** is compared against Ref ignoring the bits specified as Do not Care in Mask. **TestStatus** is **PASS** if **TestResult** is equal to **Ref** (except for masked-out bits).
Ref2	Ref-2	Typically used with digital tests where data compares the 32-bit (long) **TestResult** with a binary reference mask **Ref-2**, ignoring the bits specified as "don't care" (x). If the result and the reference/mask are identical, the **TestStatus** is Pass.
Tolerance	Value, PlusValue, MinusValue	Analog test in which TestResult is compared against Value. TestStatus is PASS if TestResult falls within the Value minus MinusValue, and Value plus ValuePlus.
Precise	Value	Analog test in which **TestResult** is compared against a precise Value. **TestStatus** is **PASS** if **TestResult** is equal to the **Value** property.
String	String	TestResult is compared against a string. TestStatus is PASS if TestResult equal to the String property.
Other	None	***ATEasy*** performs no automatic comparison. This test type is used when none of the above evaluations fit. You can write an evaluation code and assign the status to the **TestStatus** variable.

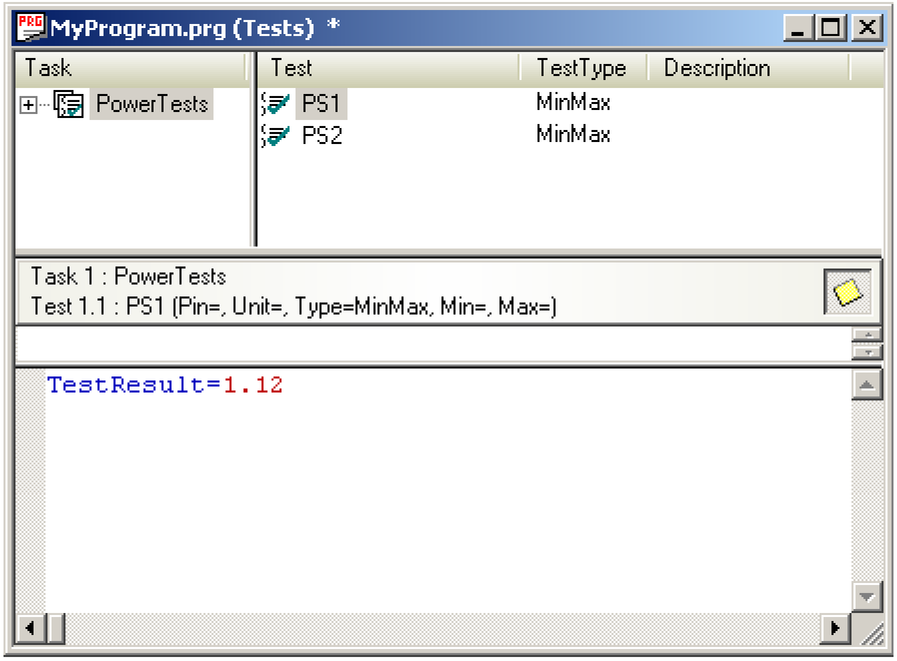
Normally, test code contains code to set up the switching and measurement instruments. A measurement will be taken and then assigned to **TestResult** as shown in the following example:

```
RELAY Close (7)
DMM Measure (TestResult)
```

For this example, we are going to enter sample data to set the **TestResult** without writing any setup or measurement statements.

▼ To set TestResult of the PS1 and PS2 tests:

1. Select **PS1** from the Tests View.
2. As shown below, click in the Test Code view.
3. Type: **TestResult=1.12** as shown here:



Task	Test	TestType	Description
PowerTests	PS1	MinMax	
	PS2	MinMax	

Task 1 : PowerTests
Test 1.1 : PS1 (Pin=, Unit=, Type=MinMax, Min=, Max=)

```
TestResult=1.12
```

4. Repeat steps 1-3 for the PS2 test. The value for PS2 is **TestResult=2.5**

As you can probably see, the tests are set for one to **FAIL** and one to **PASS** when you run them. Continue with the next section to learn how to run the application.

Running Your First Application

Your application is now ready to run. To run the application, use the **Start** command from the **Run** menu. Other Run commands are available. They are used to abort or pause the running application and change the way the test program is run.

The **Abort** command is used to stop the application from running, while the **Pause** command will suspend the application. Once the application pauses, you may use other Run commands such as **Current Test** to repeat running the last test, **Skip Test** to skip the current test to the next test, and others.

ATEasy also lets you set conditions causing the application to pause when the condition is met. You can use the **Task By Task**, or the **Test By Test** to pause in the beginning of each task or test. Another condition that can be set is the **Stop On Failure** command, which lets you pause when a test fails.

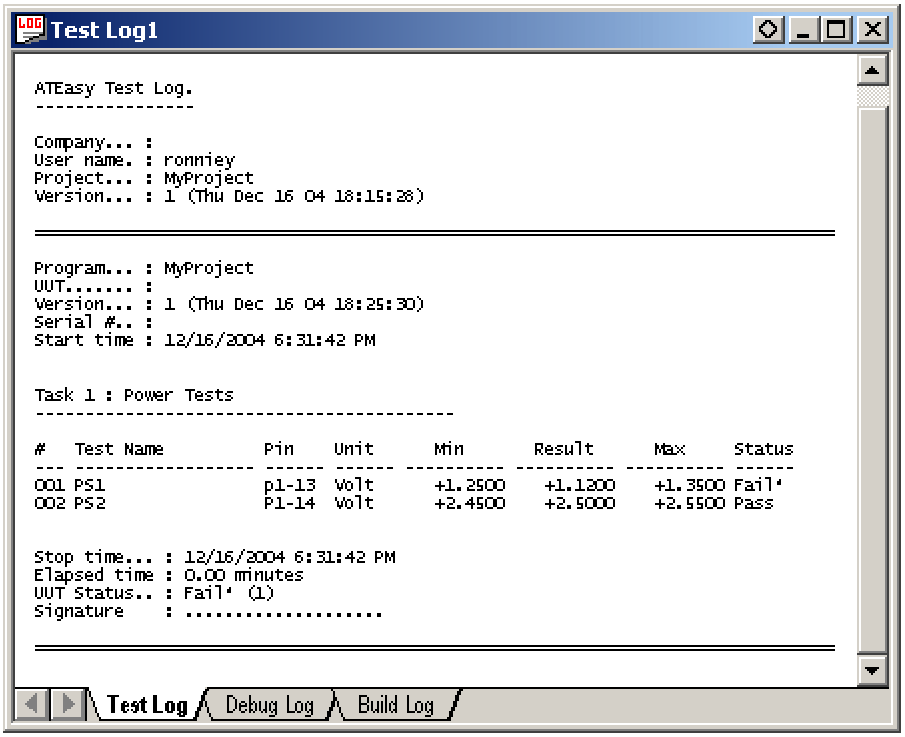
Other debugging commands are available from the **Debug** menu. These provide code level debugging. Using debug commands you can use the **Toggle Breakpoint** command to pause on a specific line of code. You can use the **Step Into**, **Step Over**, and **Step to Out** to walk through the lines of code.

To run the program for the first time select **Start** from the **Run** menu. ***ATEasy*** compiles the required parts of the application and starts running the application. As the program runs, a window appears displaying the test results report. This window is the **Log Window**.

The Log Window

The output of any test program is the test results. This data is essential information required to determine if the Unit Under Test has passed all the tests and if not, what were the failures. ***ATEasy*** provides this (and other) information through the Log Window as shown below. The Log Window is a dockable window with three tabs and can be shown by selecting the **Log Window** command from the **View** menu or from the Standard toolbar. This log is automatically generated when the program runs.

Before you go any further, click the docking button to cycle through the positions of the log window and switch to the MDI Child Window as shown here:



```
ATEasy Test Log.
----------------

Company... :
User name. : ronniey
Project... : MyProject
Version... : 1 (Thu Dec 16 04 18:15:28)

==========================================================================

Program... : MyProject
UUT....... :
Version... : 1 (Thu Dec 16 04 18:25:30)
Serial #.. :
Start time : 12/16/2004 6:31:42 PM


Task 1 : Power Tests
------------------------------------------

#   Test Name          Pin    Unit      Min        Result     Max        Status
--- ------------------ ------ ------ ---------- ---------- ---------- ------
001 PS1                p1-13  Volt      +1.2500    +1.1200    +1.3500 Fail*
002 PS2                P1-14  Volt      +2.4500    +2.5000    +2.5500 Pass


Stop time... : 12/16/2004 6:31:42 PM
Elapsed time : 0.00 minutes
UUT Status.. : Fail* (1)
Signature    : ....................

==========================================================================
```

There are three pages in the log view:

The **Test Log** displays the test results generated by ***ATEasy*** automatically when you run a program. The test results include some information about your program: for each task that was running, its number and name are output; and for each test its number, name, pin, unit, test type requirements such as Min or Max, test result, and the test status are output. The end of the test log report contains summary information including the status of the UUT. ***ATEasy*** determines this from the status of the tests you ran. When debugging the application, ***ATEasy*** appends a description of the actions taken by the user, such as abort, or skip test, to the test log. This can be later used to track and replicate user actions in order to analyze and debug the UUT. You can select the **Log Failures Only** from the **Conditions** menu if you wish the log to display only the failed tests results.

The format and the content of the test log can be customized. For example: you can set the test log to display in HTML format instead of text to provide more readable output including various fonts, color, graphics, and more. You can include an image showing a chart of data that the application acquired. You can disable ***ATEasy*** from outputting anything to the log and use the **print** statement or other log functions from the internal library to output any data you wish.

The **Debug Log** displays information printed from programs for debug purposes. The output to this window is usually done using the **trace** statement.

The **Build Log** displays ***ATEasy*** compilation information including any actions taken by ***ATEasy*** (such as Compiling, etc.) and error messages.

Scroll to the right to see that PS1 has a status of **Fail**, while PS2 has a status of **Pass**.

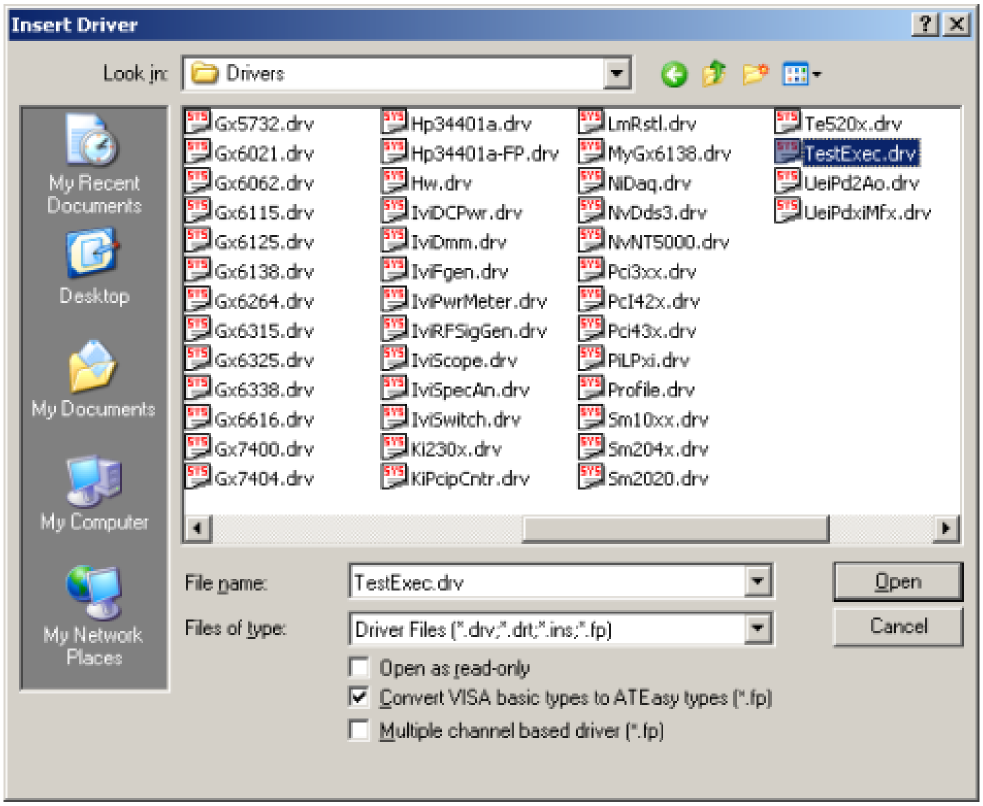
Adding the Test Executive Driver

By now, you have now your first application in ***ATEasy***. However, the application that you ran did not have any user interface. The output it generated was displayed in the IDE's Log Window. As discussed in Chapter 3, when you use the **Build** command from the IDE, the project is compiled and the result is an executable file that can be run independently of the IDE (similar to any Microsoft Windows application). The executable still needs a user interface. ***ATEasy***, similar to other programming languages, supports building of custom user interfaces, allowing you to design your own windows with controls and menus. These windows are called **Forms** and you can use them to design your own test executive. In this chapter, we will use the Test Executive, **TestExec.drv** driver supplied with ***ATEasy*** to provide our application user interface**.** This driver will be added to the application system to provide a test executive for your application. In "*Chapter 8 – Forms*," you will design your own user interface.

▼ To add the test executive driver to your application:

1. From the workspace window, expand **MyProject** by clicking on the plus sign next to its image. Expand the **System** in a similar way and then select **Drivers.**

2. Right-click on **Drivers**. The Drivers context menu appears. Select **Insert Object Below**. The Insert Driver dialog appears as shown here:



Look in: Drivers

Gx5732.drv
Gx6021.drv
Gx6062.drv
Gx6115.drv
Gx6125.drv
Gx6138.drv
Gx6264.drv
Gx6315.drv
Gx6325.drv
Gx6338.drv
Gx6616.drv
Gx7400.drv
Gx7404.drv
Hp34401a.drv
Hp34401a-FP.drv
Hw.drv
IviDCPwr.drv
IviDmm.drv
IviFgen.drv
IviPwrMeter.drv
IviRFSigGen.drv
IviScope.drv
IviSpecAn.drv
IviSwitch.drv
Ki230x.drv
KiPcipCntr.drv
LmRstl.drv
MyGx6138.drv
NiDaq.drv
NvDds3.drv
NvNT5000.drv
Pci3xx.drv
PcI42x.drv
Pci43x.drv
PiLPxi.drv
Profile.drv
Sm10xx.drv
Sm204x.drv
Sm2020.drv
Te520x.drv
TestExec.drv
UeiPd2Ao.drv
UeiPdxiMfx.drv

File name: TestExec.drv

Files of type: Driver Files (*.drv;*.drt;*.ins;*.fp)

☐ Open as read-only
☑ Convert VISA basic types to ATEasy types (*.fp)
☐ Multiple channel based driver (*.fp)

Note: If the window does not show any .drv file. Your Windows explorer is probably set the hide files that have the .drv extension. To show these files, Run the Windows Explorer and select **Folder Options…** from the **View** menu. Then Click on the **View** tab and check the **Show All Files** option.

3. Select the TestExec.drv driver from the ATEasy Drivers folder, and click Open to insert the driver. The driver is added to the Drivers folder in the system. In addition, a new document view is displayed in the IDE displaying the **TestExec.drv**. **Note**: After the TestExec driver is inserted you can disable it (without removing it from the System) by setting the Driver Shortcut **Disable** parameter to 1. To access the driver shortcut parameters page, right click on the TestExec in the Workspace window and Select Properties and then click on the Misc page.
4. You may repeat steps 1-3 to insert the **Profile.drv** driver. The profile driver allows you to create and run profiles that enable you to run selected programs, tasks and tests in a custom sequence. The Profile driver must be inserted before the Test Executive driver (use **Insert Driver At** command when the TestExec driver is selected). Once this driver is inserted the test executive menu will show special commands that allow you to create, edit, select and run profiles saved in profile files.
5. You may repeat steps 1-3 to insert the **FaultAnalysis.drv** driver. The Fault Analysis driver allows you to create conditions that are based on the program tests results. The operator can use this to troubleshoot and recommend ways of fixing the UUT. The Fault Analysis must be inserted before the Test Executive driver (use **Insert Driver At** command when the TestExec driver is selected). Once this driver is inserted the test executive menu will show special commands that allow you to create, edit, and analyze conditions saved in a condition file.

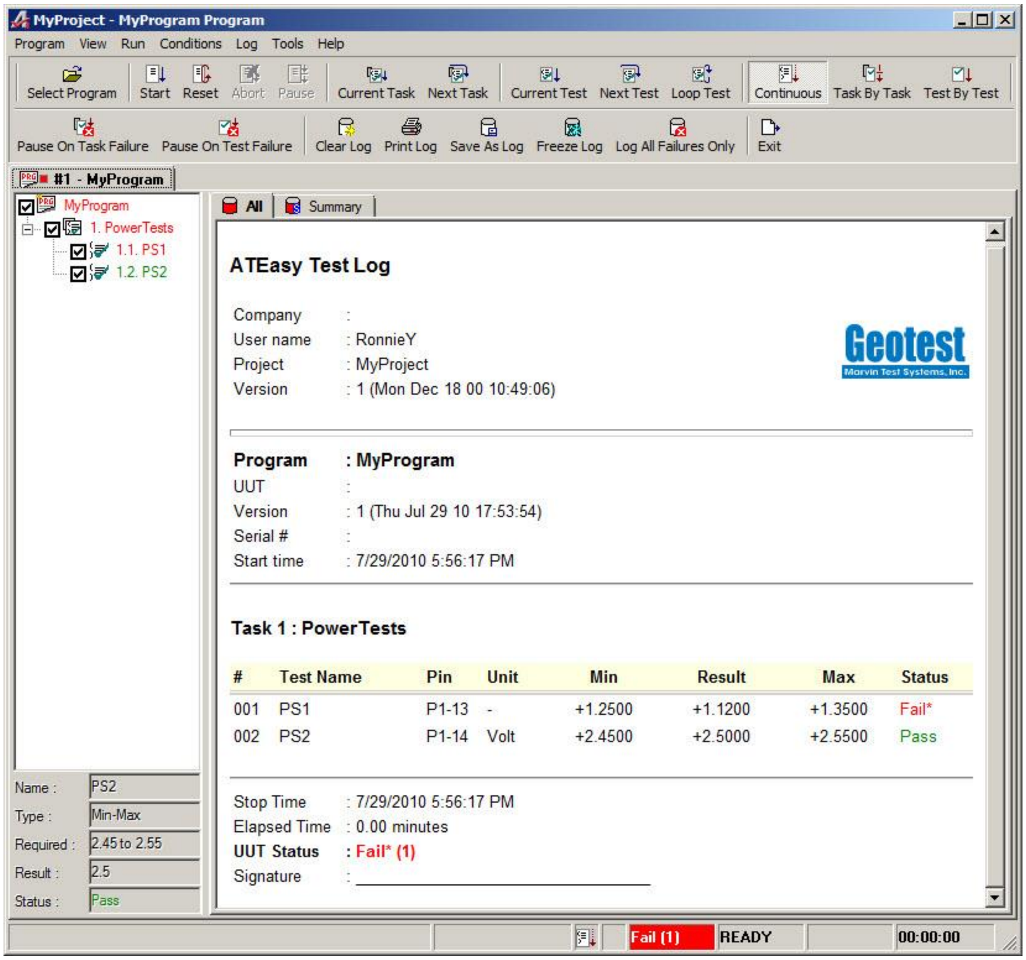
Using the Test Executive Driver

The test executive main window is an ***ATEasy*** Form displayed by the test executive driver. The main contains a menu, toolbar, status bar and a Log control to display the test results of the running program. The test executive provides support to differenet execution models of your project programs, these includes sequntial mode, parallel mode and mixed mode. These modes are mainly used when executing multiple UUTs and test programs at the same time (parallel mode) or in sequence (sequential mode).

▼ To use the Test Executive:

1. Select **Start** from the test executive **Run** menu. MyProgram will run.

   Notice the test log is now displayed in HTML format as shown here:



Name : PS2
Type : Min-Max
Required : 2.45 to 2.55
Result : 2.5
Status : Pass

ATEasy Test Log

Company :
User name : RonnieY
Project : MyProject
Version : 1 (Mon Dec 18 00 10:49:06)

Program : MyProgram
UUT :
Version : 1 (Thu Jul 29 10 17:53:54)
Serial # :
Start time : 7/29/2010 5:56:17 PM

Task 1 : PowerTests

#	Test Name	Pin	Unit	Min	Result	Max	Status
001	PS1	P1-13	-	+1.2500	+1.1200	+1.3500	Fail*
002	PS2	P1-14	Volt	+2.4500	+2.5000	+2.5500	Pass

Stop Time : 7/29/2010 5:56:17 PM
Elapsed Time : 0.00 minutes
UUT Status : Fail* (1)
Signature : ________________________

The Test Executive main window is divided to three panes: the **Tests pane**, the **Test Properties pane** and the **Log pane**. The Tests pane displays the current program or profile that is a subset of your application tests in a tree view. Each node in the tree has a checkbox that allows you to include the test or exclude it from running. The Test Properties pane display the current test properties including its name, its type the required values, result and its status. The Log pane displays the test log report showing the test log report in either text or HTML format.

At run-time when the test executive executes a test, the test node is highlighted and colored according to the test status (red for fail, green for pass, black for none), the test properties pane displays the test properties and result and the test log is appended with the test's result.

The test executive log report and the test executive window features are fully customizable from the test executive driver commands or by using the **View** menu or **Options** dialog. In addition, if the profile driver (**profile.drv**) is also included in the system, additional menu items will show to allow you to create, select and run test profile using the profile editor.

If you browse through the test executive menus, you will see that the test executive has commands allowing you to select which program to run (**Program** Menu). The **Run** and **Conditions** menus contain commands similar to the IDE **Run** menu, and the **Log** menu lets you save, clear, or print your test log.

2. Choose **Exit** from the **Program** menu. This exits the application and returns to the IDE.

You may want to browse through the test executive driver submodules to see how this driver implements the user interface displayed through the test executive window. Look under Forms to see the form that is used to create the main window of the test executive. Other interesting code resides under the test executive Events submodules. This is where the driver controls the test programs' behavior and implements some of the **Run** and **Conditions** statements. We will cover these topics in later chapters.

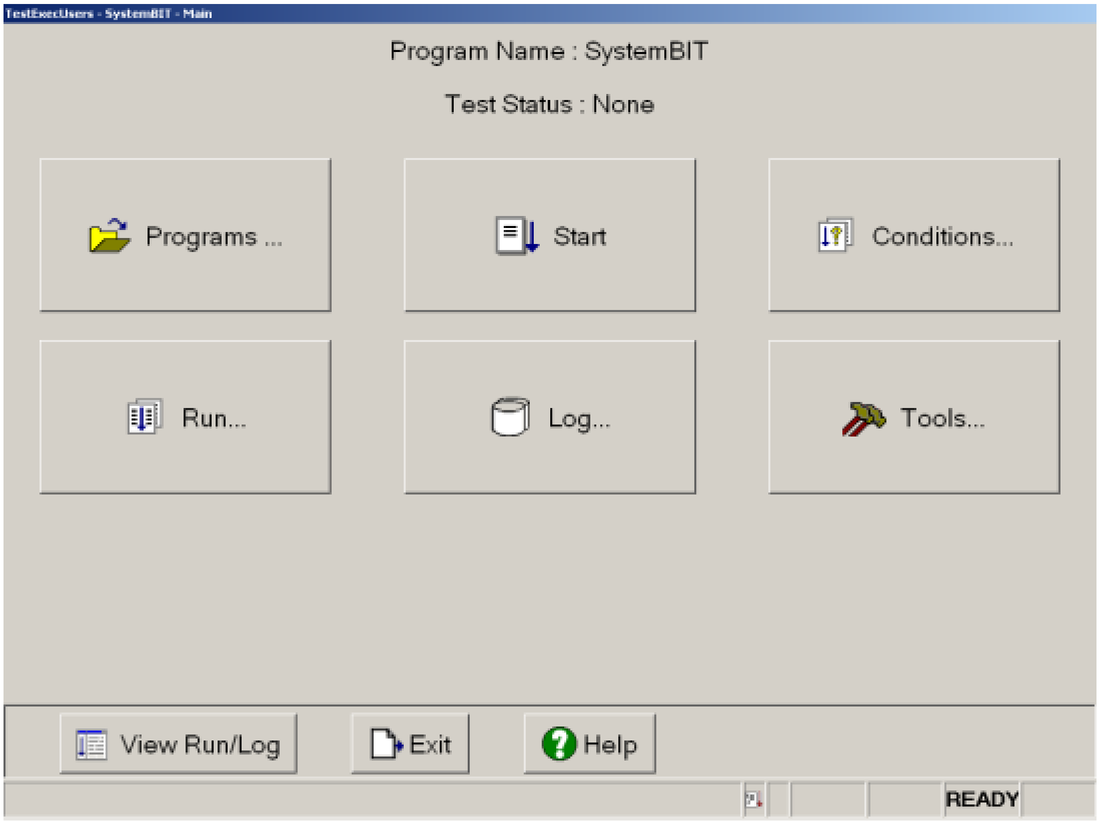
You have now created and tested a very simple application. Adding the test executive driver to the application allows you to provide a test executive to your application quickly.

More about Test Executive Driver

Not only the test executive driver provides a convenient user interface for your test programs, but also it provides multi users environment where the administrator creates user groups and user accounts. User group such as "Testers", "Supervisors", "Administrators" will be assigned with its own set of command menus, toolbars, options and different level of privileges. (Please refer to the TestExecUsers example in the ***ATEasy*** Examples.) Each user account will inherit the specific settings of the user group it belongs to. An administrator will have full privileges of the Administrators group.

Furthermore, the test executive driver provides two modes of UI operations: **Modal** and **Modeless**. The Test Executive main window shown in 'Using the Test Executive Driver' is the user interface in Modeless mode where you can access its commands through menus and a toolbar. In Modal Mode, Test Executive does not have menus and a toolbar; instead it has a series of buttons' forms, each consisting of command buttons in full screen. The Modal mode is used to support specifically for Touch Screen user interface in which all operations are performed by 'touching' the button controls instead of keyboard and mouse operations. It also provides a more directed and 'simple to use' user interface.

In Modal mode of Test Executive, the main window displays common commands as shown here:

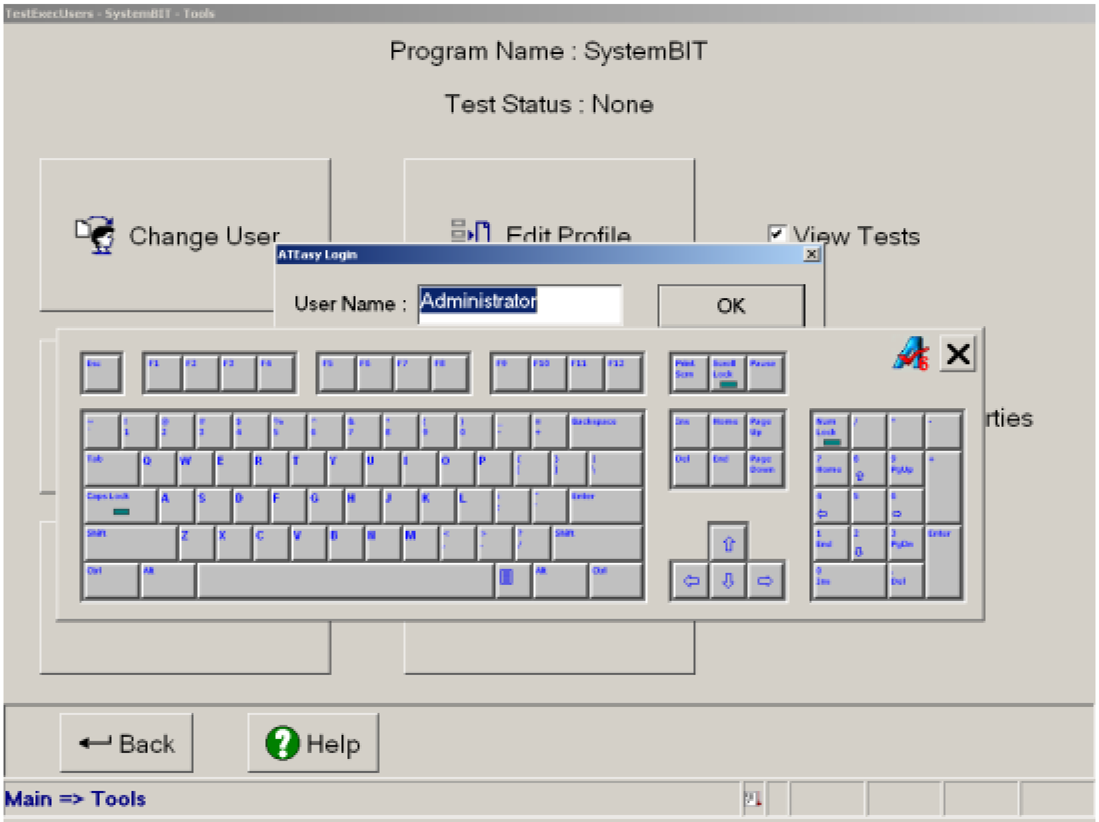


Program Name : SystemBIT

Test Status : None

You can switch between the two modes of user interface by opening the Customize or Users dialog (Options Page) from the test executive Tools menu.

In Modal Touch Panel mode, Test Executive uses a virtual keyboard for user input as shown below:

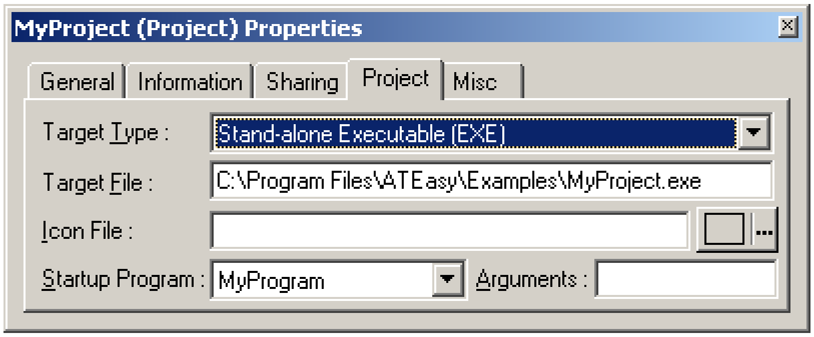


Whenever the Text Executive requires a user input, the virtual keyboard will automatically appear so that the user can input necessary information.

More information about the Test Executive driver is available in the ***ATEasy*** online help.

Building and Executing Your Application

You are now ready to build and execute your program. If you go to the Properties page for your project, you can see the default Application Target Type, the EXE file and Target File name, which will be created:



Target Type : Stand-alone Executable (EXE)

Target File : C:\Program Files\ATEasy\Examples\MyProject.exe

Icon File :

Startup Program : MyProgram

Arguments :

▼ To build and execute your application:

1. Select **Build** from the **Build** menu. Note that the Build Log tracks the compiling process and indicates when it finishes. Once the Build finishes, an EXE file is created.

2. Select **Execute!** from the **Build** menu to run the EXE file you just created. Alternately, you can use Windows Explorer to run the EXE file as in other Windows application. Your application will now run.

You can freely distribute the EXE file generated by ***ATEasy*** to your end users. Similar to other programming environments, the end user must install the run-time version of ***ATEasy*** as well as all external files your application uses, such as DLLs and ActiveX controls.

Before continuing to the next chapter, you need to remove the test executive driver from the system.

▼ To remove the test executive driver from the system:

Select the TestExec driver, in the Workspace window.

Select **Delete** from the **Edit** menu. The driver is now removed from the system.

Click **Save All** from the Standard toolbar to save your work.

You will be prompted to enter a file name for a workspace file. Type **C:\ MyProject\MyFirstWorkspace** and click **OK**.

Continue with the next chapter to add variables and procedures to your project.

Chapter 7 - Getting Starting With the GTDEMO Driver

About Getting Started With the GTDEMO Driver

This chapter discusses how to get started using the GT98901 in an ATEasy project. We will look at integrating a completed driver into an ATEasy project and making function calls to the demo board.

Topic	Description
Using a Completed ATEasy Driver	Using a driver that has been created by another developer.
Adding a Driver With the Application Wizard	Demonstrates using the Application Wizard to create a project with a pre-existing driver.
Configuring the Driver Shortcut Properties	Setting the properties of a driver to customize its functionality.
Reviewing a Driver’s Command List	How to find out about a driver’s capabilities.
Writing Test Code	How to use driver command’s to control instruments and execute tests.

Using a Completed ATEasy Driver

There are multiple interfaces that an instrument manufacturer can choose to allow the customer to utilize. Some instruments can use message-based communication protocols such as RS-232, GPIB, TCP/IP, others can use direct interaction with memory registers. Many manufacturers will package their APIs into DLLs. If a manufacturer does not provide an ATEasy driver with their instrument, a developer will need to create one using the techniques which will be described in *Chapter 9 – Drivers and Interfaces*. Creating an ATEasy driver is a one-time process.

Once a driver has been completed, it can easily be distributed and added to ATEasy Systems in any project that requires use of the instrument. In the case of a completed driver, the most pertinent functions have already been exposed and assigned to easy-to-use commands. In additional, customizable functionality will be added into the Driver’s Shortcut Properties so the test engineer can turn on and off features. The driver will also support a simple way to configure the location of the instrument

Adding a Driver With the Application Wizard

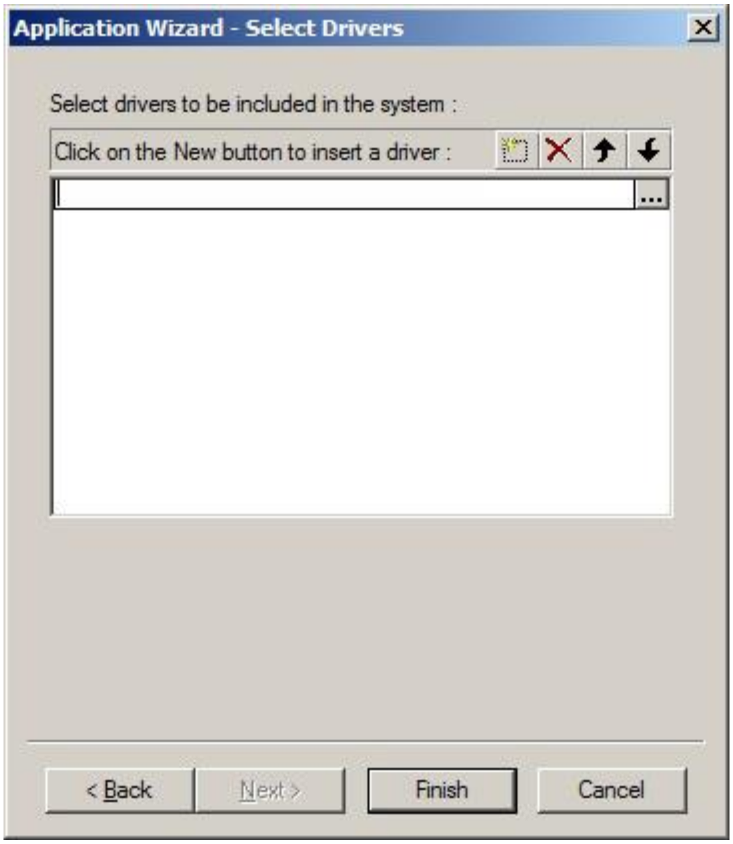
When using the Application Wizard, there is an option to add completed drivers to your newly created project To review using the Application Wizard, please look at the **Creating an Application** section of *Chapter 6 – Your First Project*.

▼ To add the GtDemo driver to your project:

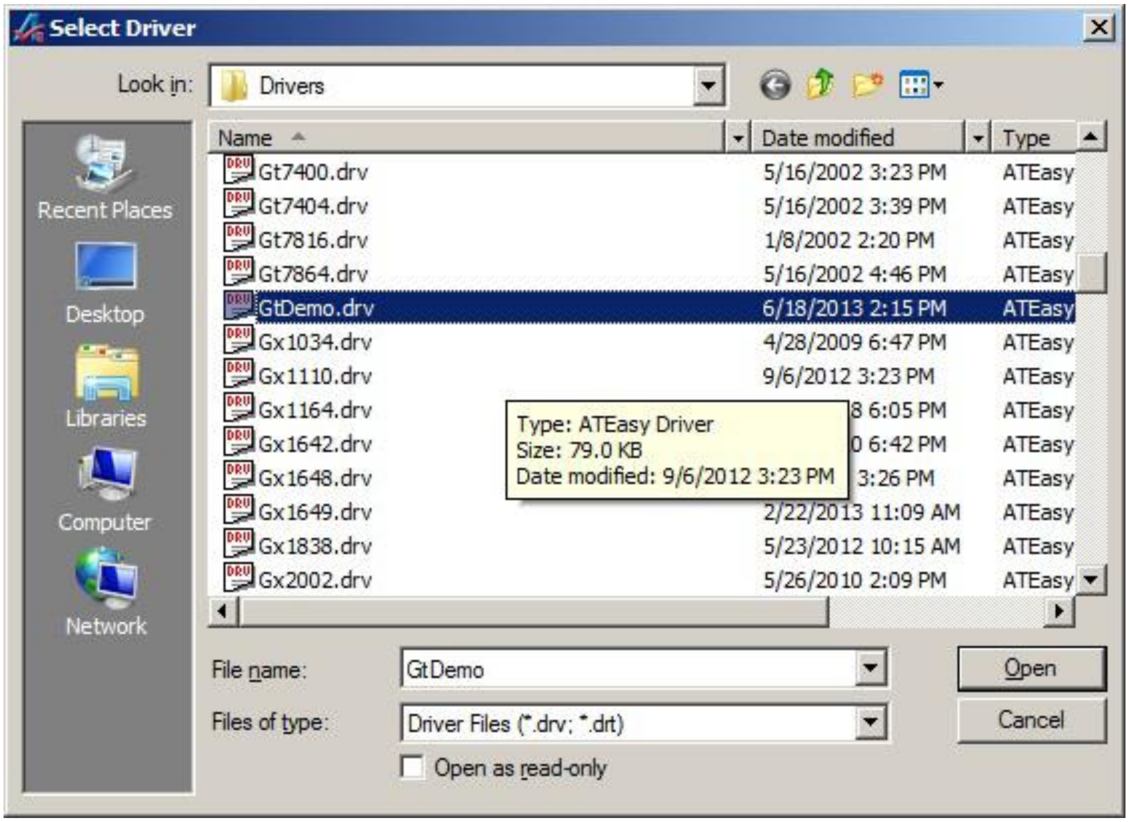
Start the Application Wizard from the **File | New** dialog box.

1. In Application Wizard dialog box, set the project file name to **GtDemoProject** and the project location to **C:\Projects\GtDemoProject**.
2. Check the **Create New Workspace** box. When you do this, the file name and location will automatically be copies from the projects data fields.
3. Click the **Next** button to advance the Application Wizard to the next screen.
4. Select **Test Application** as the project type.

5. Uncheck **Include the Test Executive** driver, the **Profile** driver **and the Fault Analysis driver**. You should have no checkboxes checked.
6. Click **Next** to advance the Application Wizard.
7. The Application Wizard will suggest a Program name. Click **Next** to advance the Wizard.
8. The Application Wizard will suggest a System name. Click **Next** to advance the Application Wizard.
9. Click the new driver icon on the toolbar of the **Application Wizard – Select Drivers** dialog.
10. Click the ellipse button ... to obtain a list of drivers (the default driver location is the ***ATEasy*** Drivers folder, for example, **`C:\Program Files[ (x86)]\ATEasy\Drivers`**).


Application Wizard - Select Drivers
Select drivers to be included in the system :
Click on the New button to insert a driver :
...
< Back
Next >
Finish
Cancel

11. Select the GtDemo.drv from the dialog. It will be located in **`C:\Program Files[ (x86)]\ATEasy\Drivers`** that should be the default location. Click **Open** when you are done.



Look in: Drivers

Name	Date modified	Type
Gt7400.drv	5/16/2002 3:23 PM	ATEasy
Gt7404.drv	5/16/2002 3:39 PM	ATEasy
Gt7816.drv	1/8/2002 2:20 PM	ATEasy
Gt7864.drv	5/16/2002 4:46 PM	ATEasy
GtDemo.drv	6/18/2013 2:15 PM	ATEasy
Gx1034.drv	4/28/2009 6:47 PM	ATEasy
Gx1110.drv	9/6/2012 3:23 PM	ATEasy
Gx1164.drv	8 6:05 PM	ATEasy
Gx1642.drv	0 6:42 PM	ATEasy
Gx1648.drv	3:26 PM	ATEasy
Gx1649.drv	2/22/2013 11:09 AM	ATEasy
Gx1838.drv	5/23/2012 10:15 AM	ATEasy
Gx2002.drv	5/26/2010 2:09 PM	ATEasy

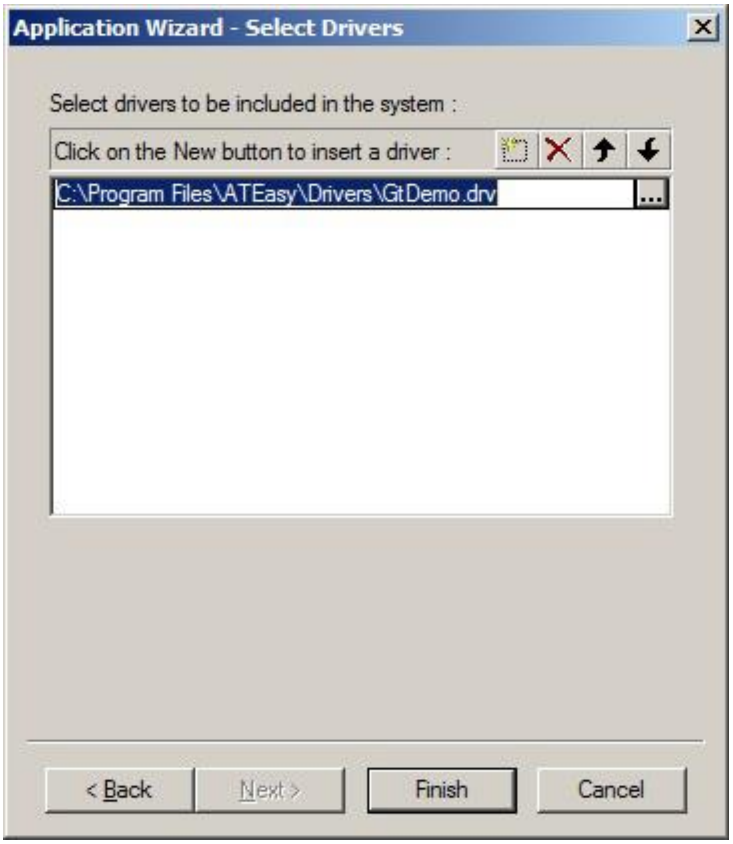
Type: ATEasy Driver
Size: 79.0 KB
Date modified: 9/6/2012 3:23 PM

File name: GtDemo

Files of type: Driver Files (*.drv; *.drt)

☐ Open as read-only

12. The GtDemo.drv driver should appear in the driver list. **Click Finish**.



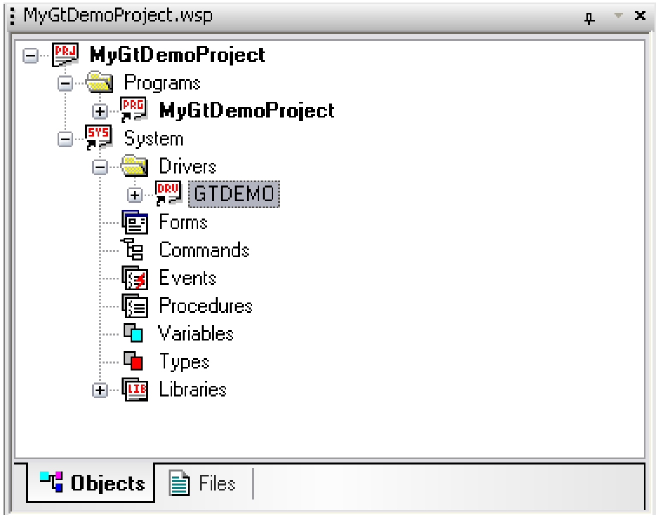
Application Wizard - Select Drivers

Select drivers to be included in the system :

Click on the New button to insert a driver :

C:\Program Files\ATEasy\Drivers\GtDemo.drv

13. Click **OK** after reviewing the Application Wizards summary page. The newly created project should look like this:



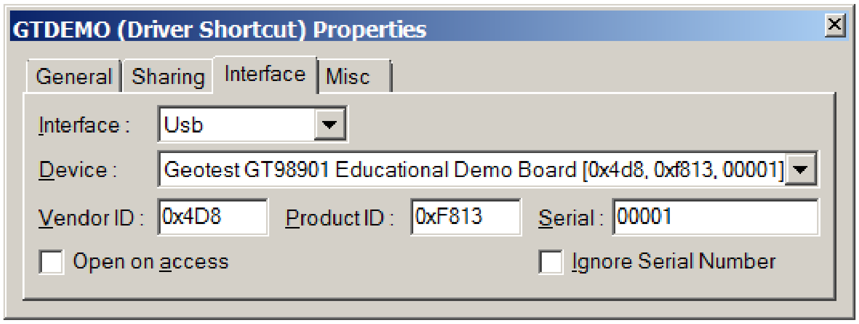
Configuring the Driver Shortcut Properties

It is important to know where to send instructions when you are communicating with instruments. When an instrument communicates through RS-232, it must first be attached one of the computer's COM ports. If you were to try communicating via TCP/IP, you would need the IP address of the instrument so your computer knows where to send the instructions. An ATEasy driver contains all of the instructions that need to be sent to the instrument, but the test engineer must configure each instance of ATEasy driver direct the communication. Consider that a system will sometimes contain two or more instruments that are the same model. Since a single GT98901 only needs one USB port, you can have as many GT98901s installed in one system as you have USB ports.

The Driver Shortcut Properties window allows you to configure an ATEasy driver to specify a target device. In the Interfaces page of the property window, you can select an interface type (COM, USB, etc.) More information on the Driver Shortcut Properties can be found in *Chapter 9 – Drivers and Interfaces*.

Make sure that your GT98901 is plugged into your computer and installed properly before continuing with the following.

1. Right-click on the GTDEMO driver and select **Properties** to open the GTDEMO (Driver Shortcut) Properties window.
2. In the Properties editor, click on the **Interface** tab.
3. Change the Interface type from **None** to **Usb**.
4. Click the Device dropbox and select the item labeled Geotest GT98901 Educational Demo Board or USBTMC or USB Test and Measurement Class.



Interface : Usb

Device : Geotest GT98901 Educational Demo Board [0x4d8, 0xf813, 00001]

Vendor ID : 0x4D8 Product ID : 0xF813 Serial : 00001

□ Open on access □ Ignore Serial Number

Reviewing a Driver's Command List

Once a driver has been added into the active project, its commands are added to the Insert menu for easy review and insertion.

Expand the **Insert** menu | **Driver Command** | **GTDEMO** to get a view of the commands available with this driver. The commands in the driver are organized by functional group. DISPLAY contains commands that clear and write to the LCD display. DIGITAL contains commands to control the digital input/output ports. For this exercise we are concerned with the ANALOG group which contains commands that manipulate the analog input and output functions.

Take some time to review the list of commands in the ANALOG group. Inside the ANALOG group, you should note that the commands are divided into INPUT and OUTPUT. Within the INPUT subgroup, you can read a single value or read an array of values using the READ SINGLE or READ ARRAY commands. Clicking on a command will insert it into the code editor at the location of the cursor.

Writing Test Code

We will now write a test which measures the voltage generated from the DACs. Specifically, we will set the voltage of the DACs to 2.2 and 5.5 volts and then measure the voltages from the ADCs.

▼ Rename the Task and Tests:

1. Rename Untitled Task to **DAC Tests**. Do the same for the Untitled Test and rename it to **AOut1**.
2. Right-click the Test AOut1 and select **Insert Test After**.
3. Rename the newly created test AOut2.

▼ Set the Tests' Properties:

4. Enter the following properties for AOut1:

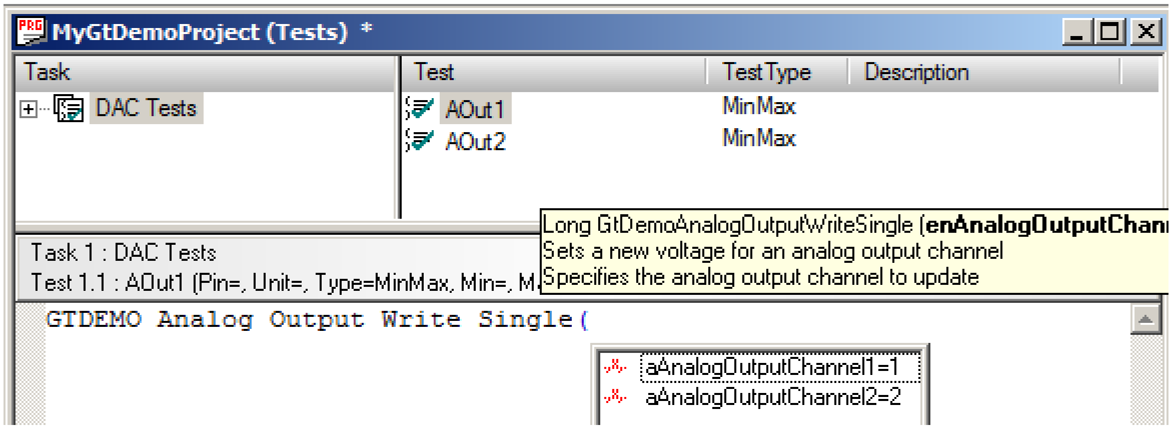
Pin	**AIn1**
Unit	**Volt** (you either type this or select from the combo box list)
Min	**2.0**
Max	**2.4**

5. Select the AOut2 test, and enter the following values in AOut2 Test Properties:

Pin	**AIn2**
Unit	**Volt**
Min	**5.0**
Max	**6.0**

▼ Write the test code:

6. Double-click the test AOut1. This will move the cursor to the code editor for the AOut1 test.
7. Use the insert menu to insert a command. Select **Driver Command | GTDEMO | Analog | Output | Write | Single**. This inserts the code into the code editor and if the command requires parameters, a suggestion box will open.



Task
DAC Tests

Test	Test Type	Description
AOut1	MinMax	
AOut2	MinMax	

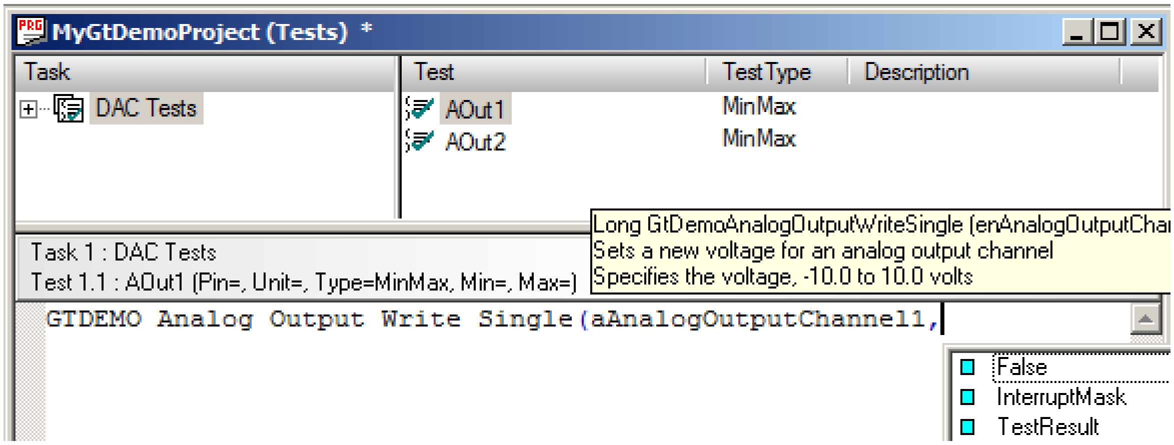
Task 1 : DAC Tests
Test 1.1 : AOut1 (Pin=, Unit=, Type=MinMax, Min=, M

Long GtDemoAnalogOutputWriteSingle (**enAnalogOutputChan**
Sets a new voltage for an analog output channel
Specifies the analog output channel to update

```
GTDEMO Analog Output Write Single(
```

aAnalogOutputChannel1=1
aAnalogOutputChannel2=2

8. The first parameter that you are adding is the analog output channel that you want to update. Highlight **aAnalogOutputChannel1** and hit enter to insert. Press the comma button (**,**) to view the next parameter.



Task	Test	TestType	Description
DAC Tests	AOut1	MinMax	
	AOut2	MinMax	

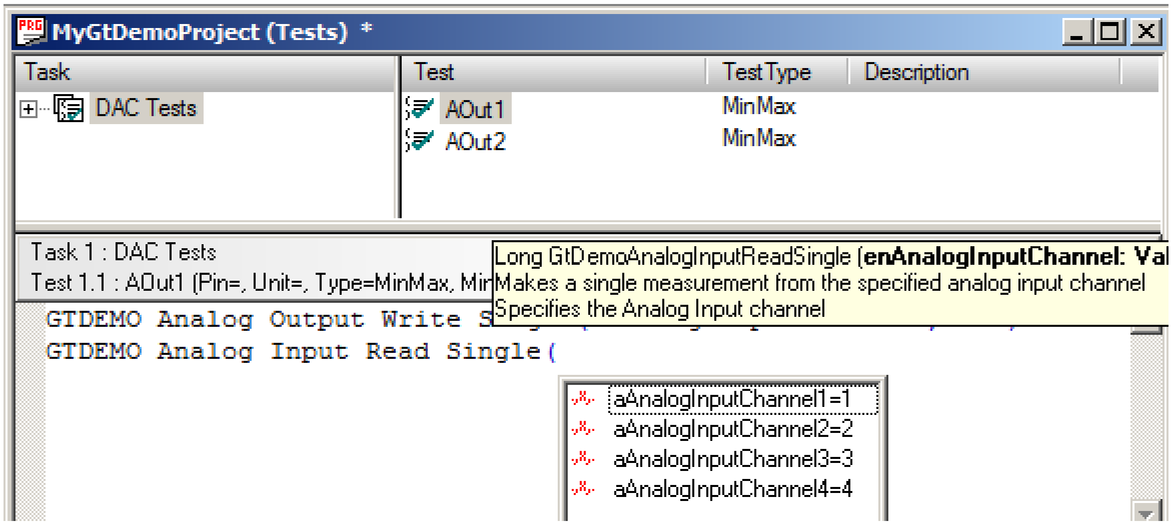
Task 1 : DAC Tests
Test 1.1 : AOut1 (Pin=, Unit=, Type=MinMax, Min=, Max=)

Long GtDemoAnalogOutputWriteSingle (enAnalogOutputChai
Sets a new voltage for an analog output channel
Specifies the voltage, -10.0 to 10.0 volts

```
GTDEMO Analog Output Write Single(aAnalogOutputChannel1,
```

9. The second parameter to add is the new voltage. Type **2.2** and a closed parenthesis to complete the command.

10. Use the insert menu to insert a second command. Select **Driver Command** | **GTDEMO** | **Analog** | **Input** | **Read** | **Single**.



Task	Test	TestType	Description
DAC Tests	AOut1	MinMax	
	AOut2	MinMax	

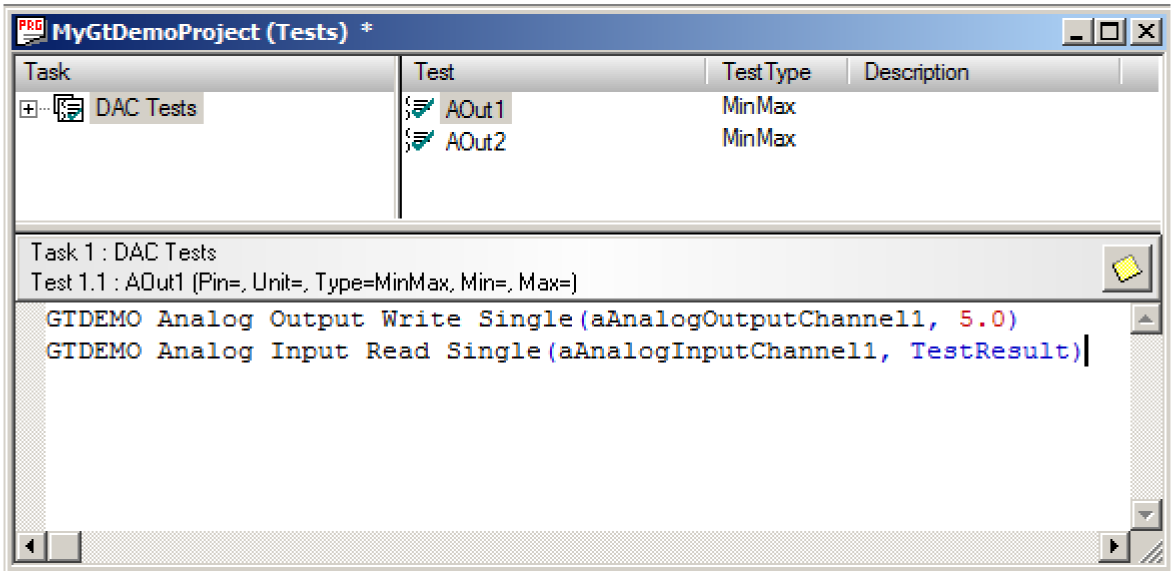
Task 1 : DAC Tests
Test 1.1 : AOut1 (Pin=, Unit=, Type=MinMax, Mir

Long GtDemoAnalogInputReadSingle (**enAnalogInputChannel: Val**
Makes a single measurement from the specified analog input channel
Specifies the Analog Input channel

```
GTDEMO Analog Output Write S
GTDEMO Analog Input Read Single(
```

11. The first parameter that you are adding is the analog input channel that you want to read from. Highlight **aAnalogInputChannel1** and hit enter to insert. Press the comma button (**,**) to view the next parameter.

12. The second parameter will be the variable that you want to save the voltage that was read from the specified input channel. Type **TestResult** and a closed parenthesis to complete the command.



Task 1 : DAC Tests
Test 1.1 : AOut1 (Pin=, Unit=, Type=MinMax, Min=, Max=)

```
GTDEMO Analog Output Write Single(aAnalogOutputChannel1, 5.0)
GTDEMO Analog Input Read Single(aAnalogInputChannel1, TestResult)
```

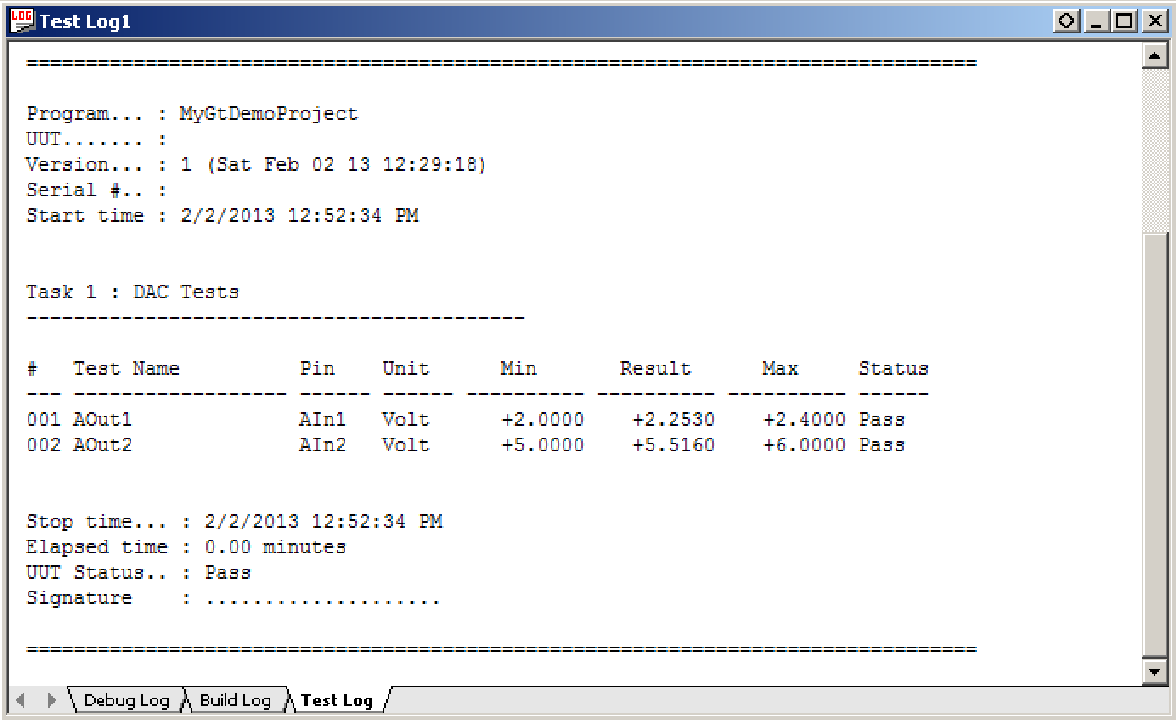
13. Double-click the second test **AOut2**. In the code editor, insert or type the following code:

```
GTDEMO Analog Output Write Single(aAnalogOutputchannel2, 5.5)
GTDEMO Analog Input Read Single(aAnalogOutputchannel1, TestResult)
```

▼ Review the Results:

14. Ensure that **AOut1** is wired to **AIn1** and **AOut2** is wired to **AIn2**.

15. **Start** the application. Observe the results in the test log. Both tests should have passed. If they have not, check your wiring, your test code and your PASS/FAIL criteria from the Test Properties editor.



```
================================================================================

Program... : MyGtDemoProject
UUT....... :
Version... : 1 (Sat Feb 02 13 12:29:18)
Serial #.. :
Start time : 2/2/2013 12:52:34 PM


Task 1 : DAC Tests
------------------------------------------


#   Test Name          Pin    Unit      Min        Result      Max      Status
--- ------------------ ------ ------ ---------- ---------- ---------- ------
001 AOut1              AIn1   Volt      +2.0000    +2.2530    +2.4000 Pass
002 AOut2              AIn2   Volt      +5.0000    +5.5160    +6.0000 Pass


Stop time... : 2/2/2013 12:52:34 PM
Elapsed time : 0.00 minutes
UUT Status.. : Pass
Signature    : ...................

================================================================================
```

Chapter 8 - Variables and Procedures

About Variables and Procedures

This chapter discusses how to create and use ***ATEasy*** variables, data types, and procedures. You will declare two program variables: **lIndex**, **adSamples**. The first one will be used as a loop counter, while the second will be defined as an array. It will contain values to be passed to a procedure named **Average**. You will write this procedure to calculate the average of an array.

You will also learn about ***ATEasy*** statements and the internal library. Use the table below to learn more about this chapter’s topics.

Topic	Description
Variables and Data Types	About the variables and data types supported by ***ATEasy***.
Variables Naming Conventions	Guidelines for naming variables and how to declare a variable.
Declaring Variables	How to declare a Variable.
Variable Properties	What Variable Properties are and how to set them.
Procedures	What Procedures are.
Creating a Procedure	How to create a procedure.
Procedure Properties	What Procedure Properties are and how to set them.
Procedure Parameters and Local Variables	Guidelines to procedures’ variables and parameters, how to create them, and how to write procedure code.
Calling the Procedure from a Test	How to use (call) a procedure from an ***ATEasy*** program.
Debugging Your Code	About ***ATEasy*** debugging tools and how to use them.
More About Writing Code	Additional information regarding writing code.
The Internal Library	What the internal library is and procedures it includes.

Variables and Data Types

A variable is an area in the computer memory used to store data of a specified type. A data type defines the type, range, and size of value or values that can be stored in a variable. ***ATEasy*** has a wide variety of many basic data types built into the language. These data types are divided into the following groups:

- **Signed Integer numbers**: **Char**, **Short**, **Long** and **DLong** for 1, 2, 4, and 8 bytes integers that can contain positive or negative values.
- **Unsigned Integers numbers**: **Byte**, **Word**, **DWord** and **DDWord** for 1, 2, 4 and 8 bytes integer that can contain only positive values.
- **Floating point numbers**: **Float** is 4 bytes and **Double** is 8 bytes floating point data type.
- **Character strings**: **Strings** can be defined for fixed or variable length strings used to hold ASCII characters. Each character is based on the **Char** data type. **BString** is used to hold Unicode characters based on the Windows internal data type that is used to communicate with COM objects. Each character is based on the **WChar** data type where a single Unicode character is stored in 2 bytes.
- **Object** data type that is used to hold instances of COM or .NET classes or controls. Object data type can be created from ***ATEasy*** internal library (that is COM based) or using external library. See chapter **Error! Reference source not found.** for more information.
- **Misc. data types**: including: **Bool** – which can hold two values, either True (-1) or False (0); **Variant** – which its internal data type can changed dynamically as you assign values of different types; and **Procedure** – which is used to hold an address of a procedure. Other available data types are: **Currency** and **DataTime** which are data types that are used sometimes for communication with external libraries.

ATEasy also supports user-defined data type. These include **Struct**, **Enum** and **Typedef**. **Struct** defines new data that contains several fields, each with its own data type grouped together defined as one data type. **Enum** defines one or more integer values each with its own name and **Typedef**, which provides a user-defined name or alias to another type name.

Variable Naming Conventions

ATEasy uses prefixes for naming variables in its internal library. While these are not requirements when creating variables, we recommend these conventions be used whenever a variable is declared. Prefixes are based on the variable data type and used to identify the data type without checking how it was defined. This expedites the debug and maintenance process as well as the coding standard that is important when multiple users are working or debugging the same module.

The recommended format of variables names is shown below:

[*scope*] [*pointer*] [*array*] [*Type*] *VariableBaseName*

Where:

- *Scope* refers to public (**g_**) or non-public (**m_**) driver or system variables. Procedure or program variables do not have a scope prefix.
- *Pointer* refers to VAR parameters or pointers having the **p** prefix.
- *Array* indicates the variable is an array by using the **a** prefix.
- Type is one of the types as follows:

c for Char, **n** for Short, **l**, **i** or **j** for Long and **dl** for DLong (for example, lCount).

uc for Byte, **w** for Word, **dw** for DWord and ddw for DDWord(for example, dwMask).

f for Float, **d** for Double (for example, dResult).

s for String and **bs** for BString (for example, sText).

b for Bool, **v** for Variant, **cy** for Currency, **dt** for DateTime, **ob** for Object, **en** for Enum and **st** for Struct (for examples, bModified, cyTotalAmount, vKey…).

VariableBaseName refers to the name of the variable. This should be one or more words with no spaces or underlines between them. Each word starts with upper case characters and continues with lower case characters (for example, nNumOfSamples)

As an example, the following variable is a public module array. Each element in the array has a type of Double:

```
g_adSampleResults
```

Other rules imposed by the ***ATEasy*** programming language for naming identifiers must be followed. These rules set the maximum number of characters for a name: 256 characters, the allowed characters for starting identifier: _, A-Z, a-z, and the allowed characters following the first character A-Z, a-z, 0-9, _.

Declaring Variables:

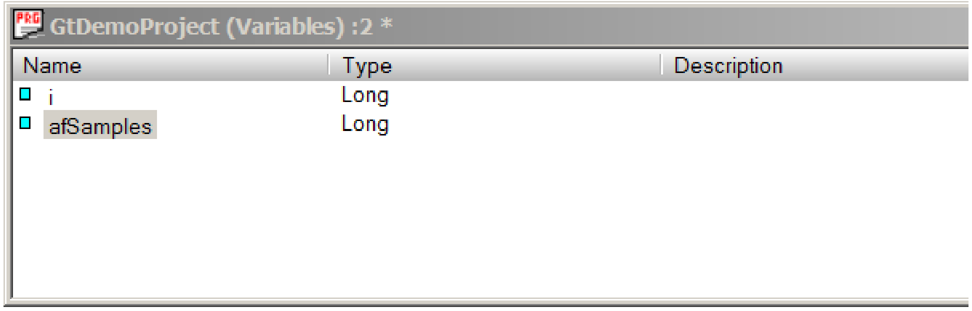
For ***ATEasy*** to recognize and use a variable, it must be declared first. When declaring a variable, you can determine its name and data type as well as give it a description for documentation purposes.

▼ To declare program module variables:

1. Double-click on the **Variables** submodule below MyProgram in the Workspace window. A Variables view opens displaying three columns: Name, Type, and Description.
2. Right-click on the view and select **Insert Variable At** from the context menu. A new variable is created and displayed in the view. An edit box displays allowing you to rename the variable name.
3. Type the name: **i**.

4. Right-click on the **i** variable and select **Insert Variable After** . A new variable is created and inserted after i. Rename it by typing **afSamples**.

5. Your screen should now look similar to the following:



GtDemoProject (Variables) :2 *

Name	Type	Description
i	Long	
afSamples	Long	

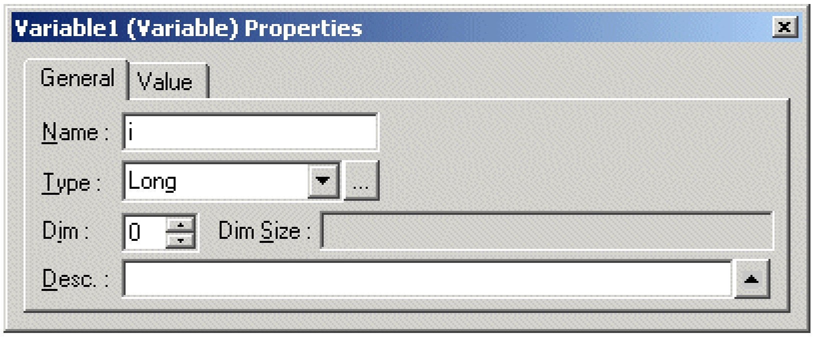
Variable Properties

The next step is to set the properties of variables that you defined. The first variable, **i**, is declared as Long. The second variable, **afSamples**, will be declared as a one-dimensional array holding 40 elements of type Float.

▼ To set the properties of the variables:

1. Right-click on the variable and select **Properties** . Alternately, you can double-click on the variable icon, or select the variable and choose **Properties** from the **View** menu or from the Standard toolbar.

   The Variable properties window displays as shown below:



Variable1 (Variable) Properties

Name : i

Type : Long

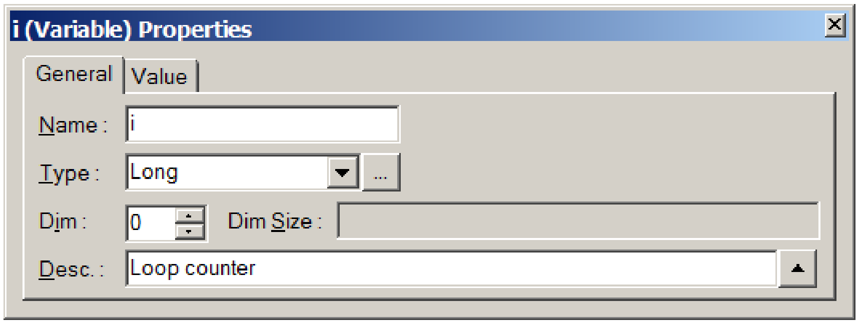
Dim : 0

Dim Size :

Desc. :

The properties of a variable include **Name**, **Type**, array dimension (**Dim**) and size (**Dim Size**), description (**Desc**) , and **Public** (The **Public** does not show here since it is not applicable for program variables). The **Public** property indicates whether this variable can be used from other modules. The **Value** property page contains the variable's initial value, and whether the variable is constant and cannot be changed programmatically (the **const** property).

2. Click in the **Desc** field and type **Loop Counter** for the description.



i (Variable) Properties

Name : i

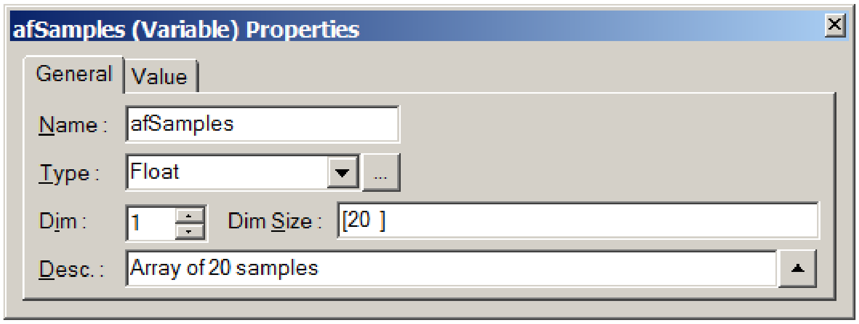
Type : Long

Dim : 0 Dim Size :

Desc. : Loop counter

3. Leave the Properties window open and select the next variable **afSamples**. Note that you do not have to close the Properties window; the properties window updates and displays the object you selected. Repeat steps 1-2 for **afSamples** and type **Array of 20 samples** in the description field.

4. Click the drop down arrow next to the Type combo box and select **Float** as the type. Set the **Dim** to **1**, the **Dim Size** to [**20**].



afSamples (Variable) Properties

Name : afSamples

Type : Float

Dim : 1 Dim Size : [20]

Desc. : Array of 20 samples

Procedures

A **Procedure** is a set of command instructions that can be executed at run-time as one unit. A Procedure that returns a value is called a **function**. A Procedure that does not is called a **subroutine**. Procedures typically contain code used multiple times throughout the application. By using procedures, the total code is reduced, improving code reuse and maintenance of your application.

Procedures typically have a **name**, **parameters**, **local variables**, and **code**. The *name* is used when calling the procedure. *Parameters* are variables used to pass arguments containing values from the caller to the procedure. *Code* is programming statements included in the procedure, and *local variables* are used within the code if intermediate values need to be stored in the procedure while it is executing.

Several types of procedures exist in ***ATEasy***:

User Procedures – These procedures are defined under a **Procedures** submodule or in a **Form Procedures** submodule. User procedures contain code written by the user. The code may contain calls to other procedures or even to the current procedure (recursive call).

Events – Events are ***ATEasy*** procedures called by ***ATEasy*** when an event occurs. Two types of events are available in ***ATEasy***: Module events and Form events. *Module events* are called to notify the module that a certain event occurred in the application, for example, **OnAbort** is called when the application is aborted. ATEasy calls *form events* because of user interaction with a form, menu, or control. Examples of form events include **OnClick**, **OnMouseMove**, and more. Events names and parameters are pre-defined by ***ATEasy*** and cannot be changed by the user.

IO Tables – These are procedures used to send or receive from a device or instrument using an ***ATEasy*** interface such as GPIB. An I/O table does not contain code; rather; it contains I/O operations.

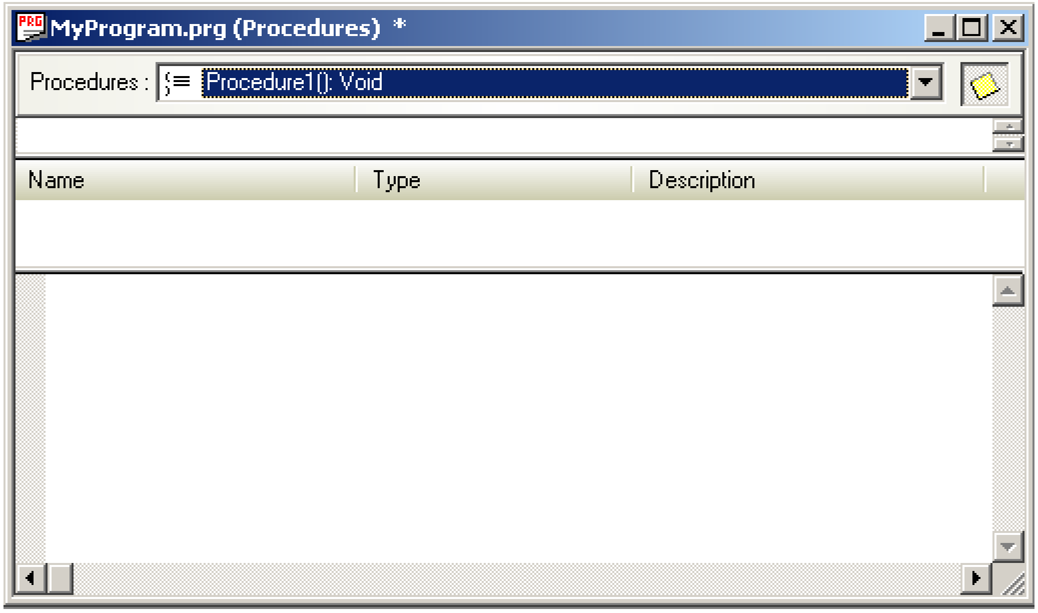
DLL – These are procedures residing in, and exported from, an external library (DLL). You can define and call them in ***ATEasy***.

Type library/COM or .NET methods and procedures – These are similar to DLL procedures as they reside in an external library. They are defined automatically when you import the COM based Type Library or .NET assembly describing these procedures.

Creating a Procedure

▼ To create a program module procedure:

1. Select **Procedures** from the Tree View.
2. Right click on Procedures and select **Insert Procedure Below** on the context menu. The Procedures View opens displaying as shown here:



The procedures view display the module procedures in a combo box on the top of the view. Below it is an area where the procedure description can be entered. The procedure parameters and variables area follows this and the procedure code area is displayed at the bottom of the view.

The procedures combo box displays the available procedures and is used to select the **current procedure**. The other areas display the current procedure description, variables, and code.

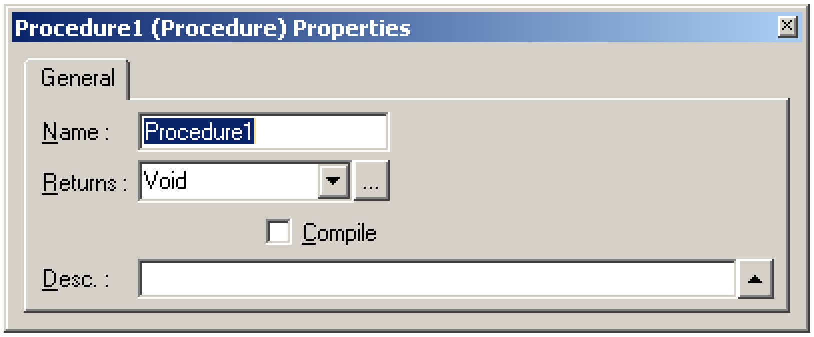
Procedure1 is the newly created procedure. It is the current procedure. The return type was set to **Void** indicating the procedure does not return any value and is therefore a **subroutine**.

Procedure Properties

In our example, you will be creating a procedure to calculate the average value of an array containing floating-point numbers.

▼ To change the properties of the procedure:

1. Select **Properties** from the **View** menu or click the Properties Window tool on the Standard toolbar. The procedure's properties window appears:



Name : Procedure1

Returns : Void

□ Compile

Desc. :

The properties of a procedure include **Name**, Return value type (**Returns**), Description (**Desc**.), and **Public** (the **Public** does not show here since it is not applicable for program procedures) indicating whether the procedure can be called or used by other modules.

2. Change the procedure name to **CalculateRMS**.
3. Select or type **Double** from the Returns combo box to set the return value.
4. Type the following description: Calculates the root mean square of the values of an array.

At this point, the procedure properties are defined. You now need to declare the procedure parameters, variables and write the procedure code.

The Compile flag (checkbox "Compile") is used to force ***ATEasy*** to compile and include this procedure during build of the target file (EXE or DLL). Normally, only procedures that are called or referenced from a test or a procedure will be included and compiled during the build in the target file.

Procedure Parameters and Local Variables

Variables used by procedures can be local variables, module variables, or parameters. Parameters are used when calling the procedure in order to pass data and variables to the procedure. The caller passes the arguments to the procedure parameters in the same order they were defined as parameters. The procedure then uses the parameters to perform calculations or any other tasks it needs to perform.

Two types of parameters are available in ***ATEasy***: **Val** parameters and **Var** parameters. A *Val* parameter receives its initial value from the argument passed to the procedure. It can be changed by the procedure; however, that change is reflected only in the procedure while the value of the argument is not changed. The Var parameters hold the address of the argument variable passed to the procedure. Any change to the parameter will be reflected in the argument variable.

Local variables are created each time the procedure is called and their initial value is set before the procedure code is executed.

Your procedure, **CalculateRMS,** has two parameters: **pafData,** used to receive the array, and **lSize** used to tell the procedure how many elements of the array are needed to calculate the average.

Additional procedure variables will be needed to perform the average calculation. These are: **d** to hold the sum of the array elements and **i** to be the loop counter. The loop counter is used to iterate through the array in order to sum the value of all array elements.

▼ To create the procedure variables and parameters:

1. Right-click in the variables pane of the procedure view. This is the pane with the headings **Name**, **Type**, and **Description**. It appears below the procedure description pane. Select **Insert Parameter/Variable After** . A new variable named **Variable1** is inserted.
2. Rename it to **pafData** by typing or pressing **F2** (if not already in edit mode) and typing.
3. Repeat steps one and two and define the following variables: **lSize**, **d**, and **i**. By now, you should have four variables defined as Val Long.
4. Right-click on the **pafData** parameter and select **Properties** . Set the following properties:

Name:	**pafData**
Parameter:	**Val**
Type:	**Float**
Dim:	**1**
Desc.:	**Array to calc RMS**

1. Repeat step four for the **lSize** parameter as follows:

   Name: lSize
   Parameter: Val
   Type: Long
   Dim: 0
   Desc.: Number of elements, use the whole array if omitted

   Check the **Optional** check box. This allows the user to pass or not pass an argument here.

   Set the initial value from the Value property page for this parameter as –1. If the caller will not provide this argument the value of **lSize** will be –1.

2. Repeat step four for **d**, the local variable as follows:

   Name: **d**
   Parameter: **None**
   Type: **Double**
   Dim: **0**
   Desc.: **Sum of elements**

3. Repeat step four for **i**, the local variable as follows:

   Name: **i**
   Parameter: **None**
   Type: **Long**
   Dim: **0**
   Desc.: **Loop Counter**

Writing the Procedure Code

You are now ready to start writing the code. To average a group of numbers, you need to sum the elements of the array (ad) and then divide the total by the number of elements (lSize).

▼ To write the procedure code:

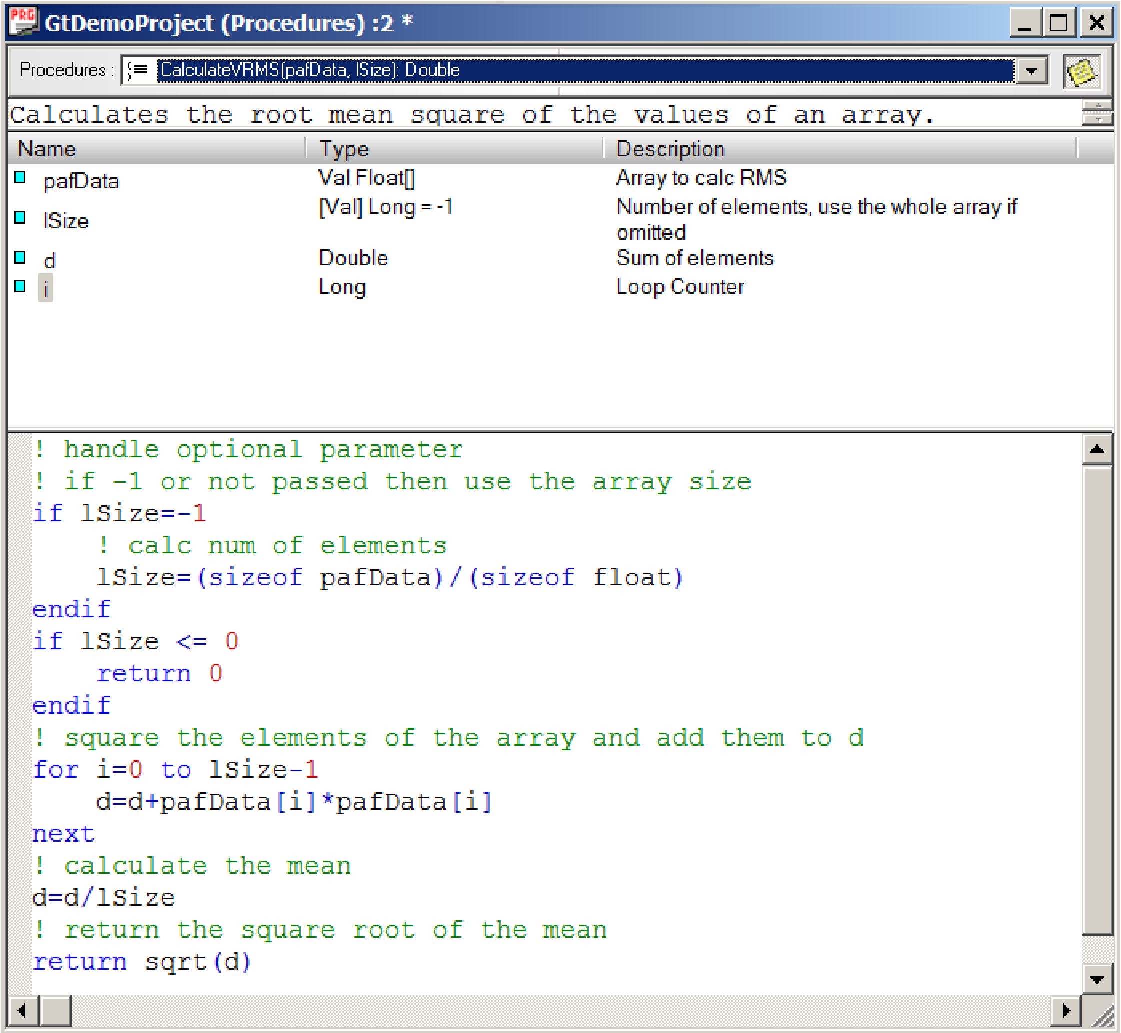
1. Type the following statements in the procedure code area:

```
! handle optional parameter
! if -1 or not passed then use the array size
if lSize=-1
   ! calc num of elements
   lSize=(sizeof pafData)/(sizeof float)
endif
if lSize<=0
   return 0
endif
! square the elements of the array and add them to d
for i=0 to lSize-1 do
d=d+pafData[i]*pafData[i]
next
! calculate the mean
d=d/lSize
! return the square root of the mean
return sqrt(d)
```

   The first two statements determine how many array elements to calculate the average (lSize), followed by a For-Next loop starting with 0 and ending with the last element you want to average (lSize-1). d is used to store the sum of all elements. The last statement returns the average (total divided by number of elements) to the caller.

2. To verify the code was typed correctly and that you do not have syntax errors, click on the **Checkit**! on the **Build/Run** toolbar. If no errors are found, the status bar, shown at the bottom of the main window, should display **No Errors**. Otherwise, the cursor will move to the place where the error occurred and the status bar will show a description of the compiler error, in this case, fix the error and repeat this until **No Errors** displays in the status bar.

The Procedure View should now look as illustrated in the figure below:


GtDemoProject (Procedures) :2 *
Procedures :
CalculateVRMS(pafData, lSize): Double

Calculates the root mean square of the values of an array.

Name	Type	Description
pafData	Val Float[]	Array to calc RMS
lSize	[Val] Long = -1	Number of elements, use the whole array if omitted
d	Double	Sum of elements
i	Long	Loop Counter

```
! handle optional parameter
! if -1 or not passed then use the array size
if lSize=-1
    ! calc num of elements
    lSize=(sizeof pafData)/(sizeof float)
endif
if lSize <= 0
    return 0
endif
! square the elements of the array and add them to d
for i=0 to lSize-1
    d=d+pafData[i]*pafData[i]
next
! calculate the mean
d=d/lSize
! return the square root of the mean
return sqrt(d)
```

Calling the Procedure from a Test

To see if the RMS calculating procedure you just created works, create a test. In the test, assign five elements to an array with values ranging from 1 to 5. Then, call the CalculateRMS procedure. The procedure should return the square root of 11 (approximately 3.3), which is the RMS of the filled array. Use a tolerance test type with a value of 3.3 plus or minus 0.1.

▼ To write a test using the CalculateRMS procedure:

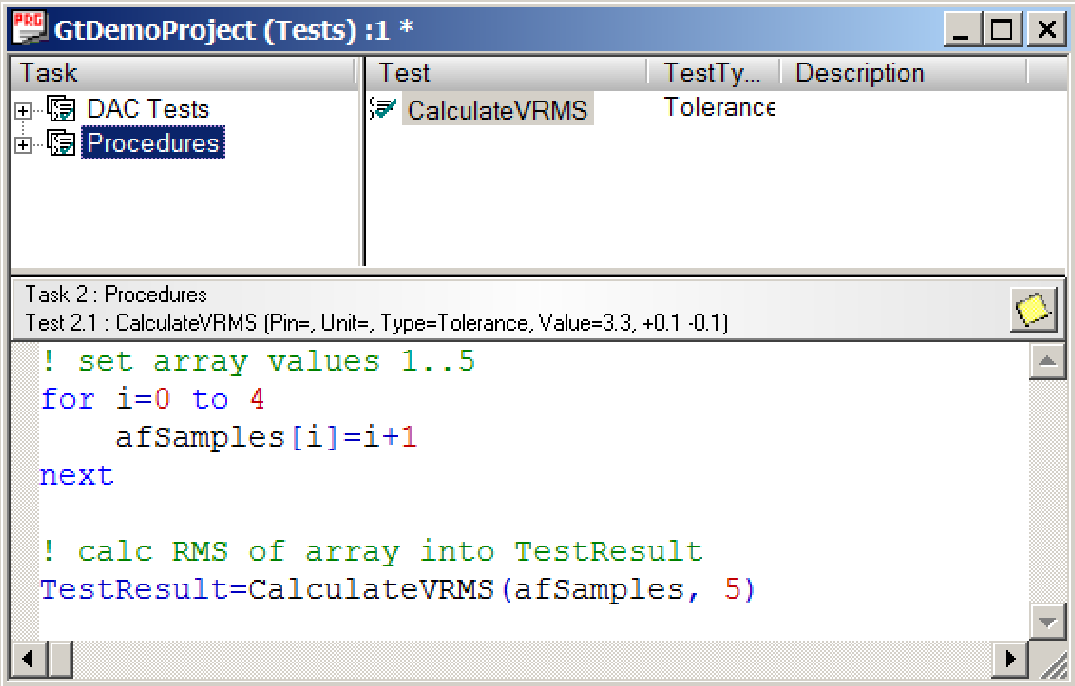
3. Double-click on the **Tests** submodule below the **GtDemoProject** program module in the Workspace window. The tests view should appear in a new document view.
4. Right-click on the Power Tests task, and select **Insert Task After** . A new task with a test is inserted.
5. Rename the task to **Procedures** by typing in the edit box or pressing **F2** and typing.
6. Rename the **Untitled Test** to **CalculateRMS** by clicking on the Untitled Test text and typing **CalculateRMS**.
7. Right-click on CalculateRMS and select **Properties** . Change the test type to **Tolerance,** set the value to be **3.3**, the plus to **0.1** and the minus to **0.1**.
8. Now enter the following code in the code pane:

```
! set array values 1..5
for i=0 to 4
   adSamples[i]=i+1
next
! calc RMS of array into TestResult
TestResult=CalculateRMS(afSamples, 5)
```

The first three statements in this example are a simple For-Next loop filling the array, **afSamples**. The next line calls the CalculateRMS procedure and sets the result to be the test result. Note that if you called the CalculateVRMS procedure by omitting the second optional argument (**TestResult=CalculateRMS(afSamples)**), this called would have returned the wrong value since the array is size 20 and most of the array is filled with the value 0.0.

At run-time, after the test is completed, ***ATEasy*** will use the TestResult variable to determine if the test passed or failed.

The program should now look as illustrated in the figure below:



Task 2 : Procedures
Test 2.1 : CalculateVRMS (Pin=, Unit=, Type=Tolerance, Value=3.3, +0.1 -0.1)

```
! set array values 1..5
for i=0 to 4
    afSamples[i]=i+1
next

! calc RMS of array into TestResult
TestResult=CalculateVRMS(afSamples, 5)
```

You can test your code by selecting the **Testit!** command from the **Debug** menu. This command will run only this test. After the test runs, take a look at the test log and verify that the test you just wrote has a “Pass” status.

In the next section, you will learn several ways to debug your code.

Debugging Your Code

ATEasy provides extensive tools to allow you to debug your code. These include the following commands:

Continue / **Pause** (F4) – continues or pauses the debugged application.

Abort (ALT+F5) – aborts the debugged application.

Doit! (CTRL+D) – executes the current code view selection. If no code is selected, the whole content of the code view is executed. The command is available only when the current view is code view (in the tests view or procedures view).

Step Into (F8) – allows you to execute your code line by line. Step Into executes the current line and pauses. If the line is an ***ATEasy*** procedure, ***ATEasy*** pauses before executing the first line in the procedure.

Step Over (F10) – is similar to Step Into, however, if the current line is a procedure ***ATEasy*** executes the procedure as a unit and pauses after the procedure is returned.

Step Out – executes the remaining code of the current procedure and pauses at the next statement following the procedure call.

Toggle Breakpoint (F9) – sets or removes a special mark in your code to tell the debugger to pause before executing the code.

Run to Cursor – sets a temporary breakpoint at the current insertion line and then continues execution.

Several debugging windows are also available. These let you watch the value of the application variables during execution. These following debugging windows are available:

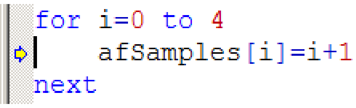
- **Call Stack/Locals** – displays variables values of modules variables and procedures variables when the application is paused. The user can change values of variables.
- **Watch** – allows you to type expressions in order to evaluate their value. ***ATEasy*** calculates and displays the value of the expression every time the execution pauses.

Other debugging commands and windows are available from the **Run** and **Debug** menus and the **View** menus.

ATEasy contains two execution modes when executing code from the IDE. You can select lines from the code view and execute them – this is called **Selection Run Mode.** Alternatively, you can execute the application or a portion of your application (e.g. a test). This is called **Application Run Mode**. You can start debugging using Selection Run Mode when the active view (the view with the input focus) is the code view. Use the **Doit!**, **Loopit!**, **Formit!**, and step commands. If the active view is not a code view, run mode is always used.

▼ To use Selection Run Mode for debugging:

1. Activate the CalculateRMS test code view by clicking on the test code view.
2. Select the **Step Over** command from the **Debug** menu. ***ATEasy*** executes the code in the test. Since no code was selected, all the test code in the view will be compiled and scheduled for execution. ***ATEasy*** will pause before starting the execution and the code view mark area (the left bar) displays a yellow arrow showing where the execution paused. This is the Next Statement Mark as shown here:

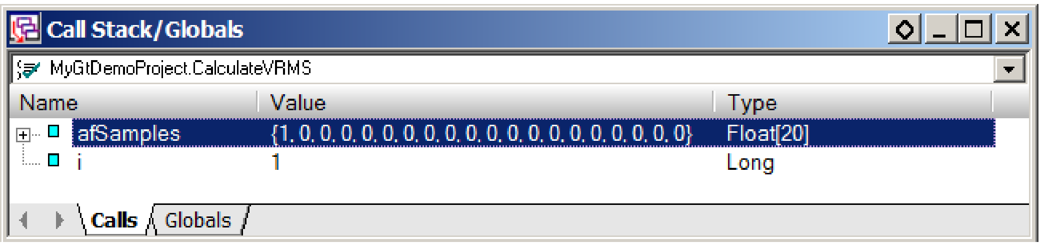


```
for i=0 to 4
    afSamples[i]=i+1
next
```

3. Click **Step Over**. At this point, the next statement advances to the assignment statement.

4. Select **Call Stack** from the **View** menu. The Call Stack window is displayed. Notice the current module variables and their values are displayed. **i** should be zero and the array **afSamples** elements should all be zeros.

5. Click **Step Over**. At this point, the next statement advances to the assignment statement. The first element of the array should be set to **1** as shown here:



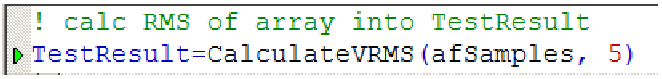
Name	Value	Type
afSamples	{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}	Float[20]
i	1	Long

You can repeat this step to see how the value of the program variables changes as you step through the code.

6. Set the insertion point to the line containing the call to the CalculateRMS procedure. Select **Run to Cursor** from the **Debug** menu. ***ATEasy*** continues the loop and pauses before calling the procedure; filling the array elements values from 1 to 5. You can expand the array in the Call Stack window to see the array elements by clicking on the **+** sign next to the array.

7. Select **Step Into** from the **Debug** menu. A new document view will be displayed showing the Average procedure code. Note also, the Call Stack window now displays the procedure variables. The combo box displaying the call stack chain in that window shows two items: the top one is the CalculateRMS procedure and the second one the CalculateRMS test.

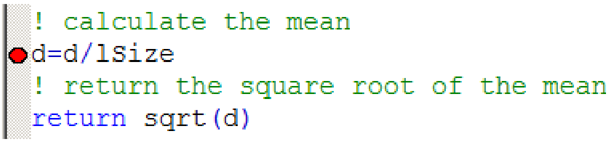
8. Select the second item in the Call Stack combo box. The Test is shown. Notice the green arrow mark next to the line that called your procedure. This mark is the **Call Mark** and it shows the line that called your procedure as shown here:



```
! calc RMS of array into TestResult
TestResult=CalculateVRMS(afSamples, 5)
```

To display the next statement, select **Show Next Statement** from the **Debug** menu. The average is displayed again.

9. Set the insertion point in the line containing the division of **d** with **lSize**. Select the **Toggle Breakpoint** command from the **Debug** Menu or from the Standard toolbar. A red **Breakpoint Mark** will appear next to the line as shown here:

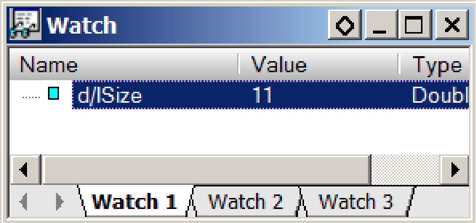


```
! calculate the mean
d=d/lSize
! return the square root of the mean
return sqrt(d)
```

10. Select **Continue** from the **Run** menu. The debugger stops where you placed your breakpoint. At this point, you can examine the value of **d** in the Call Stack window.

11. Open the Watch window by selecting **Watch** from the **View** menu. The Watch window will show.

    Right-click on the view, and select the **Insert Object At** . Type **d/lSize**. The value displayed should be **11** as shown here:



Name	Value	Type
d/lSize	11	Doubl

12. To complete this debug session select **Continue** or **Abort** from the **Run** menu.

Writing Procedures for the GT98901

As a test engineer, you will be called upon to apply both apply stimulus to devices and to and to make measurements and analyze the results. Since the GT98901 is capable of both waveform generation and waveform sampling and digitizing, we can loop the analog output back to the analog input and test both features simultaneously. In this section, we will write functions to evaluate the RMS voltage of common wave types such as the sinusoidal, triangular and square waves.

▼ Create new variables to support the new tests:

1. Create the following program variables:

   Name: **afInput**
   Parameter: **Val**
   Type: **Float**
   Dim: **1**
   Dim Size: **100**
   Desc: **Array of 100 samples**

▼ Create the procedure to generate a sine wave:

2. In the **GtDemoProject**'s program module, right-click on the **CalculateRMS** procedure and select **Insert Procedure After**. A new procedure is created.

3. Using the Properties window, change the name of the new procedure to **GenerateSineWave** and the description to **Populates an array with a sinusoidal waveform**.

4. Create the following procedure variables:

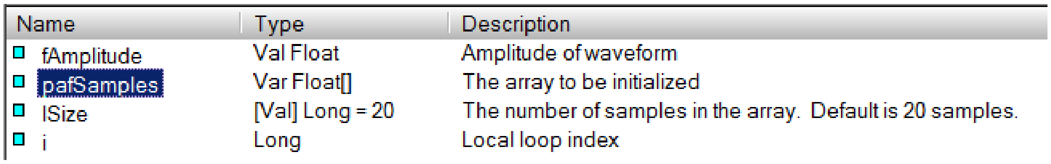
   Name: **fAmplitude**
   Parameter: **Val**
   Type**: Float**
   Dim**: 0**

   Name: **pafSamples**
   Parameter: **Var**
   Type: **Float**
   Dim: **1**

   Name: **lSize**
   Parameter: **[Val]**
   Type: **Long**
   Dim: **0**
   Default Value: **20**

   Name: **i**
   Parameter: **None**
   Type: **Long**
   Dim: **0**

Your screen should now look similar to the following:



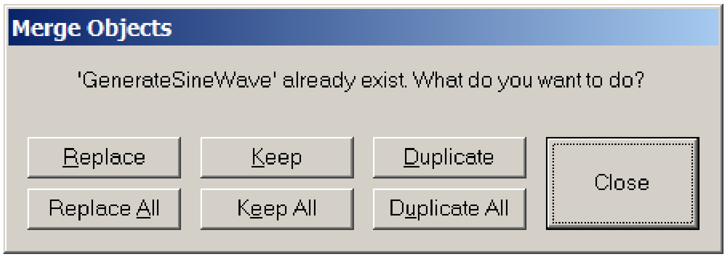
Name	Type	Description
□ fAmplitude	Val Float	Amplitude of waveform
□ pafSamples	Var Float[]	The array to be initialized
□ lSize	[Val] Long = 20	The number of samples in the array. Default is 20 samples.
□ i	Long	Local loop index

5. Add the following code in the procedure code area:

```
for i=0 to lSize-1
   pafSamples[i]=fAmplitude*sin((2*PI*i)/lSize)
next
```

▼ Create the procedure to generate a triangle wave:

6. In the **GtDemoProject**'s program module, right-click on the **GenerateSineWave** procedure and select **Copy**. Right-click **GenerateSineWave** again and select **Paste**. A dialog will pop up asking you what you are trying to do.



Merge Objects

'GenerateSineWave' already exist. What do you want to do?

7. Select **Duplicate**. A duplicate copy of GenerateSineWave will be created within the program module's procedures and named **GenerateSineWave1**.

8. Using the Properties window, change the name of the new procedure to **GenerateTriangleWave** and the description to **Populates an array with a triangular waveform**.

9. The variables are not different from the last procedure. Erase the code in the procedure's code editor and add the following code:

```
for i=0 to lSize-1
   pafSamples[i]=abs((2*fAmplitude*i/lSize)-fAmplitude)
next
```

▼ Create the procedure to generate a square wave:

10. In the **GtDemoProject**’s program module, duplicate **GenerateSineWave** procedure Rename the newly created procedure to **GenerateSquareWave** and change the description to **Populates an array with a square waveform.**

11. Leave the variables alone for **GenerateSquareWave**. Erase the code in the procedure’s code editor and add the following code:

```
! Load the low values
for i=0 to lSize/2
   pafSamples[i]=-1*fAmplitude
next
! Load the high values
for i=lSize/2 to lSize-1
   pafSamples[i]=fAmplitude
next
```

Testing the Procedures with the GT98901

We will now verify that the RMS calculation procedure you created works with a physical signal by creating tests that exercise the GT98901. We will be sourcing a sine wave from the GT98901's analog output ports and digitizing that signal which an analog input port.

▼ To write a test for the GenerateSineWave procedure:

1. Right-click on the CalculateRMS test, and select **Insert Test After**. A new test is inserted.
2. Rename the Untitled Test to GenerateSineWave.
3. The expected RMS for a sine wave is the amplitude divider by the square root of two. Right-click on GenerateSineWave and select **Properties** . Change the test type to **Tolerance,** set the value to be **3.5**, the plus to **0.3** and the minus to **0.3**.
4. Now enter the following code in the code pane:

```
!Generate a twenty sample sine wave
GenerateSineWave(5, afSamples, 20)
!Configure the GT98901 so the sine wave will output from AOut1 at
    26kHz
GTDEMO Analog Output Abort()
GTDEMO Analog Output Set ClockDivider(9)
GTDEMO Analog Output Write Array(aAnalogOutputChannel1,
   afSamples, 10)
!Begin generating the waveform
GTDEMO Analog Output Trigger()
!Begin digitizing data on AIn1
GTDEMO Analog Input Read Array(aAnalogInputChannel1, afInput, 100)
TestResult=CalculateRMS(afInput, 20)
```

   AIn1 should be wired to AOut1 for this test to PASS successfully. With the GenerateSineWave test in focus, call **TestIt!** from the **Debug** menu. The test log should show a PASS.

5. Repeat the process to find the RMS value of the sine wave by duplicating the GenerateSineWave test and replacing:

```
!Generate a twenty sample sine wave
GenerateSineWave(5, afSamples, 20)
```

   With the following for GenerateTriangleWave:

```
!Generate a twenty sample triangular wave
GenerateTriangleWave(5, afSamples, 20)
```

Or with the following for GenerateSquareWave:

```
!Generate a twenty sample square wave
GenerateSquareWave(5, afSamples, 20)
```

6. Adjust the properties of the GenerateTriangleWave and GenerateSquareWave tests with the new expected value. Remember that the expected RMS of a triangle wave is the amplitude divided by the square root of three and the expected RMS of a square wave is equal to the amplitude. Run the project by pressing **Start** in the **Run** menu and check the results.

More about Writing Code

ATEasy has a large number of options and features to help you when writing code:

You can insert flow control statements as a **for…next** statement by right-clicking on the code editor where you want the statement to start and selecting the **Insert Flow-Control Statement** from the context menu. This will insert text that can serve as a template for the statement; you will need to edit it to add the missing parts.

You can insert a symbol or a procedure call by right clicking on the code editor where you want the symbol to start and selecting the **Insert Symbol** command from the context menu. This shows a browser window with the available symbols that you can use from the current procedure or test.

You can use the ***ATEasy*** **Auto Type Information** feature to provide information regarding procedures, variables and other programming elements. You can move the mouse cursor on the programming element to see the syntax and a description of that symbol.

You can use the code completion options **Parameter Suggestion** and **Parameter Information** to suggest parameters for procedures when you type parameters. The syntax, type, and description of the parameter that you typed in will show in a small tool tip window next to the insertion point.

You can turn on the code **syntax highlighting** to color code the programming statement. This makes the code more readable and shows you which words are keywords, literals and more.

Other code completion features are available for using objects, structures and commands.

Additional resources explaining about the programming language elements and statements can be found in the on-line help and in the examples provided with ***ATEasy***.

The Internal Library

ATEasy's internal library is based on Microsoft's Component Object Model (COM) technology. This is a software architecture that allows software components made by different vendors to be combined into a variety of applications using different programming languages. COM allows you to describe programming components in **type libraries**. These libraries can be imported and used by programming environments such as ***ATEasy***. ***ATEasy*** is supplied with a type library called the **internal library**. The internal library contains the following components:

- **Classes** are objects containing data and procedures grouped together. Class data members are retrieved and set using **Properties**. These Properties can be considered as variables, which can be set or retrieved by using functions. Procedures in classes are called **Methods** and are used to perform actions on the object. Classes also contain **Events** that are called when the object notifies the application that a certain event has occurred. Examples of ***ATEasy*** classes are: the **ADriver** class providing access to driver properties set by the user at the design time, the **AForm** class providing a window, and more.

- **Controls** are classes adhering to specific COM standards to provide design and run-time behavior when placed on a form to provide a user interface component. Examples of the internal library controls are: the **AButton** control that displays a button and provides notification when the button is pressed (OnClick event), the **AChart** control used to display charts, and more.

- **Procedures** are used when writing code in procedures and tests. The internal library is supplied with a large number of procedures. The procedures are divided into groups that are called **Library Modules**. The internal library has modules used for mathematical calculations, string manipulation, file I/O, GPIB, VXI, serial communication, port I/O, DDE, and more.

- **Variables** provide the application a way to get, set, and perform actions on your application components. These include **TestResult** and **TestStatus** that provide a way to set the test result and test status; objects such as the **Test** object that provide a way to the application to get and set the current test properties; and more.

- **Types** are data types defined by ***ATEasy*** and used by the internal library classes, procedures, and variables. An example is the **enumATestStatus** providing the various constants for the **TestStatus** variable.

The internal library can be browsed under the Libraries submodule. You can expand the internal library components. You can retrieve help on any item that you see in the internal library by pressing the **F1** key.

Chapter 9 - Drivers and Interfaces

About Drivers and Interfaces

This chapter discusses how to create, add, configure, and use drivers in the system. You will learn how to configure interfaces such as the USB board; how to add interfaces to the driver; how to select the driver interface; and how to set the driver address in the system using the driver shortcut. In this chapter, you will use I/O Tables to send and receive data to a USB instrument such as the GT98901 Educational Demo Board. You can apply similar techniques when using a different instrument.

Use the table below to learn more about this chapter's topics:

Topic	Description
Interfaces and Interface Types	What are interfaces and what types of interfaces ATEasy supports.
Adding an Interface	How to add an interface.
Creating and Adding Drivers	How to add drivers to a system and how to create new drivers.
Driver and Driver Shortcut	Differences between the driver shortcut properties and the driver's properties.
Driver Default Name	How to define the default name of a driver.
Defining the Driver Interface	How to define a driver's interface.
Configuring the Driver in the System	How to configure the driver for an application.
I/O Tabless	What are I/O Tables?
Creating an I/O Table: DisplayClear	How to create an I/O Table.
Using the Output Discrete Mode	How to take advantage of discrete properties to reduce the number of I/O tables necessary in the system.
Creating an I/O Table: DisplaySetText	How to create an I/O that takes a parameter and converts to ASCII
Reading Data from the Instrument	How to create an I/O table to read data from an instrument.
Calling an I/O Table from a Test	How to call I/O Tables from a program.
Using the Monitor View	How to set up and turn on the Monitor View.
More I/O Tables	A list of I/O Tables to develop for practice.

Interfaces, Interface Types, and Drivers

Interfaces are elements allowing ***ATEasy*** to communicate with external devices such as instruments, computers, files, and more.

ATEasy supports two types of interfaces: internal (built-in) and external. **Internal** interfaces are built into your machine and do not require any special software or configuration. These interface types include:

COM – for serial communication.

FILE – for file I/O.

WinSock – for TCP/IP communication (also low level LXI instruments).

ISA – for PC based ISA bus based instruments.

USB – Universal Serial Bus, used to access an external USB instrument connected to your PC with a USB cable.

NONE – for drivers that do not use ***ATEasy*** interfaces.

In addition, ***ATEasy*** supports **external** interfaces requiring configuration and vendor specific DLL libraries. The following external interface types are available:

GPIB – General Purpose Interface Bus or IEEE-488, used to access an external GPIB instrument connected to a GPIB interface board installed in your machine with a GPIB bus cable. ***ATEasy*** supports many GPIB interface board vendors including Computer Boards, Keithley (CEC), HP, and National Instruments.

VXI – VME eXtension Interface. Allows ***ATEasy*** to communicate with VXI based instruments using a National Instruments MXI-VXI board installed in your machine.

Creating and Adding Drivers

In this section, you will create a new driver. Typically, if you already have the driver for your instrument, you will only need to add and configure it for your system before using it like we did in *Chapter 7*. In this example, you will be developing a new driver for the GT98901 Educational Demo Board instrument. We will be creating a new project called **MyGT98901Driver** to differentiate driver development from test application development which we did with **MyGtDemoProject**.

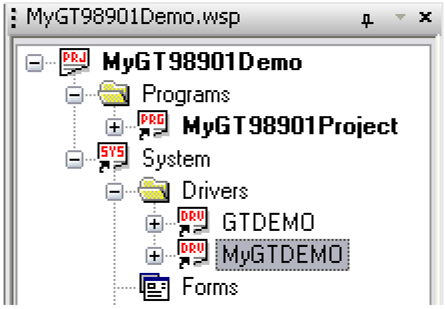
▼ To add an existing driver to the System:

1. Select the **Drivers** submodule below the **System** module in the Workspace Window.
2. Select **Driver Below** from the **Insert** menu or from the Standard toolbar. ***ATEasy*** displays a list of available drivers. By default, ***ATEasy*** drivers are installed in the Drivers folder below the main ***ATEasy*** folder.
3. Select **GTDEMO.drv** and click **Open**. ***ATEasy*** loads the driver and names a **driver shortcut** as **GTDEMO**. This driver is for the USB based Educational Demo Board (GTDEMO). The document view for the driver opens in the client area.

▼ To create a new driver in the System:

4. Right-click on the **Drivers** submodule below the **System** module in the Workspace Window and select **New Driver** from the context menu. ***ATEasy*** adds a new driver and names the driver shortcut **Driver1**. The document view for the new driver opens in the client area.
5. Click on the **Save All** command from the file menu and save the new driver (Driver1) to **MyGTDEMO** in the MyProject folder. Notice that ***ATEasy*** renamed the shortcut from Driver1 to MyGTDEMO.

   At this point, you should have two drivers below the system **Drivers** submodule as displayed here:



Driver and Driver Shortcut

It is important to understand the difference between a driver shortcut, shown in the workspace window, and the driver itself, shown in the document view. Visually, as you can see, the driver shortcut has a little arrow , and the driver image does not have one.

The driver is based on the instrument or the device describing the device default name, supported interfaces, and more. The driver shortcut is based on the configuration of the device. It contains the name used to identify the driver in the application (and usually uses the driver default name if not taken), the selected interface being used in the system, and the address of the device.

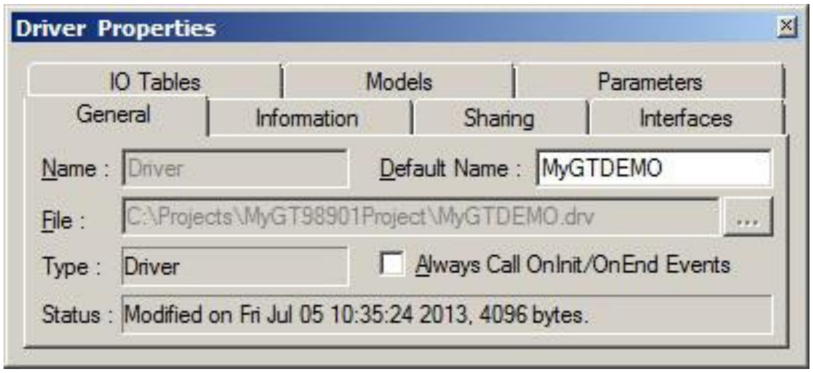
Changes made to the driver will be saved in the driver file, while changes made to the driver shortcut are saved in the System module.

Driver Default Name

The driver default name usually indicates the type of instrument this driver accesses. For a digital Multimeter, use DMM. This name is used when you add the driver to a system as the default identifier name. It is used to identify the driver in your system programmatically.

▼ To define the driver's default name:

1. Select **Driver** in the MyGTDEMO document view and select **Properties** from the **View** menu or from the Standard toolbar. The properties window appears displaying the Driver Properties:



Name : Driver

Default Name : MyGTDEMO

File : C:\Projects\MyGT98901Project\MyGTDEMO.drv

Type : Driver

□ Always Call OnInit/OnEnd Events

Status : Modified on Fri Jul 05 10:35:24 2013, 4096 bytes.

2. Type **GTDEMO** in the Default Name edit box.

Defining the Driver Interface

The driver interface to be created is a USB driver. You will be adding this interface to the **MyGTDEMO** driver you created.

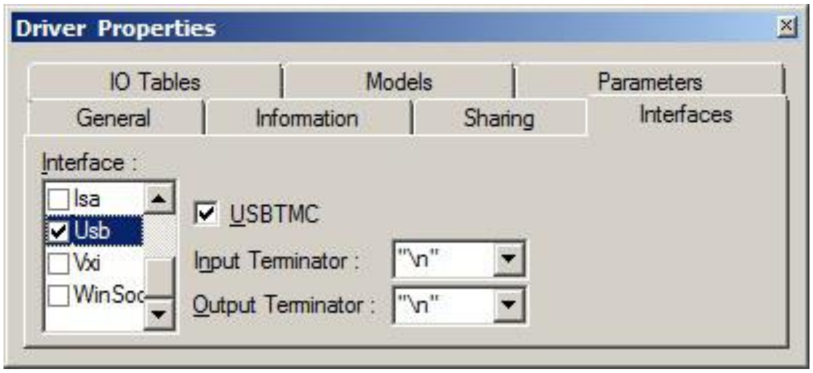
▼ To define the driver's interface:

1. Select the driver in the document view and either click the Properties command from the Standard toolbar, or select **Properties** from the **View** menu. When the properties window appears, click the **Interfaces** tab.

2. Check the **Usb** Interface in the list box to add USB support to this driver. As shown below, select the **LF** (Line Feed or **"\n"**) for the **Input** and **Output Terminator**. Also, select **USBTMC** to indicate that the target adheres to the USB Test and Measurement Class standards.

3. Uncheck the **None** interface, to make the USB interface the only interface supported by this driver.

   The driver Interfaces property page should like similar to the following dialog:



Interface :
☐ Isa
☑ Usb
☐ Vxi
☐ WinSoc

☑ USBTMC

Input Terminator : "\n"

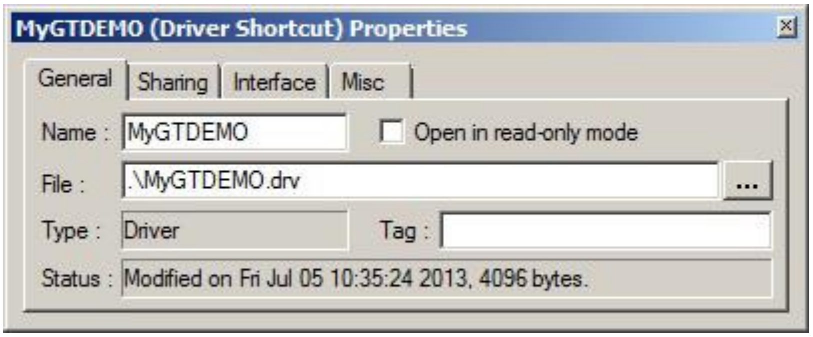
Output Terminator : "\n"

Configuring the Driver in the System

A driver with a USB interface has been created. Now, configure the driver to be used in your system.

▼ To configure the driver in the system:

1. Select the **MyGTDEMO** driver shortcut in the Tree View of the Workspace window. Open the properties dialog box by either clicking the Properties icon from the Standard toolbar, or selecting **Properties** from the **View** menu. The driver shortcut object properties dialog displays as below:



MyGTDEMO (Driver Shortcut) Properties

Name : MyGTDEMO

☐ Open in read-only mode

File : .\MyGTDEMO.drv

Type : Driver

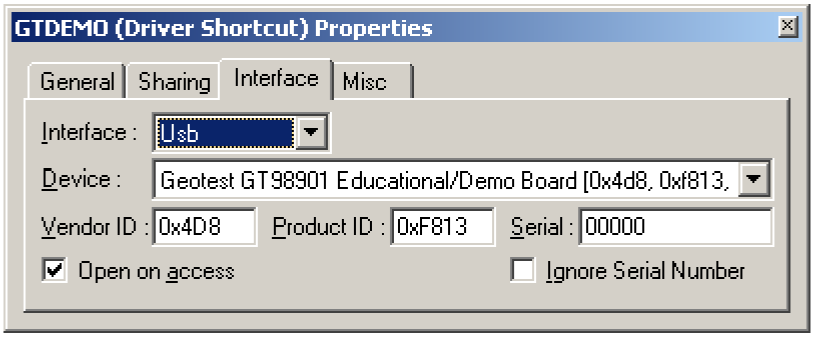
Tag :

Status : Modified on Fri Jul 05 10:35:24 2013, 4096 bytes.

2. Select the **Interface** page.

3. Select **Usb** from the combo box list for Interface.

4. In the **Device** dropdown, find and select the **Geotest GT98901 Educational/Demo Board** or **USBTMC** or **USB Test and Measurement Class.** If this is not found in the list, the device is not installed properly. A properly configured GT98901 should appear as follows:



GTDEMO (Driver Shortcut) Properties

Interface : Usb

Device : Geotest GT98901 Educational/Demo Board [0x4d8, 0xf813,

Vendor ID : 0x4D8

Product ID : 0xF813

Serial : 00000

☑ Open on access

☐ Ignore Serial Number

I/O Tables

To communicate with instruments using an Interface such as USB, you need to send data while handling all the low-level requirements of the protocol. This is an intricate task as some of these protocols (for example, GPIB and VXI) are complicated and require many actions for every string of data sent over the bus.

ATEasy has a unique mechanism called **I/O Tables** to handle this task. An I/O Table is a procedure containing **operations** instead of code to provide the implementation. Similar to a procedure, an I/O table uses parameters to transfer data between the device and the application.

I/O Table Operations include:

Output – appends data to the output buffer. The buffer is used to accumulate data from one or more output operations, which are later sent to the device using the Send operation. Data can be specified or passed as a parameter to the I/O Table.

Send – sends the content of the output buffer to the device.

Receive – receives data from the device via the interface and places it in the input buffer.

Input – reads data from the input buffer and stores it via arguments passed to the I/O Table.

Delay – adds a delay between operations.

Trig – triggers a device (applicable to GPIB and VXI only).

An I/O Table is one of the methods used to communicate with an instrument. ***ATEasy*** also provides procedures with lower level and protocol-specific ways to control instruments. These procedures reside in the ***ATEasy*** internal library.

Using an I/O table provides the driver a way to become interface independent and let the driver support more that one interface (for example, USB and RS-232) without the need to write interface-specific code inside the driver.

Creating an I/O Table: DisplayClear

I/O Tables perform a variety of functions. The first you will create will send data to the GT98901 to update the LCD display. The I/O table you will create here will be used to clear all text from the LCD display. The SCPI command used in this example and all the commands in this chapter are defined in *Appendix B – SCPI Function Reference*. To execute this command, you need to send a string to the demo board instructing it to clear the display. For that, you need one output operation and one send operation.

▼ To create an IO Table:

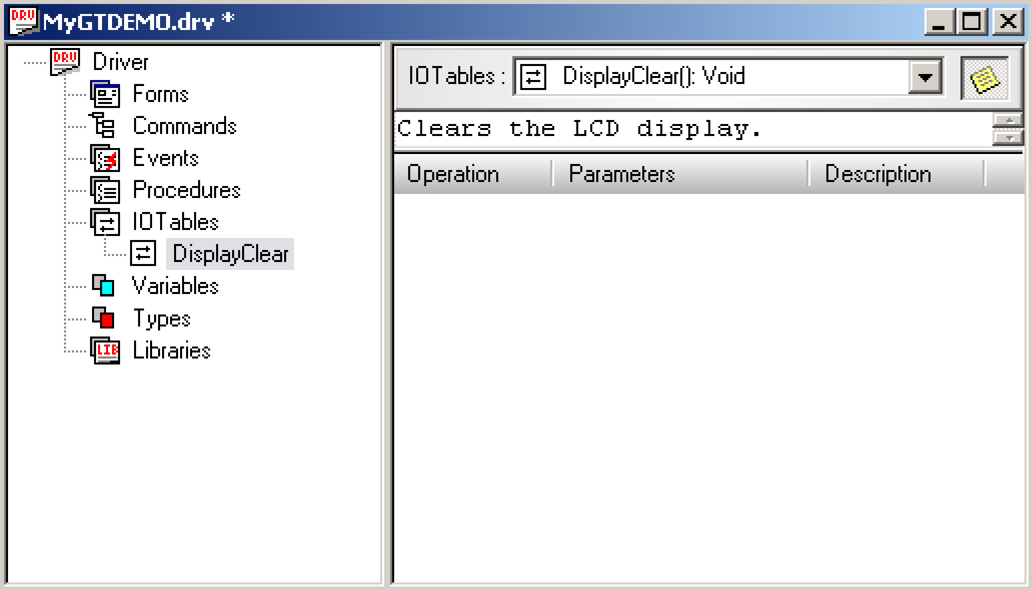
1. Select the **IOTables** submodule from the tree view in the Document View of the MyGTDEMO driver.

2. Select **IoTable Below** from the **Insert** menu or from the Standard toolbar. A new I/O table called **IOTable1** is created.

3. Rename the I/O Table by typing **DisplayClear**.

4. Type the following description for this operation in the description view: **Clears the LCD display.**

   Your screen should now look similar to the following:



Clears the LCD display.

The I/O Table object view displayed at the right side contains a combo box showing a list of the module I/O Tables (the current I/O Table is shown when the list is collapsed). The area below it is used to for the current I/O Table description, and the lower area contains a list showing the current I/O Table operations.

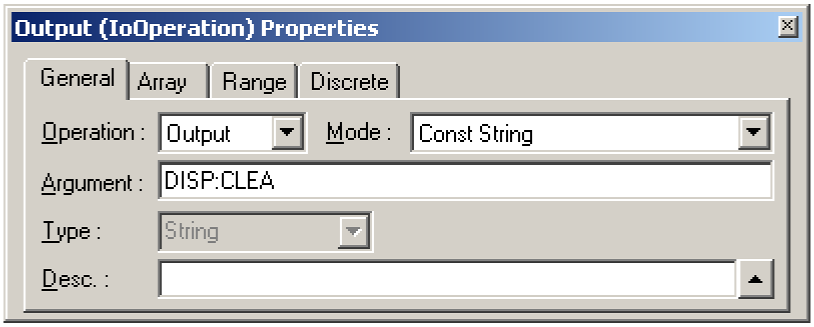
▼ To create the Output operation:

5. With the I/O Table **DisplayClear** selected, right-click on the operations view and select **Insert IoOperation After** from the context menu. An Output operation is created.

6. Right-click on the Output operation and select **Properties**. You need to enter a string into the Argument field to designate the VDC mode. Enter the following string required by the Demo Board to clear the display:

```
DISP:CLEA
```

Appendix B of this user guide contains a listing of the SCPI commands that are used to control this instrument. The SCPI command listing specifies that this is the string to be sent in order to clear the LCD display. No other changes are required as the default mode of the Output operation is Const String (Constant String). Your properties window should look similar to the following:



Output (IoOperation) Properties

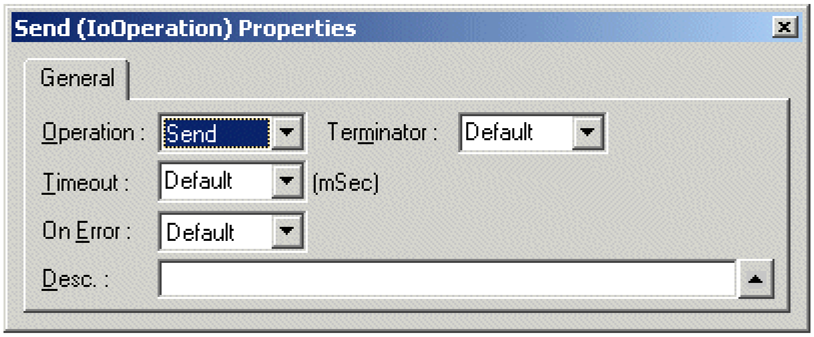
Operation : Output
Mode : Const String
Argument : DISP:CLEA
Type : String
Desc. :

The Output operation you just inserted appends the string to the output buffer. However, you need to transmit the buffer content to the demo board. To send the data over the USB bus, you need to add a Send Operation.

▼ To create the Send operation:

7. Right-click on the Output operation in the operations view and select **Insert IoOperation After** from the context menu. A new Output operation is created.

8. Right-click on the new Output operation and select **Properties**. In the properties dialog box, select **Send** from the **Operation** combo box. No other changes are required. Your screen should now look similar to the following:



Send (IoOperation) Properties

Operation : Send
Terminator : Default
Timeout : Default (mSec)
On Error : Default
Desc. :

The operations view now shows two operations: Output, followed by the Send operation. Your I/O table is now completed and is ready to use.

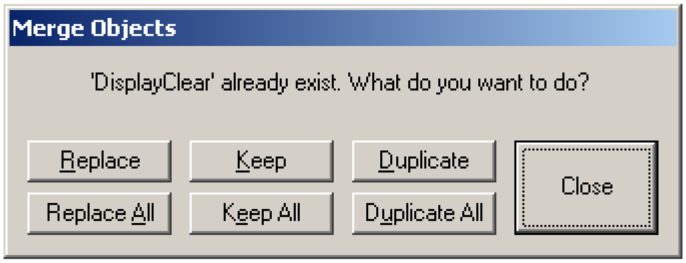
Using the Output Discrete Mode

In this section, you will create a similar I/O Table called **DisplaySetLineNumber**. The DisplaySetLineNumber changes the DMM measurement mode to measure in Volts AC. Instead of repeating the steps to create the previous I/O Table, you will use the clipboard commands to duplicate the DisplayClear I/O Table. Then, you will modify the I/O Table and the Output operation to the VAC functionality.

▼ To create the DisplaySetLineNumber I/O Table:

1. Right-click on DisplaySetLineNumber and select the **Copy** command from the context menu.

2. Right-click again on the DisplaySetLineNumber and select the **Paste** command from the context menu. A dialog box displays as shown here:



Merge Objects

'DisplayClear' already exist. What do you want to do?

3. Click on the **Duplicate** button. A new I/O Table is created and named **DisplayClear1**.

4. Select the **DisplayClear1** I/O Table and rename it to **DisplaySetLineNumber**. (Use the **F2** key if you need to.)

5. Type the following description for this table in the description view: Sets the currently selected display line. The selected display line is the line which is updated when DISP:TEXT is called. (Hint: you can just edit VDC to VAC.)

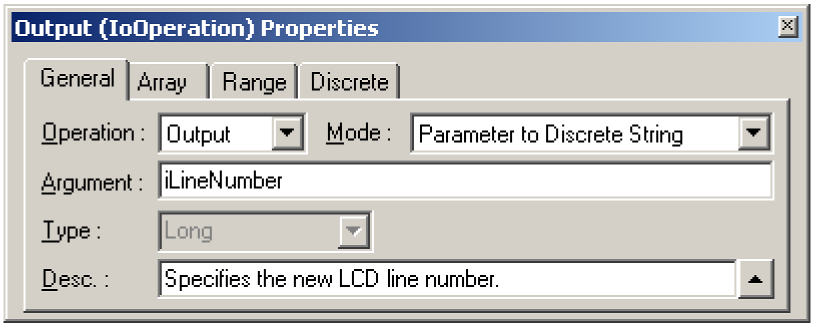
6. The syntax for the SCPI command to set a new line number is DISP:LINE <X> where X is a line number between 1 and 8. Select the Output operation, open the properties window and change the argument to display "DISP:LINE " without the quotes. Be sure that you include the whitespace after the text.

7. Right-click the Output operation that you just edited and select**Insert IOOperation After**. Open the properties window for this newly created operation.

8. Select **Parameter to Discrete String** from the Mode combo box.

9. Set the Argument name to **iLineNumber**. Enter the description **Specifies the new LCD line number.**

   Your screen should now look similar to the following:



Output (IoOperation) Properties

Operation : Output

Mode : Parameter to Discrete String

Argument : iLineNumber

Type : Long

Desc. : Specifies the new LCD line number.

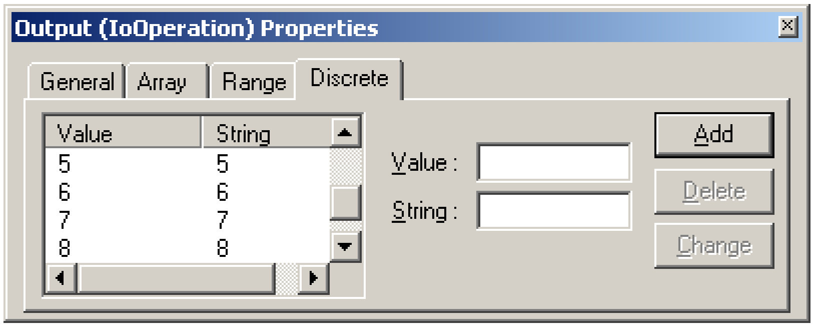
10. Select the **Discrete** page of the Output properties. This page contains a cross-reference table for multiple I/O Table discrete values. You can enter a different string for each value and when the I/O table is called with a specific value, the corresponding string will be sent to the instrument.

11. Enter **1** in the Value field. Type **1** in the String field. Click **Add**.

12. Repeat steps 7 and 8 for each of the following:

Value	String
2	2
3	3
4	4
5	5
6	6
7	7
8	8

The discrete output properties page should now look similar to the following:

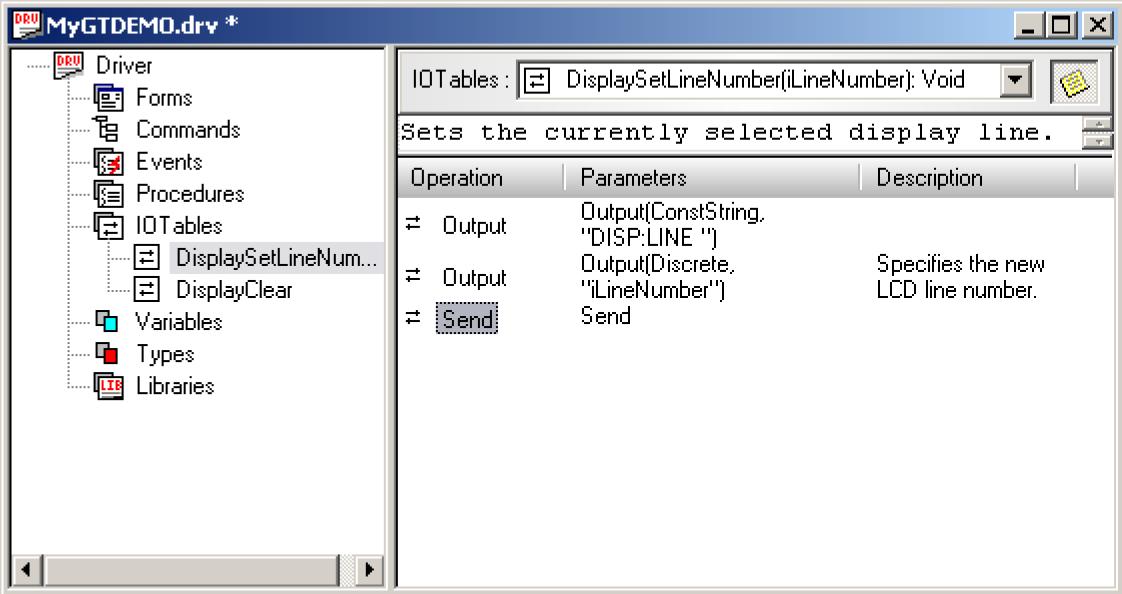


Output (IoOperation) Properties

Value	String
5	5
6	6
7	7
8	8

13. Add another Send operation.

Your document view should now look similar to the following:



IOTables : DisplaySetLineNumber(iLineNumber): Void

```
Sets the currently selected display line.
```

Operation	Parameters	Description
⇄ Output	Output(ConstString, "DISP:LINE ")	
⇄ Output	Output(Discrete, "iLineNumber")	Specifies the new LCD line number.
⇄ Send	Send	

You have created a single I/O table to handle all eight display line commands. When this I/O Table is called, the caller can specify the line number as a parameter.

Creating an I/O Table: DisplaySetText

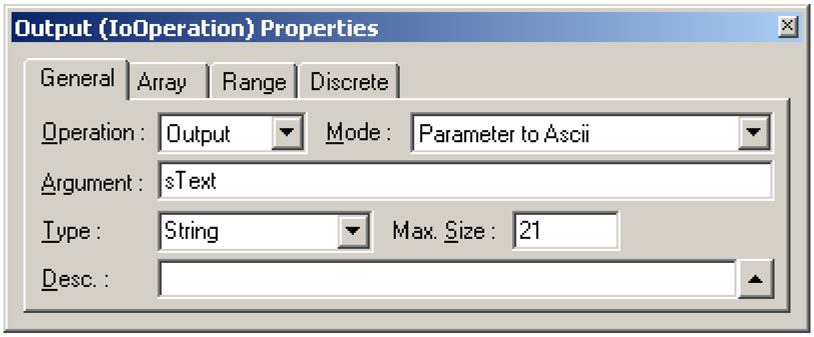
This next I/O Table will allow the caller to specify new text to the demo board’s LCD display.

▼ To create the DisplaySetText I/O Table:

1. Create a new I/O Table and name it **DisplaySetText**. Give it the description: **Sets the new text to the LCD's current line.**

2. Create an **Output** Operation and enter “**DISP:TEXT** “ as the argument without the quotes. Be sure to leave a whitespace after the text.

3. The caller of this I/O Table will specify the new text. Create another **Output** Operation. Change the mode from ConstString to **Parameter to Ascii** so the caller can pass a parameter with the new text. Change the type to **String** and the Max. Size to **21**. This is ensure that the caller will pass a valid parameter.



Operation : Output

Mode : Parameter to Ascii

Argument : sText

Type : String

Max. Size : 21

Desc. :

4. Finally, create a Send operation to transmit the data to the demo board.

Reading Data from the Instrument

The I/O Tables you have created to this point all send data to an instrument. The next step is reading data from an instrument, which typically involves sending out an instruction to the instrument to first provide the data and then read the data.

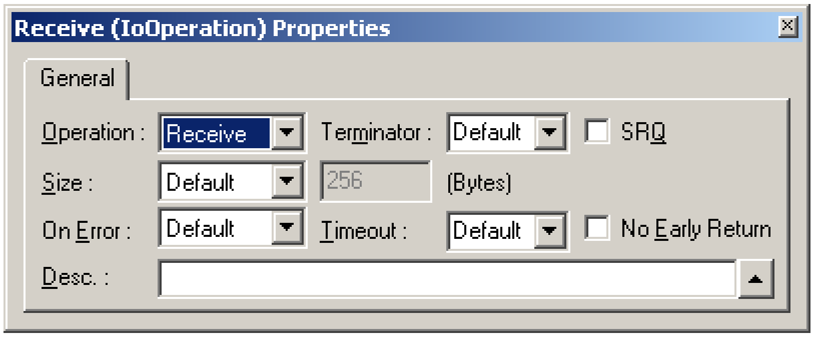
▼ To read data back from the device:

1. Create a new I/O Table and name it **DisplayGetLineNumber**. Give it the following description: **Gets the currently selected display line.**

2. Create an **Output** Operation and enter **DISP:LINE?** as the argument.

3. Create a **Send** Operation. These two operations direct the GT98901 to send data back over the bus. You need to read this data into ***ATEasy***.

4. Create a third operation. Change its type to **Receive** from the operation properties window as shown here:

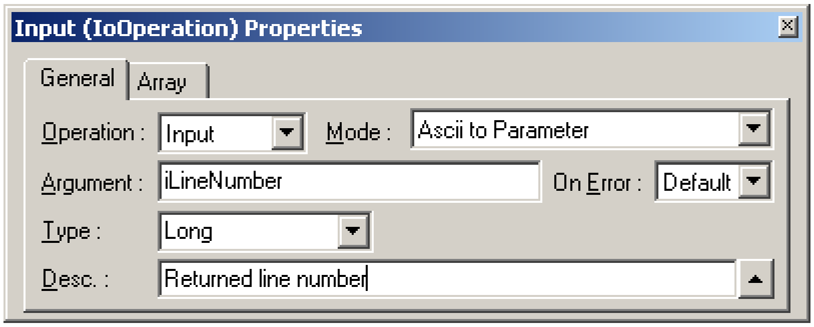


Receive (IoOperation) Properties

General

Operation : Receive
Terminator : Default
☐ SRQ
Size : Default
256 (Bytes)
On Error : Default
Timeout : Default
☐ No Early Return
Desc. :

5. Create another operation and change the operation type to **Input**. Leave the Mode as **ASCII to Parameter**, which causes ***ATEasy*** to convert ASCII data in the buffer to the parameter type you select. Enter **iLineNumber** in the Argument. The parameter type should be set to **Long**. Enter the description of the parameter as: **Line number**.



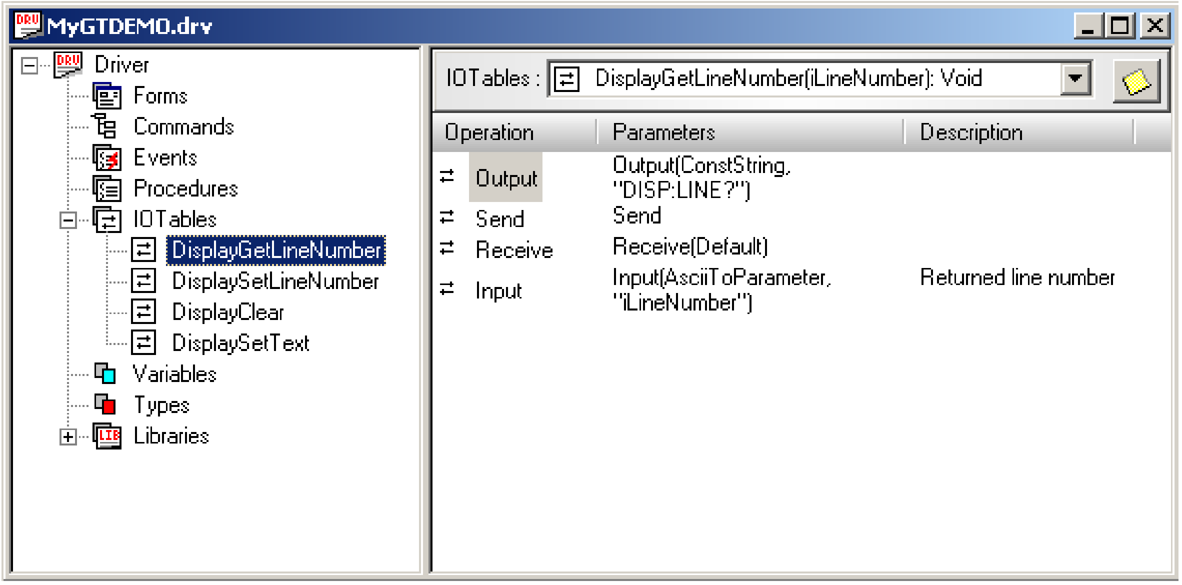
Input (IoOperation) Properties

General | Array

Operation : Input
Mode : Ascii to Parameter
Argument : iLineNumber
On Error : Default
Type : Long
Desc. : Returned line number

6. This I/O Table is now complete. When called, ***ATEasy*** first sends a string (DISP:LINE?) to the demo board and then reads back data and converts the data from ASCII to the parameter iLineNumber of type Long.

As a quick check, the driver's tree view in the document view should now have the four I/O Tables and look as illustrated below:



IOTables : DisplayGetLineNumber(iLineNumber): Void

Operation	Parameters	Description
⇄ Output	Output(ConstString, "DISP:LINE?")	
⇄ Send	Send	
⇄ Receive	Receive(Default)	
⇄ Input	Input(AsciiToParameter, "iLineNumber")	Returned line number

In the next section, you will call the I/O table from a new test that you will create in MyProgram. By default, ***ATEasy*** does not export I/O Tables to other modules. Normally I/O tables are used only within the driver by Commands. You can override this behavior by making the I/O Table **public**, so you can use it from other modules.

▼ To make I/O Tables public:

7. Right-click on the DisplayClear, and select **Properties** from the context menu.
8. Check the **Public** checkbox. This will make the I/O table visible to other modules.
9. Repeat step 2 for the rest of the I/O tables.

Calling an I/O Table from a Test

An I/O Table is a type of ***ATEasy*** procedure. As such, they can be called directly from tests or procedures (if declared public). I/O Tables may also be called using Driver Commands as explained in the *Commands* chapter. There, you will be calling the I/O Tables you created in this chapter via driver commands. The examples below are provided for your information.

When using a procedure or any symbol that is defined in another module or test, you can either make an *explicit* call specifying the module the symbol belongs to, or make an *implicit* call in which the module is not specified. In such cases, ***ATEasy*** will search the system and all configured drivers for the specified symbol.

The following example is for an *implicit* I/O Table call for the first I/O Table you have created.

```
DisplayClear()
```

The next example is for an *explicit* call to the same I/O Table.

```
MyGTDEMO.DisplayClear()
```

The third example demonstrates an implicit call to an I/O Table with an argument (parameter). As you recall, this I/O Table uses the Parameter to Discrete String argument and “1” represents the first .

```
DisplaySetLineNumber(1)
```

The fourth example demonstrates an explicit call to the **DisplayGetLineNumber** I/O Table with an argument **TestResult**.

```
MyGTDEMO.DisplayGetLineNumber(TestResult)
```

Using the Monitor View

Use the Monitor window to view the actual communication between ***ATEasy*** and the devices it controls. ***ATEasy*** displays data sent and received using USB, RS-232, VXI, WinSock, File IO, and more.

▼ To use the Monitor window:

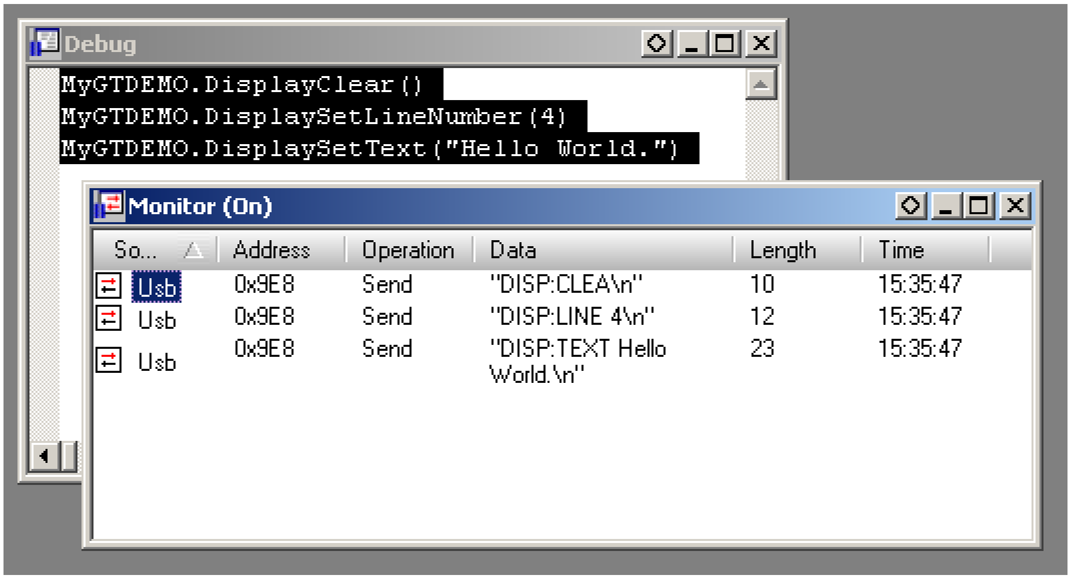
1. Select **Monitor** from the **View** menu. A dockable window appears. By default, the Monitor is turned off.

2. Right-click in the Monitor window, and select **Start Logging**. This starts the monitor.

3. To see how the sdmonitor displays information, open the Debug Window from the View menu and enter the following code:

```
MyGTDEMO.DisplayClear()
MyGTDEMO.DisplaySetLineNumber(4)
MyGTDEMO.DisplaySetText(“hello world.”)
```

4. Select the lines and select **Doit!** from the **Debug** menu. The monitor should display the following:



```
MyGTDEMO.DisplayClear()
MyGTDEMO.DisplaySetLineNumber(4)
MyGTDEMO.DisplaySetText("Hello World.")
```

So...	Address	Operation	Data	Length	Time
Usb	0x9E8	Send	"DISP:CLEA\n"	10	15:35:47
Usb	0x9E8	Send	"DISP:LINE 4\n"	12	15:35:47
Usb	0x9E8	Send	"DISP:TEXT Hello World.\n"	23	15:35:47

In addition, if the operation was successful, your GT98901 will display your message.

Checking for GT98901 Errors

The GT98901 automatically checks for errors after each SCPI command is received and executed. If the tutorials in this chapter were followed precisely, then the each IOProcedure called would be executed without error and the demo board would have responded as expected. But if a SCPI command has a syntax error or an invalid parameter, the demo board would respond by disregarding the command and adding a note to the error buffer. This buffer will continue to accumulate errors until the user reads the errors back or until the buffer is full. In order to prevent an error buffer overflow and to identify issues as quickly as possible, it is recommended that the error status be checked each time a command is sent to the GT98901.

▼ Create the GetError IOTable Procedure:

1. Create a new I/O Table and name it **GetError**.
2. Create an **Output** Operation and enter **SYST:ERR?** as the argument.
3. Create a **Send** Operation.
4. Create a **Receive** Operation. The response from the demo board consists of two parts, an error number and an error description.
5. Create an **Input** Operation. Enter **lError** as the argument and make sure the Type is **Long**. This will save the error number response as a parameter.
6. Create another **Input** Operation. Enter **sDescr** as the argument and make sure the Type is **String**. This will save the error description response as a parameter.
7. Set this this IOTable Procedure to public from the GetError IOTable properties editor.

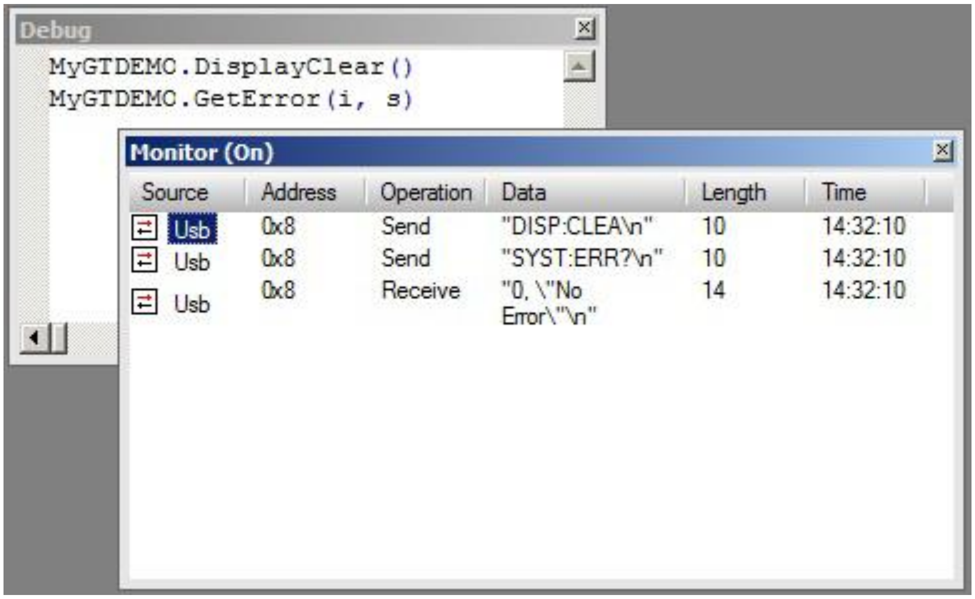
You now have a useful procedure that you can use to query the GT98901's error buffer for any error messages.

▼ Test the GetError IOTable Procedure using the Monitor Window:

8. A variable is needed to store the error description information. Create a variable **s** with type **String** in the program module.
9. Open both the Monitor Window and the Debug Window. Ensure that the Monitor Window is currently logging.
10. When no error has occurred, we expect the error number to be '**0**' and the error description to be '**No Error**'. Verify this status by entering the following code into the Debug Window:

    ```
    MyGTDEMO.DisplayClear()
    MyGTDEMO.GetError(i, s)
    ```

11. Run **DoIt!** and check the Monitor Window and verify that the commands were executed and there were no errors found. Note: If errors were generated in previous exercises and still reside in the error buffer on the GT98901, you will need to run GetError multiple times to empty the buffer. If run correctly, the monitor window will look like this:



```
MyGTDEMO.DisplayClear()
MyGTDEMO.GetError(i, s)
```

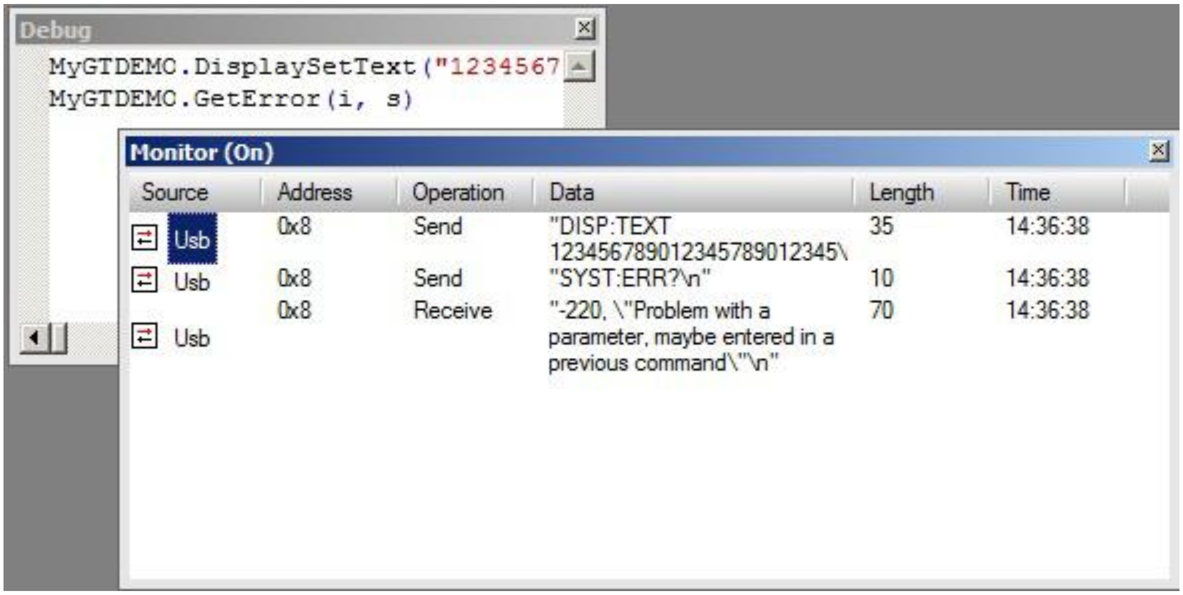
Source	Address	Operation	Data	Length	Time
Usb	0x8	Send	"DISP:CLEA\n"	10	14:32:10
Usb	0x8	Send	"SYST:ERR?\n"	10	14:32:10
Usb	0x8	Receive	"0, \"No Error\"\n"	14	14:32:10

12. The **DisplaySetLineNumber** procedure sets the line number on the LCD display to print ASCII when DisplaySetText is called. There is room for 21 characters on the display so valid parameter strings must be no more than 21 characters long. To generate an error, we will attempt to set the text to a 25-character long string. Replace the code in the Debug Window with the following:

    MyGTDEMO.DisplaySetText(“123456789012345789012345”)

    MyGTDEMO.GetError(i, s)

13. Run **DoIt!** and check the Monitor Window. You should observe a parameter error has been generated:



```
MyGTDEMO.DisplaySetText("1234567
MyGTDEMO.GetError(i, s)
```

Source	Address	Operation	Data	Length	Time
Usb	0x8	Send	"DISP:TEXT 123456789012345789012345\	35	14:36:38
Usb	0x8	Send	"SYST:ERR?\n"	10	14:36:38
Usb	0x8	Receive	"-220, \"Problem with a parameter, maybe entered in a previous command\"\n"	70	14:36:38

Writing an Error Checking Subroutine

The convention for driver error response is for an error number of 0 to signify no error, an error number of greater than 0 signifies a warning from the device, and an error number less than 0 signifies an error from the device. Placing the return-value-checking code after each call in the test program makes the program very long and hard to read. Instead, we can creae a single error-checking procedure, the **CheckError** driver procedure, to perform that test. Call **CheckError** after each SCPI command sent to the GT98901.

▼ To create the CheckError procedure:

1. Insert a new procedure within your MyGTDEMO driver module. Rename the new procedure to **CheckError**.

2. Create the following local procedure variables:

   Name: **lStatus**
   Parameter: **None**
   Type**: Long**
   Dim**: 0**

   Name: **sError**
   Parameter: N**one**
   Type: **String**
   Dim: **0**

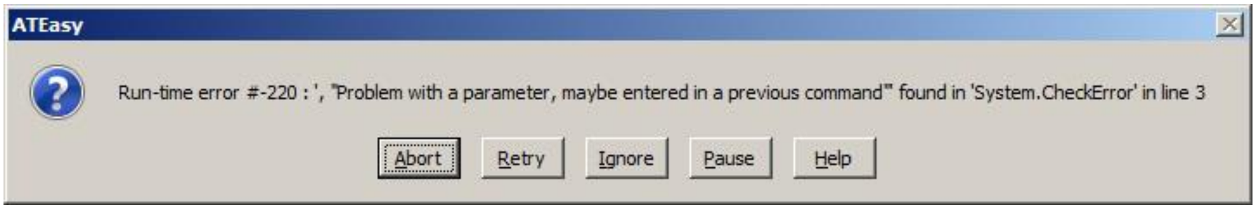
3. The CheckError procedure will call the GetError IOTable procedure to retrieve any error information from the GT98901. If an error has occurred (error code < 0), then the error is reported to the user using the ATEasy default error dialog window:

```
GetError(lStatus, sError)
If lStatus<0
   Error lStatus, sError
EndIf
```

4. The CheckError procedure will call the GetError IOTable procedure to retrieve any error information from the GT98901. If an error has occurred (error code < 0), then the error is reported to the user using the ATEasy default error dialog window:

```
MyGTDEMO.DisplaySetText(“1234567890123457890l2345”)
CheckError()
```

5. Run **DoIt!** and observe the error message that will now appear when an error is generated:



ATEasy

Run-time error #-220 : ', "Problem with a parameter, maybe entered in a previous command"' found in 'System.CheckError' in line 3

More about Error Handling

ATEasy run-time errors can be generated using the **error statement** or as a result of a run-time error such as divide by zero, communication failure and more. By default, ***ATEasy*** will display a message box displaying the error number, text and location of where the error occurred in your code. The message will contain the following buttons:

Abort – Pressing abort will call the **OnAbort()** event sequence and could abort the program.

Ignore – continue execution with the statement following the statement that caused the error.

Retry – will display only retry able errors such as communication error. Retry will cause the statement causing the error to be called again.

Pause – this button is available only when running from the development environment. Pressing pause will cause the execution to be paused and will cause ATEasy to display the statement causing the error. The user, then can watch variables, changes the current statement and debug.

ATEasy applications can trap and handle errors before the default message box is displayed. You can place code in the **OnError**() module event to handle error and can handle errors programmatically using the **abort**, **retry**, **ignore** and **pause statements**. In addition the **try-catch statement** can handle errors locally and provide local error and exception handling. In addition the **GetErrorModule()**, **GetErrorNum()** and **GetErrorMsg()** internal functions can be called to retrieve error information.

The driver procedure and the **CheckError** procedure causes the user to concentrate on the test code without the need to check for errors after each statement. It also provides the test program or the application with a single point (the **OnError()** module event) area in which to place the error handling code.

More I/O Tables

In this chapter, we explored the functions from one of the GT98901's subsystems: the LCD display. In future chapters, we will be exploring the use of the other subsystems through the use of ATEasy commands. Since we won't be creating IOTables in future chapters, take this opportunity to practice creating more IOTables with your MyGTDEMO driver.

Similar to DISP:CLEA

```
*CLS
*RST
DAC:TRIG
DAC:ABOR
```

Similar to DISP:LINE <x>

```
DIO:PWM
ADC:TRIG
LED
REL:OUTP
```

Similar to DISP:LINE?

```
PUSH:READ?
ADC:SCAN?
BUZZ?
DAC:BUFF:PAGE?
```

Similar to DISP:TEXT <s>

```
REF:DIV
DIO:OUTP
ADC:SCAN
DAC:CLOC:DIV
```

Chapter 10 - Commands

About Commands

This chapter discusses user-defined statements that extend the ***ATEasy*** programming language. These are called *Commands*. Use the table below to learn more about this chapter's topics.

Topic	**Description**
Overview of Commands	What are ***ATEasy*** commands, the syntax of commands, and the benefit of using Commands?
Commands and Modules	Discusses the modules that can have commands and provide examples of commands.
The Commands View	Describes the Commands view used to create commands.
Creating Driver Commands	Provides a detailed, step-by-step example of creating driver commands.
Attaching Procedures and I/O Tables to Commands	How to attach the procedures and I/O Tables you previously created to commands.
Replacing Parameters with Arguments	How to replace parameters with constants and variables arguments with parameters for commands that are attached to procedures with parameters.
Using Commands from Other Modules	How to use commands created in other modules. Provides rules and recommendations for using commands from other modules.
Creating System Commands	Provides an example explained in a step-by-step procedure to create a system procedure and command. Also, demonstrates how to use auto command completion and insert command cascading menu to insert command to your code.
Program Commands	Provides examples of program commands.

Overview of Commands

One of the notable ***ATEasy*** features is the ability to define and extend the programming language by adding user-defined statements that look like English statements. Command statements have the following syntax:

Syntax:	*Module Name*	*Command Items...*	*[(Arguments)]*
Examples:	`GTDEMO`	`Display Clear`	
	`GTDEMO`	`Digital Get Data`	`(dwResult)`

The module name comes first in the command. This is either the current module (Program, System or Driver) or the specific name of a driver (GTDEMO). Next in the command is a set of words that makes up the command item. When you create a command item, you may attach a procedure to it. At run-time, the procedure or I/O table is called when the command statement is executed. The last portion of the command is called an argument. The argument is taken from the list of procedure that may be attached to the command.

ATEasy lets you substitute a supplied parameter when writing the commands or, alternatively, you can supply them when you use the command statement in your code.

Commands replace procedures. There are many reasons to use commands instead of procedures:

- Commands are self-documented. They look like plain English and they reduce the need for documentation. They replace cryptic procedure names with English-like statements.
- Commands make your test program looks like a TRD (Test Requirement Document).
- The command items structure makes it easy to locate, browse, and categorize them. A typical instrument driver may contain hundreds of commands. By grouping command items into categories such as Setup, Measure, etc., you can locate them more quickly when you need to use them.
- Commands can be used to hide arguments passed to the procedure, thereby simplifying coding.
- Commands encourage you to create a standard programming interface for an instrument. This can later be used for similar instrument types (for example, DMM), making your test programs instrument-independent. For example, you can create a template containing commands for a DMM, which can be reused for each DMM you use.
- Once defined, commands appear in cascading menus under **Insert** on the ***ATEasy*** Menu bar. Choosing commands via menus eliminates typing and syntax errors. In addition, automatic command completion provides another way for the user to use commands.

Commands and Modules

ATEasy commands can be defined in each of the module types:

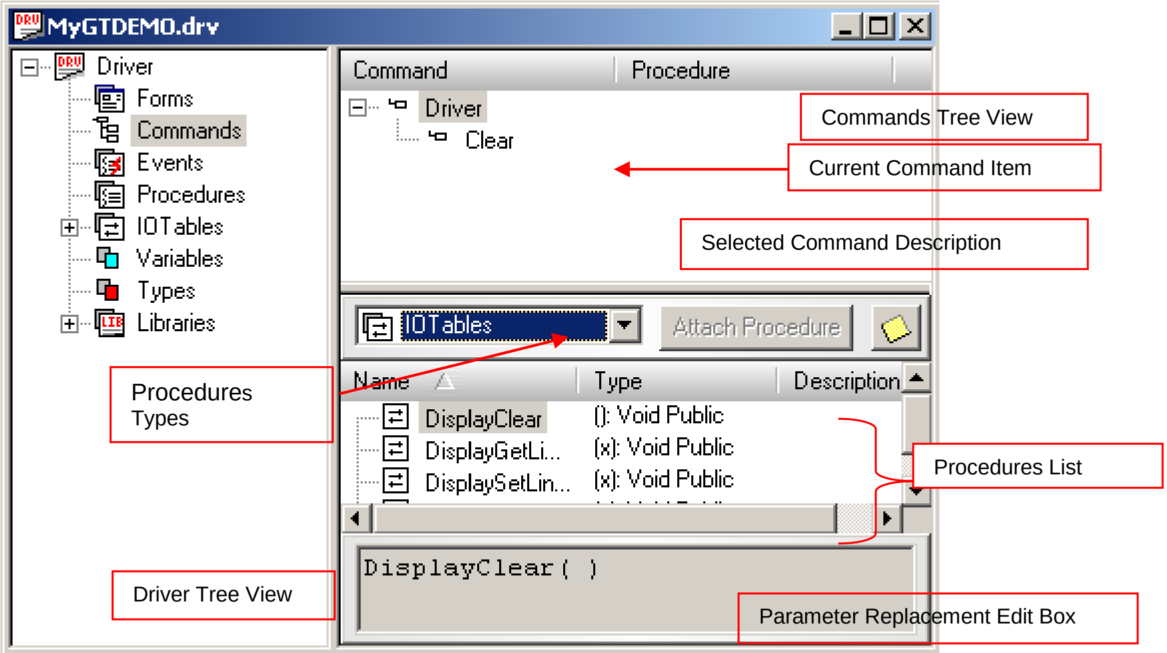
- **Driver commands** provide a layer between the programs and the instruments instructions (for example, I/O Tables, DLL procedures). Although you can call I/O Tables and DLL procedures directly from programs, it is more convenient and more organized to do it using driver commands. Driver commands can start with the driver shortcut name (for example, DMM) or with the **Driver** keyword. Using **Driver** in a command statement can be done only within the driver procedures. Used this way, it refers to the current driver. When using a command in a driver procedure that was defined within the driver, you should use the **Driver** name instead of the driver shortcut name. This is recommended because the driver shortcut name can be changed from system to system (or you may have two DMMs in your system: DMM1, DMM1).
- **System commands** provide a layer between several instrument drivers and the program. A single command can be linked to a procedure using several instruments to perform a single task. Typically, system commands are used when a specific function or task needs to be accessible by all the programs in a given project. System commands always start with the **System** module name.
- **Program commands** improve programming by creating language elements specific to a UUT for repeated actions unique to a specific test program. While System and Driver commands are accessible by all modules in a given project, the program statements can only be used within that program. Program commands always start with the **Program** module name.

Here are some examples of commands:

Command	**Description**
`DMM Set Function VDC`	Sets the DMM to Volts DC measurement mode
`DMM Set Range 300V`	Sets the DMM’s range to 300 Volt
`DMM Measure (dResult)`	Reads a measurement from the DMM’s buffer
`RELAY Close (1)`	Closes relay #1 on a relay card
`FUNC Set Frequency (15000)`	Sets the frequency of a function generator to 15KHz
`System Measure J1_23 VDC (dResult)`	System command: Switches to route signals to J1_23, sets DMM to VDC, and takes a measurement.
`Program Start Engine Left`	Program command: Closes a relay for a specific time to start an engine.

The Commands View

The Commands view is used to create and edit commands. An example of this view is shown below:



The left pane of the Commands View shows the module's tree (in this case, the Driver Tree View). The top right pane of the Commands View shows the Commands Tree View listing all the available command items. The highlighted command item in this view is the **Current or Selected Command Item (Function** here**)**. Below the commands tree is a textbox for the selected command item description. Below this textbox, there is a drop down list on the left listing the available procedures types (for example, I/O Tables, DLL procedures). To the right of the list box, the Attach/Remove procedure button allows you to attach or remove procedures to/from the current command item.

In the pane below the procedure types is the **Procedures List** of the available procedures for the selected procedure type. The default procedure type for all ***ATEasy*** modules (i.e. driver, system and program) is Procedures (local procedures).

The bottom pane displays the currently selected procedure. This is the **Parameter Replacement Edit Box** where you may substitute values for parameters in a procedure, so the user will not need to enter them when using the command.

The Commands View is almost identical for Drivers, System, or Programs. The only difference is the type of procedures available to each. All have local procedures, the ***ATEasy*** Internal Library procedures, as well as any other library linked to that module. The Driver's Commands View also adds I/O Tables to the list of available procedure types as only ***ATEasy*** drivers have I/O Tables.

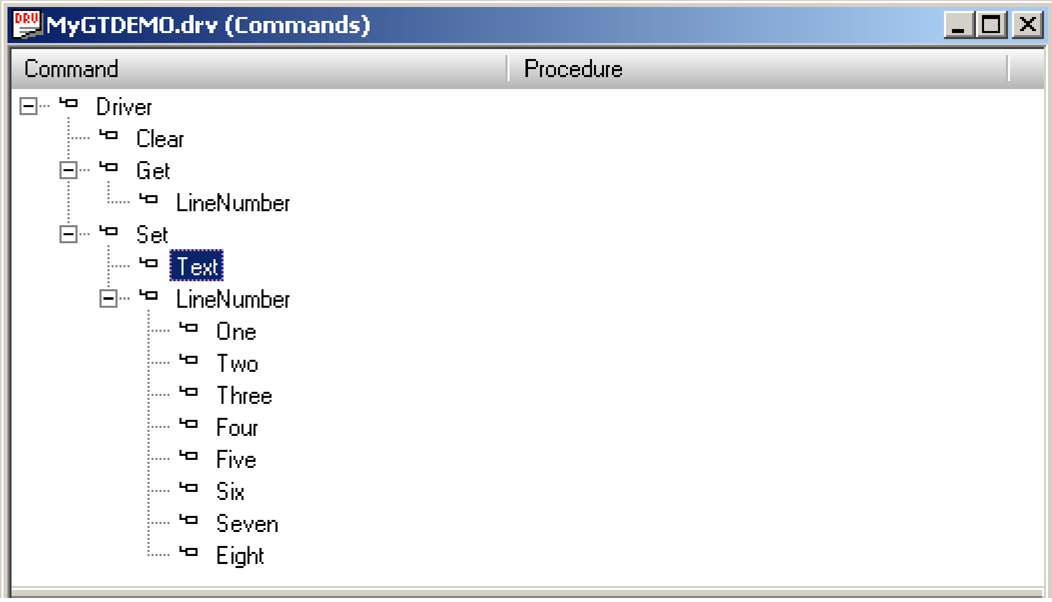
Creating Driver Commands

In Chapter 9, you created several I/O Tables for the GTDEMO driver. Now create Driver Commands for these I/O Tables.

▼ To create driver command items:

1. Double-click on the **Commands** submodule under **MyGTDEMO** in the Workspace window. The commands view is displayed.
2. Right-click on Driver and select **Insert Command Below** from the context menu. A new command item is inserted called **Untitled1**. Type **Display** in the edit box to rename the command item.
3. Insert a new item below **Display**. Rename it **Clear**.
4. Repeat step 3 and insert two items after **Clear**. Rename them **Set** and **Get.**
5. Insert two items below **Set**. Rename them to **Text** and **LineNumber.**
6. Insert one item below **Get**. Rename it to **LineNumber**.
7. Under the **Driver Set LineNumber** command, insert the following subcommands: **One**, **Two**, **Three**, **Four**, **Five**, **Six**, **Seven**, **Eight**.

At this point, the commands view should look as shown here:



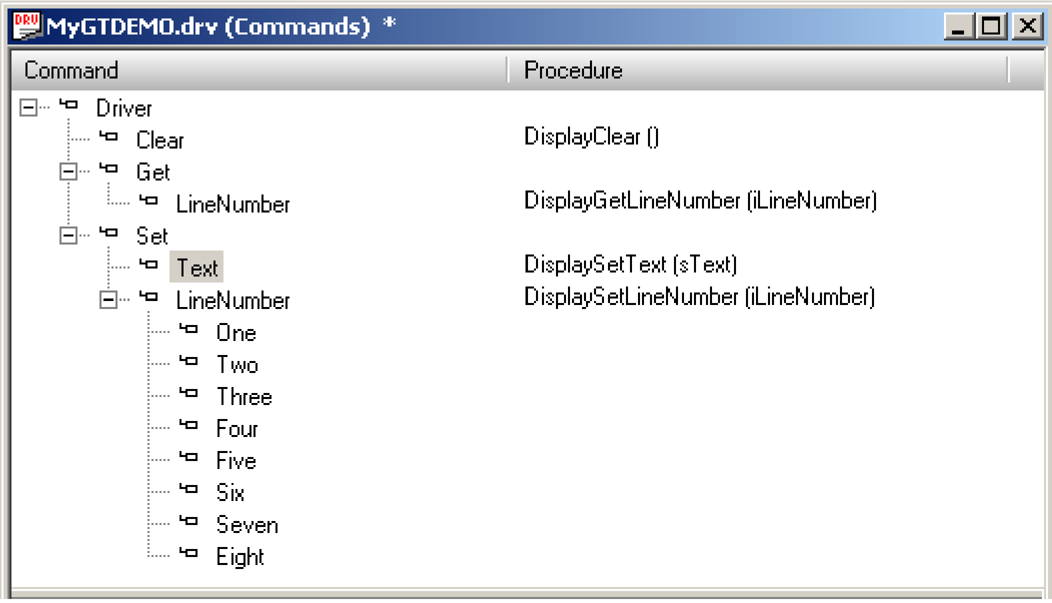
Attaching Procedures and I/O Tables to Commands

A command item becomes a command only after you attach a procedure or an I/O Table. Since the driver MyDMM uses I/O Tables, attach them to its command items.

▼ To attach I/O Tables to Command items:

8. Select the **Clear** command item in the command items view.
9. Select **IO Tables** from the Procedures combo-box. The available I/O tables are displayed in the list below.
10. Select **DisplayClear** from the procedures list.
11. Click **Attach Procedure**. The Procedure is now displayed next to the command item in the command items view.
12. Repeat steps three and four for the **LineNumber** and **Text** command items.

The commands view should look as shown here:



```
Command                      Procedure
Driver
    Clear                    DisplayClear ()
    Get
        LineNumber           DisplayGetLineNumber (iLineNumber)
    Set
        Text                 DisplaySetText (sText)
        LineNumber           DisplaySetLineNumber (iLineNumber)
            One
            Two
            Three
            Four
            Five
            Six
            Seven
            Eight
```

Four commands were created:

```
Clear
Get LineNumber (iLineNumber)
Set LineNumber (iLineNumber)
Set Text (sText)
```

Replacing Parameters with Arguments

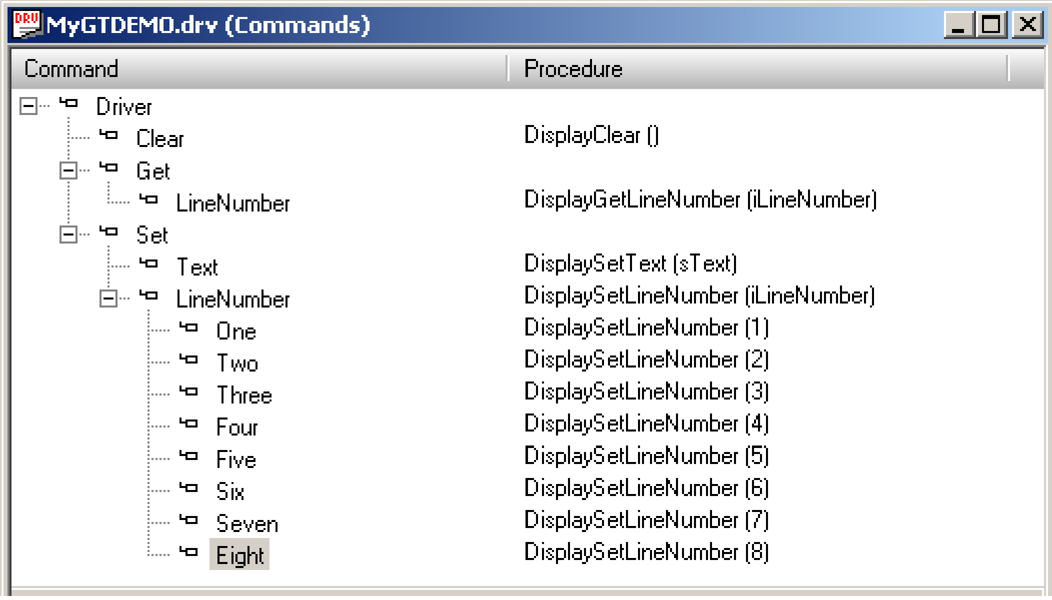
When attaching procedures with parameters to command items, you can replace the parameter with an argument. The argument you specify will be used instead of the parameter and the command will not require the user to supply an argument. Parameters are usually replaced with literals that you supply, but can also be replaced with variables that you define.

In the following example you will use the **DisplaySetLineNumber** I/O Table to implement the remaining commands under Set LineNumber: **One**, **Two**, **Three**, **Four**, **Five**, **Six**, **Seven**, and **Eight**.

▼ To implement the remaining command:

1. Attach **DisplaySetLineNumber** to the **One** command item.
2. Select the **iLineNumber** text in the Parameter Replacement edit box (at the bottom of the commands view) and type **1** to replace the text. The command items view is automatically updated to display DisplaySetLineNumber (1).
3. Repeat steps one and two for **Two** through **Eight** and use **2** through **8** respectively.

The commands view should look as follows:



Command	Procedure
Driver	
Clear	DisplayClear ()
Get	
LineNumber	DisplayGetLineNumber (iLineNumber)
Set	
Text	DisplaySetText (sText)
LineNumber	DisplaySetLineNumber (iLineNumber)
One	DisplaySetLineNumber (1)
Two	DisplaySetLineNumber (2)
Three	DisplaySetLineNumber (3)
Four	DisplaySetLineNumber (4)
Five	DisplaySetLineNumber (5)
Six	DisplaySetLineNumber (6)
Seven	DisplaySetLineNumber (7)
Eight	DisplaySetLineNumber (8)

Eight commands were created:

```
Set LineNumber One
Set LineNumber Two
Set LineNumber Three
Set LineNumber Four
Set LineNumber Five
Set LineNumber Six
Set LineNumber Seven
Set LineNumber Eight
```

Using Commands from Other Modules

By default, command items are created as **Public**. This makes command items available for use by other modules as well as for use within the same module. Turning off the public flag prevents the user from using them in other modules. The Public property can be set from the command item Properties window.

Commands are available between modules as follows:

- Program commands can access all of the commands defined within the program as well as the public commands of both the system and drivers.
- System commands can access all of the commands defined within the system itself as well as the public commands of the drivers.
- Driver commands can access commands defined within the driver as well as public commands defined by the system and other drivers. It is recommended to not use other driver or system commands from a driver, since it makes that driver dependent on the current system and on the driver shortcut names. This can make the driver work only on one system and can reduce the re-usability of the driver.

Creating System Commands

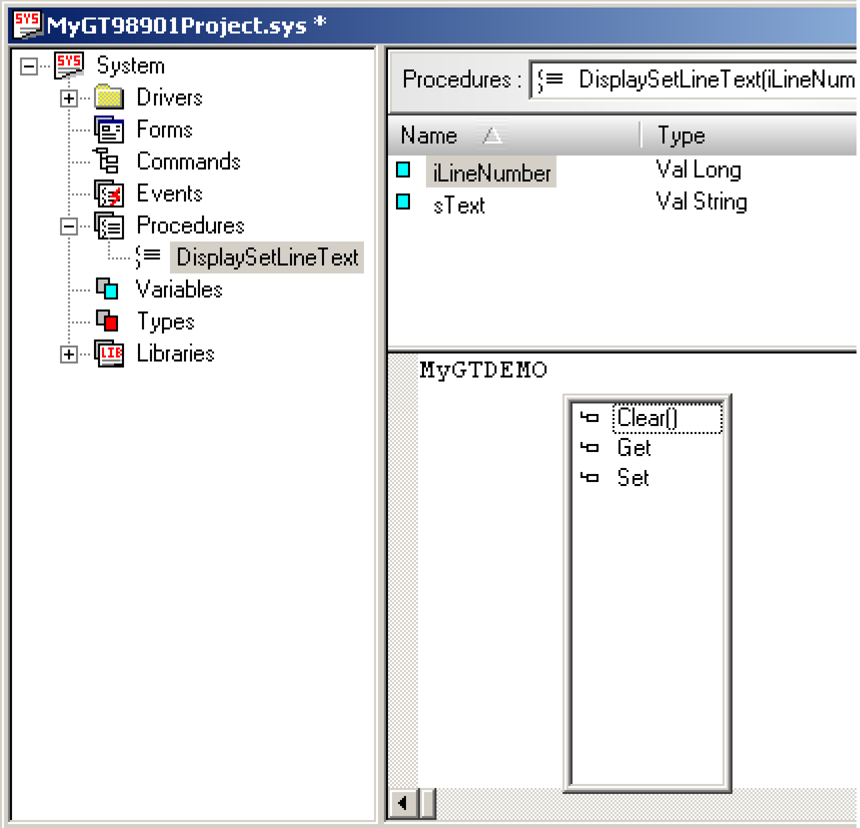
Before you can employ System commands, you need to create a System Procedure. In this example, you will create a procedure named DisplaySetLineText(sText). The procedure will call two commands defined in the MyGTDEMO driver.

Typically, your system will have procedures that route signals from the UUT to the measurement instrument. The signal will be routed using a switching instrument you may have in your system. Then the procedure will call functions to both set up the measurement and to take a measurement. Since your system contains only measurement instruments, only use the GTDEMO to implement the procedure.

▼ To create a System procedure:

1. Open the System document view by double-clicking on the system shortcut from the Workspace window.
2. Right-click on **Procedures** in the tree view of the system document view and select **Insert Procedure Below** . A new procedure is created.
3. Open the Properties window and rename the procedure to **DisplaySetLineText**.
4. Right-click on the procedure variables view and select **Insert Parameter/Variable At** . A new variable is inserted. Rename it to **iLineNumber**.
5. Insert another variable. Rename it to **sText**.
6. Right-click on **sText**. Select **Properties** and change the variable type to **String**.

7. Type **MyGTDEMO** followed by a space in the procedure code view. The command auto completion will display the commands available from the MyGTDEMO driver as follows:

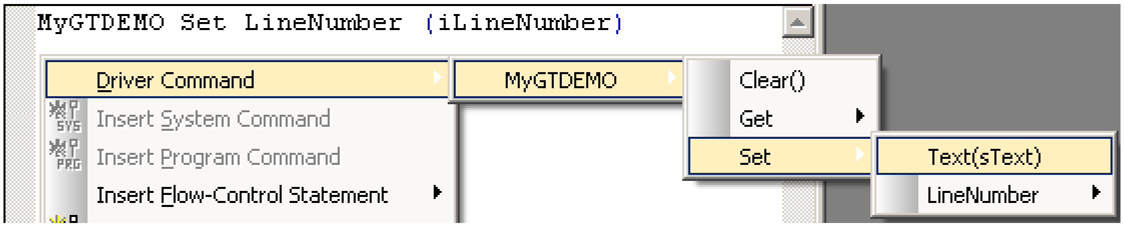


Name	Type
iLineNumber	Val Long
sText	Val String

```
MyGTDEMO
```

8. Press the down arrow key to select **Set.** Press ENTER and continue to select **LineNumber** and **(** **)**. The parameter suggestion will display the appropriate potential variables that can be used. Select **iLineNumber** and then type **)** to close the command call.

9. On the next line, use the cascading menu to insert the next command. Right-click on the beginning of the next line. Select **Driver Command**. Select **MyGTDEMO**, and then select **Set Text(sText)** as shown here:



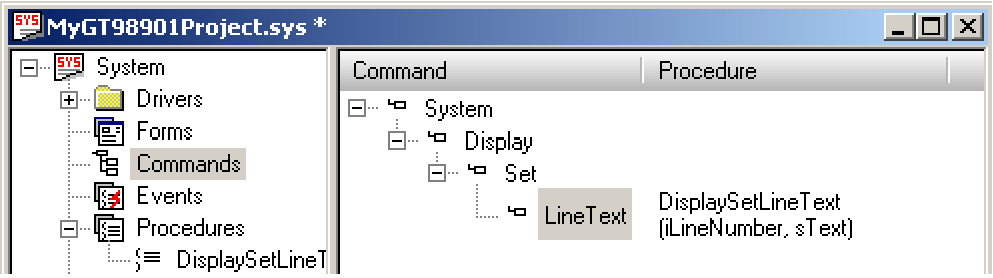
```
MyGTDEMO Set LineNumber (iLineNumber)
```

The command is inserted into the code view.

You have now finished writing the system procedure. Your next step is to create a command using the system procedure.

▼ To create the System command:

10. Select **Commands** from the tree view in the system document view. The commands view displays in the right pane.
11. Right-click on **System** and select **Insert Command Below** from the context menu. A new command item is created.
12. Rename the command item to **Display**.
13. Right-click on **Display** and select **Insert Command Below** from the context menu.
14. Rename the new command item to **Set**.
15. Insert another command item below **Set** and rename it to **LineText**.
16. Select **DisplaySetLineText** from the procedures list and click on the **Attach Procedure** button. The view should look as follows:



At this point, the system command is ready. You can insert it into a code view within a procedure or a test using the techniques learned here by:

Using the auto command completion.

Using the Insert System Command from the Insert menu or from the context menu.

Directly typing the command into the code view.

You can use the same techniques learned here to create program commands, which can be used in the program module.

Program Commands

In the following examples, Program Commands are used to set, apply and remove UUT power. Since the UUT power combination in this example applies only to the UUT tested by this program, program commands were used rather than System commands. Please note that this section is for reference only. You will not be creating any program commands in your example project.

Program Command	Performs the Following

Apply UUT Power	`PS1 Set Voltage (28)`
	`PS1 Set Current Limit (1.5)`
	`PS1 Set Output ON`
	`PS2 Set Voltage (5)`
	`PS2 Set Current Limit (3.25)`
	`PS2 Set Output ON`
	`RELAY Close (5)`
	`RELAY Close (6)`

Remove UUT Power	`PS1 Set Output OFF`
	`RELAY Open (5)`
	`PS1 Set Voltage (0)`
	`PS1 Set Current Limit (0)`
	`PS2 Set Output OFF`
	`RELAY Open (6)`
	`PS2 Set Voltage (0)`
	`PS2 Set Current Limit (0)`

In the first example, PS1 and PS2 are programmable power supplies being set to the correct voltage and current limit for a specific UUT. The power supplies' outputs are then turned ON and the outputs are applied to the UUT using a RELAY card.

The second example is a reverse of the first one where the power supplies are removed from the UUT and then reset to 0. Since you probably need to apply and remove power to/from the UUT several times during the program, these Program commands simplify programming and reduce debug and integration time.

Chapter 11 - Working with Forms

About Working with Forms

This chapter discusses Forms and Controls; how to create and use them. You will also learn about ***ATEasy*** Form Events, variables, and procedures. You will create a form to incrementally control and monitor all of the subsystems of the GT98901. This form does *not* depend on any of the modules you have already created. Use the table below to learn more about this chapter's topics.

Topic	Description
Overview of Forms	What the ***ATEasy*** forms are used for and what types of Forms are available?
The Form Development Process	Which steps are required for form development?
Creating a Form	Explains how to create a form and about the form view used to design and write code for forms.
Setting the Form Properties	Explains the various form properties and property pages.
Form Controls	Explains form controls and menus. Shows the Controls toolbar and provide an overview of the ***ATEasy*** built-in controls.
Adding Controls to Manipulate the LEDs	Shows how to add controls to the form.
Setting Controls Properties	Shows how to view and update properties specific to the target control.
Using Events	Explain what are events and how ***ATEasy*** calls them.
Writing an Event for the LEDs	Shows how to write control events to manipulate instruments.
Adding Controls to Monitor the DIP Switches	Shows how to write control events to monitor instruments.
Writing an Event for the Timer	Shows how to use a timer to automatically readback from an instrument and update your GUI.
The Load Statement	Explains about the **Load** statement used to create the form object.
Using the Form	Shows how to create a test used the load the form.
Testing the Form	Explains how to use the **Formit!** command to test the form.

Overview of Forms

Forms are one of the building blocks of ***ATEasy*** applications. Forms are windows or dialogs used to display data to the user in various formats and to provide interaction between the user who is using the application and the application itself.

Forms can include menus or controls. ***ATEasy*** provides an extensive library of ActiveX controls, as well as accepting any third-party control library of ActiveX controls.

ATEasy forms are commonly used to:

- Manage a test environment (a test executive)
- Display values and control a test instrument (a virtual instrument panel)
- Handle messages to the user such as text or virtual indicators (lights, analog and digital displays, progress bars, etc.)
- Display data such as test results in various formats (numerical, charts, graphs, etc.)

Forms are ***ATEasy*** submodules that can be placed into any one of ***ATEasy***'s modules: Driver, System, and Program. Forms are placed in the modules according to the function they serve:

Module	**Form Function**
Driver	Virtual panel – provides a way to control the instrument interactively
System	Control of an entire test system with many test programs and drivers (for example, a Test Executive)
Program	Display of information regarding a specific UUT (for example, instructions to connect test leads or flip switches)

The Form Development Process

The form development process contains multiple steps because forms themselves contain many elements. While the steps described here do not necessarily have to be followed in the order shown, these steps must be completed for the Form to be fully functional.

▼ To develop a form, the following steps should be performed:

Step	Description
Create a form	Adding a new form to the Forms submodule.
Add and arrange controls and menus	Adding the required controls and menus to the form and arrange their layout on the form.
Set the form, controls and menus properties	Setting the default properties for these objects. These properties may be modified during run-time by statements from your code as a response to an event or by test or procedures code.
Write code to events	Filling in the form, control and menu event procedures to respond to user actions.
Add form variables	Creating form variables used by the form events and procedures or externally by tests and module procedures.
Write form procedures	Writing form procedures that can be called by form events or externally by tests and module procedures to perform additional tasks required by the form.

▼ To use a form, the following steps should be performed:

Step	Description
Create a form variable	Creating a variable in your module or in a procedure and setting its type to the form name as it appears under the form submodule.
Load the form	Adding a Load statement to display your form on the screen in your test or procedure. The Load statement uses the form variable created.
Interact and Test the form functionality	Verifying the form functions properly by interacting with the form menus and controls. Writing code to set and get the form properties, procedures, and variables using the form variable.

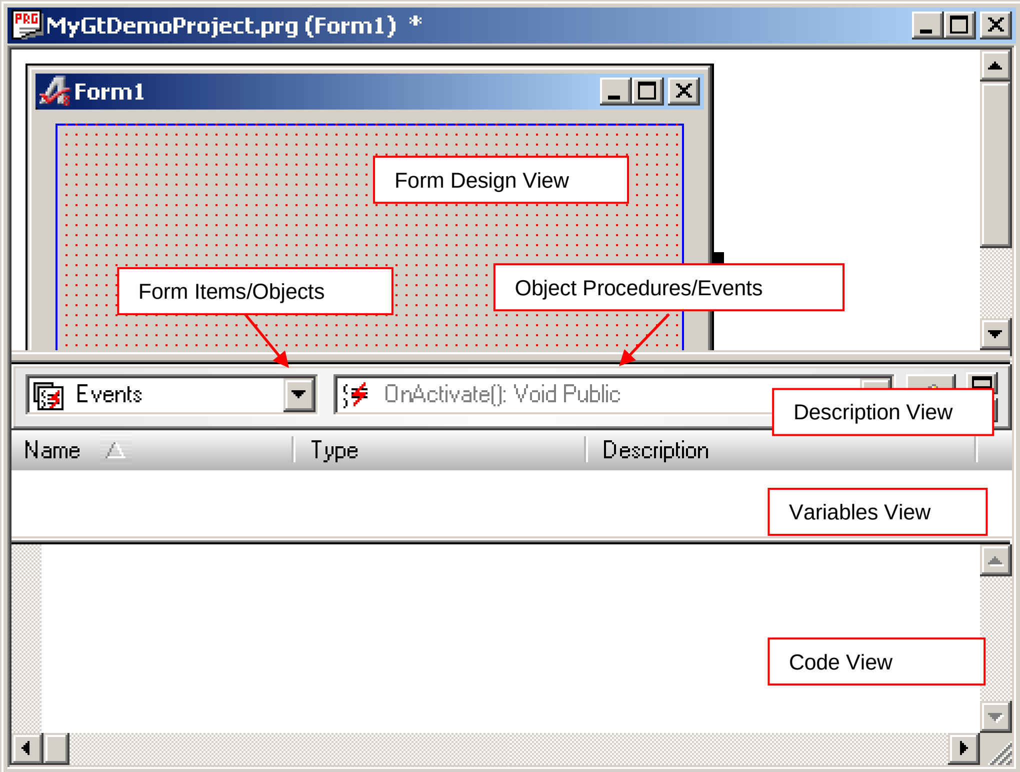
Creating a Form

The first step in the form development process is to create the form. In this example, you will create a form within an ***ATEasy*** Program. Forms can also be created under other ***ATEasy*** modules such as System or Drivers. Open the MyGtDemoProject that we created earlier and continue the process to begin working with forms.

▼ To create a form:

Right-click on the **Forms** submodule under the Program module and select **Insert Form Below** from the context menu. A new Form called Form1 is created. Rename the new Form to **MyForm**.

Creating the new form causes ***ATEasy*** to display the Form View as shown here:



The top pane of the form view contains the Form Design View showing the form and its controls and menus. A grid used for control alignment is shown on the form client area; in addition, blue margin lines are shown on the form. The margin is used to limit the area where controls can be placed using a mouse on the form. The grid can be adjusted by using the **Grid and Margins** command below the **Arrange** command on the **Edit** menu. The Margins can be also dragged using the mouse to adjust the distance from the form border.

The area below the form design view displays two drop-down lists. The left drop-down list shows the **Form Items** that can be edited including the form events, procedures and variables, as well as the controls and menus included in the form. Selecting an item from the Items drop-down list will refresh the second drop-down list and displays the procedures of the selected item. Selecting a procedure from the procedures combo box makes that procedure the **current procedure**.

The area below the combo box controls displays the **current procedure description**, **variables and parameters**, and the **procedure code**.

Setting the Form Properties

The form's properties window has several pages defining the different aspects of the form. The most important elements are found on the **General** page. Here, the Form Name and type are defined as well as the form's caption, default menu bar, size and whether the form is public (that is, can be used by other modules)

Additional Form properties pages include:

- **Window** – contains properties to determine the border style, its initial position, state and other window properties.
- **Drawing** – contains the default drawing attributes such as pen, fill style, draw width and more. This is used when using the drawing form procedures to draw on the form.
- **Scale** – contains the scale mode. The default scale mode is pixels. Scale mode is used to specify different units for the coordinate system.
- **Misc.** – contains help files support for a form.
- **Pictures** – allows you to set a background picture for the form.
- **Colors** – contains properties to set the foreground and background colors of the form. The foreground color is used when drawing text and lines on the form. The background color is used to paint the client area.
- **Fonts** – contains a font selection that is used as the default font for controls. In addition, the font is used when drawing text on the form.

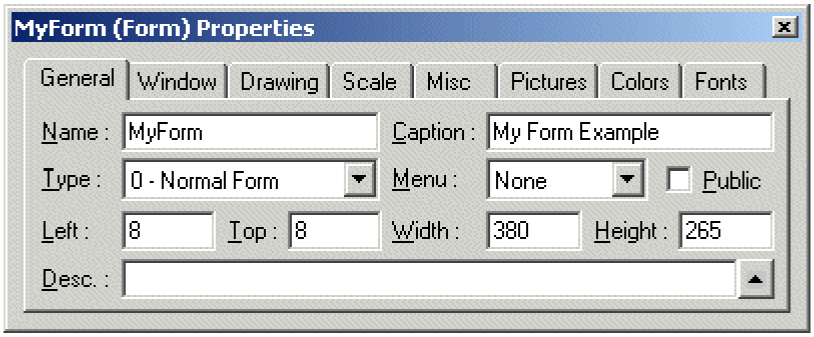
As you can see, the form contains many properties. The ***ATEasy*** *User's Guide* and the *Reference Guide* cover them in more detail.

▼ To set the Form Properties:

Open the MyForm Properties window by clicking the right mouse button on **MyForm** and selecting **Properties**. The properties window opens.

Rename the form to **MyForm**.

Change the caption to **My Form Example**. The caption displays in the title bar of the form. The properties window should look similar to the following:



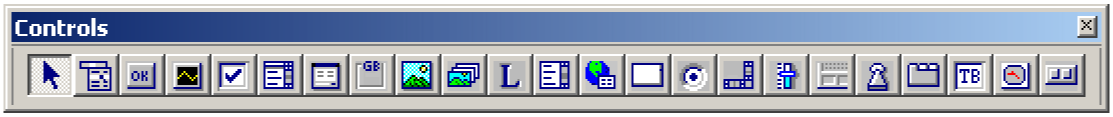
MyForm (Form) Properties

Name : MyForm
Caption : My Form Example
Type : 0 - Normal Form
Menu : None
□ Public
Left : 8
Top : 8
Width : 380
Height : 265
Desc. :

Form Controls

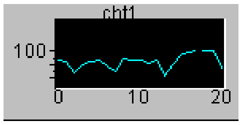
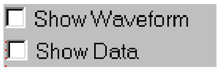
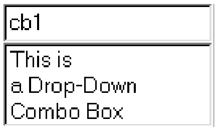
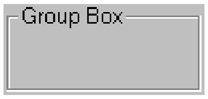
Forms need to be populated with controls and menus in order to be useful. Controls are the interface elements through which the end user makes choices or obtains information. Buttons, text boxes, checkboxes, list boxes, and charts are all examples of controls supplied with ***ATEasy***. Controls have their own properties, methods and events to make them suitable for particular purposes; for example, displaying text, or allowing the user to scale a value.

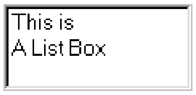
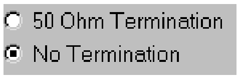
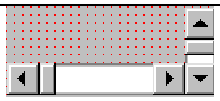
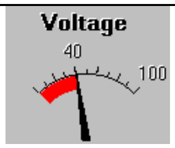
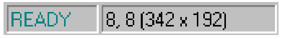
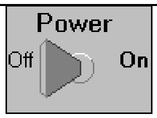
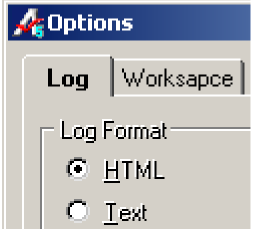
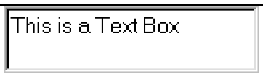
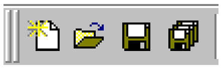
Controls and menus are added from the **Controls** toolbar. The Controls toolbar appears on the screen whenever the form view is active. It contains all the available controls that can be placed on a form as shown here:



The first button in the toolbar (appears on the left side) is the **selection tool**. The second button (showing a menu) is used to add a **menu** to the form. The rest of the buttons are controls. To add any one of these controls, click on it, then click in the form client area, and drag the control to the appropriate size.

ATEasy provides the following controls:

Type	Description	Appearance	Tool
tton	Typical Windows button with some added features. Used for confirmation (OK, Cancel, etc.)		OK
AChart	Displays a set of Y-data versus a set of X-data using one of several predefined plot templates.		
ACheckBox	A check box indicates whether a particular condition is on or off. Use check boxes in an application to give users true/false or yes/no options.		☑
AComboBox	Combines the features of a Text Box and a List Box. Allows the user to select either by typing text into the Combo Box or by selecting an item from its list.		
AGroupBox	Used as a frame to group several controls together.		GB
AImage	Displays a graphic image.		
AImageList	Does not have any user interface. Stores a list of images to be used by the AStatusBar control.	N/A	
ALabel	Displays text. Also, used to label controls such as AListbox to describe their content.		L

Type	Description	Appearance	Tool
AListBox	Allows the user to select an item from a given list.		
ALog	Based on Internet Explorer. Provides a versatile way to record test results. Test data can be routed to the Log Control (instead of the ***ATEasy*** standard Log) in plain Text or HTML formats.		
APanel	Container control. Used to group several controls under same container. Hiding this control will hide all its controls.		
ARadioButton	Presents a set of two or more choices to the user. Unlike check boxes, radio buttons work as part of a group; selecting one radio button immediately clears all the other buttons in the group.		
AScrollBar	Horizontal and vertical scroll bars allow you to select a value by moving the scrollbar thumb.		
ASlider	A control containing a thumb, numerical, and text labels. Supports a variety of styles from three main groups: slider, knob, and meter (shown).		
AStatusBar	Used for displaying a status bar on a form. The status bar contains panes that can display text, images, keyboard state and more.		
ASwitch	Represents a switch with enhanced style and mode features. The style can be as a toggle (shown), a slide, LED, push button, and more.		
ATab	Allows definition of multiple pages for the same area of a form. Each page consists of a certain type of information or a group of controls that the application displays when the user selects the corresponding tab.	 Options Log Worksapce Log Format ● HTML ○ Text	
ATextBox	Can be used to get text input from the user or to display text.		TB
ATimer	Does not have any user interface. Used for generating events periodically. Can be used to refresh the controls on a form periodically.	N/A	
AToolBar	Used for displaying a toolbar on a form. The toolbar can contain buttons, check button, group buttons and menus.		

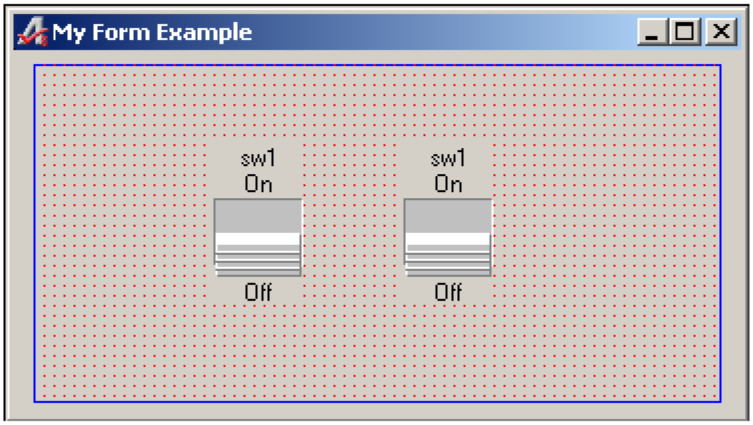
Adding Controls to Manipulate the LEDs

In this example, you will be adding three ASwitch controls and an ATimer control on the left to monitor the GT98901's DIP switches and two ASwitch controls on right of your form to manipulate the demo board's LEDs. We will begin by adding the controls for the LEDs, setting their properties and adding events.

▼ To add the controls to the form:

Click the **ASwitch** control on the Controls toolbar. Position your mouse over the form and then click and drag a small rectangle. This will create a new ASwitch control.

Copy the switch. To copy, click on the button while holding down the CTRL key. Drag the button to a position immediately below the first button. When you are finished, your form will look similar to the following:



You now have all the controls you need and its time to set the visual appearance of the Form. Next, you will need to place the controls and space them evenly. You also need to make the button height and width to be the same.

▼ To adjust the size and location of the controls:

Click on **sw1** to select the control. You can move the control using the keyboard's arrow keys. This is more precise than moving the control by dragging it with the mouse. You can also size the control by dragging the selection handles or using the keyboard by pressing the SHIFT key down and pressing the arrow keys.

If you grab the edge of a control, the new height and width of the will be displayed next the to the mouse current in the following format: Left, Top (Width x Height). Adjust **sw1** so that it is 45 pixel wide by 60 pixel tall.

Once sw1 is in the correct position, you can make **sw2** align equal to the top edge of **sw1**. To do that you must select multiple controls. Click on **sw2** to select the control, press the **SHIFT** key, and then click on **sw1**. Notice that the two controls are now selected. The last control that you select is called the **pivot control**. Click on the **Align Top** command from the **Form Design toolbar** or from the **Arrange** menu under the **Edit** menu. Notice that sw2 is now aligned equal to the top edge of sw1.

While the two buttons are still selected, make sw2 have the same width and height of sw1. Click on the **Make Same Width** command and then click on the **Make Same Height** command (alternatively, you can select the **Make Same Width and Height** command). At this point sw1 and sw2 are aligned and are the same size.

Setting Control Properties

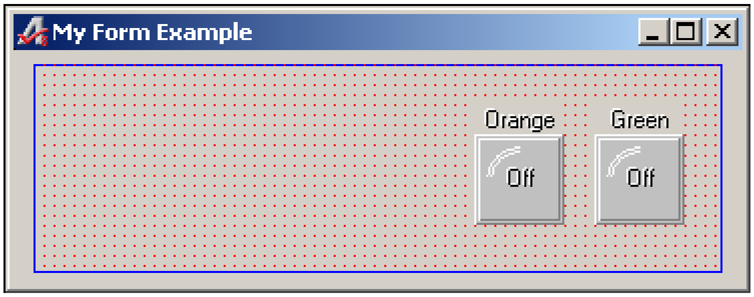
Before you can continue, you need to change some of the properties of the controls. One property to change is the Control's name. ***ATEasy*** assigns default names to Controls such as btn1, btn2, etc. Controls must have different names in order for ***ATEasy*** to be able to access them individually. Since the default names do not mean much, you should change them to a meaningful name describing the Control's use. However, you should keep the prefix in order to have standard naming conventions and to be able to distinguish between different types of Controls.

As an example, you should use the prefix "**sw**" for all Switch Controls. This way, when you see a reference in the program to **MyForm.swGreen**, you know it refers to a Switch Control probably called "**Green**."

▼ To set the control properties for the buttons:

1. Double-click on **sw1** to display its properties. The switch property page displays.
2. On the General properties page, change the Name to **swOrange**.
3. Click on the **Control** page to activate the page. Change the Caption to **Orange**. Change the Font3D property to **1 – raised w./light shading**. This sets a unique look to the font text on the buttons in your form. Change the Style to **Square LED**.
4. Click on the **Colors** page now. Change the OffPictureColor to **Gray**. Change the OnPictureColor to **Red**. This will cause the control to be Gray when the switch is off and Red when it is on.
5. Select **sw2** and repeat step two, changing the Name to **swGreen**.
6. Repeat step three for **swGreen**, changing the Caption to **Green**.
7. Repeat step four for **swGreen**, changing OnPictureColor to **Green**.

Your form should appear similar to the following:



Using Events

ATEasy forms, controls and menus generate notification messages to your application when a certain condition occurs. This can be when the user moves the mouse on top of the object, when the user clicks on the object, and more.

You can respond to the message by placing code in the event procedure that is associated with the notification message. ***ATEasy*** will only call event procedures that have code. If you leave the event empty, ***ATEasy*** will not call the event. The code that you do place in your event should be short. This is because when the event code is executing, no other events are sent to the form. In addition, if a test program is running while the form is executed, the test program is suspended until the event is complete.

Form Events are events related to the form itself and not to the controls in the form or the menus. Form Events include **OnLoad** – when the Form is initially loaded, **OnClick** – when the mouse is used to click on the form (in an area without controls), **OnResize** - when the form is sized, and more. When the notification arrives, ***ATEasy*** calls the message you programmed the form to use. For example, you can program the Form to change its Caption or its background when the Form is selected.

Control and menu events are similar except they refer to notification messages received from the control. For example, the **AButton** control has an **OnClick** event, the **ATimer** control has an **OnTimer** event, and so forth.

Writing an Event for the LEDs

You will now add a simple event to MyForm. This event controls what happens when the Green switched is clicked.

▼ To write the swOrange.OnChange Event:

1. From the form view, select **swOrange** from the form items combo box. The right combo box now displays the available events for the **ASwitch** control.

2. When the user clicks of the switch, it will toggle the state from OFF to ON or vice versa and the switch's OnChange event will be fired. Select the **OnChange** event. Click in the procedure space and begin typing:

   ```
   !Update the LED state
   if Control.Value=aswValueOn
      GTDEMO Led Set State On(aLedOrange)
   else
      GTDEMO Led Set State Off(aLedOrange)
   endif
   ```

   This code will check the state of the switch control and use that value to update the LED's state.

3. Now, if you collapse the right combo box you will see that all events are shown with gray text while **OnChange** now appears normal text, showing that the event is used.

4. Add similar code to **swGreen.OnChange** event to toggle the green LED on and off.

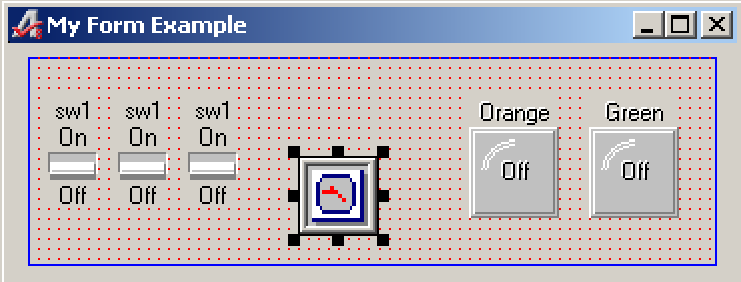
5. Before you continue, you should test your switches and events. Click the **FormIt!** Button on the toolbar or access it from the Debug Menu. This will launch the form and let you test the switches by clicking them. As they change state, you should see the LEDs on your GT98901 toggle on and off.

Adding Controls to Monitor the DIP Switches

Next, we will add the controls to support the dip switch monitoring.

▼ To add the controls to the form:

1. Click the **ASwitch** control on the Controls toolbar. Position your mouse over the form and then click and drag a small rectangle. This will create a new ASwitch control.
2. Do this three times to create three ASwitch controls on the left hand side of the screen.
3. Click the **ATimer** control on the Controls toolbar. Position your mouse over the form and then click and drag a small rectangle. This will create a new ATimer control. Note: The ATimer control will only show up at design time and at run-time the control will not be visible.



You now have all the controls you need and its time to set the visual appearance of the Form. Next, you will need to place the controls and space them evenly. You also need to make the button height and width to be the same.

▼ To adjust the size and location of the controls:

4. Click on **sw1** to select the control. You can move the control using the keyboard's arrow keys. This is more precise than moving the control by dragging it with the mouse. You can also size the control by dragging the selection handles or using the keyboard by pressing the SHIFT key down and pressing the arrow keys.
5. If you grab the edge of a control, the new height and width of the will be displayed next the to the mouse current in the following format: Left, Top (Width x Height). Adjust **sw1** so that it is 25 pixel wide by 55 pixel tall.
6. Once sw1 is in the correct position, you can make **sw2** and **sw3** align equal to the top edge of **sw1**. To do that you must select multiple controls. Click on **sw3** to select the control, press the **SHIFT** key, and then click on **sw2** and then **sw1**. Notice that the two controls are now selected.

   The last control that you select is called the **pivot control**. Click on the **Align Top** command from the **Form Design toolbar** or from the **Arrange** menu under the **Edit** menu. Notice that sw3 and sw2 is now aligned equal to the top edge of sw1.
7. While the two buttons are still selected, make sw2 have the same width and height of sw1. Click on the **Make Same Width** command and then click on the **Make Same Height** command (alternatively, you can select the **Make Same Width and Height** command). All the switches should be at the same position.

8. To set the properties for the new switches and timer:

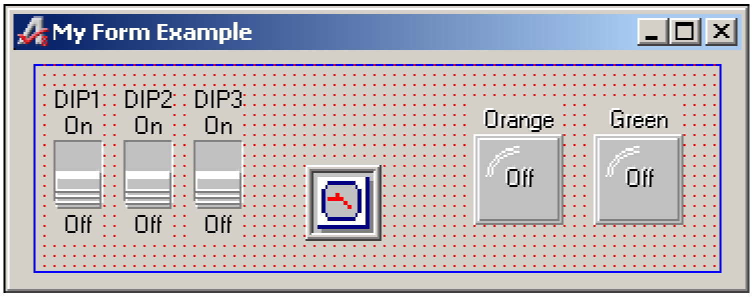
9. Double-click on **sw1** to display its properties. The switch property page displays. Change the name of the control to **swDip1** and change the caption to **DIP1**. Change the mode in the **Controls** page to **0 – Indicator**. This ensures that the user cannot update this value by clicking on it.

10. Double-click on **sw2** to display its properties. The switch property page displays. Change the name of the control to **swDip2** and change the caption to **DIP2**. Change the mode in the **Controls** page to **0 – Indicator**. This ensures that the user cannot update this value by clicking on it.

11. Double-click on **sw3** to display its properties. The switch property page displays. Change the name of the control to **swDip3** and change the caption to **DIP3**. Change the mode in the **Controls** page to **0 – Indicator**. This ensures that the user cannot update this value by clicking on it.

12. Double-click on **tmr1** to display its properties. Rename it to **tmrUpdate**. Change the interval in the **Controls** page to **500**.

Your form should appear similar to the following:



Writing an Event for the Timer

The timer will fire its OnTimer event every X milliseconds, X is equal to the time interval which is a property of the ATimer control. What we will do is write code so the state of the switches in the form will update to reflect the physical state of the demo board.

▼ To write the tmrUpdate.OnTimer Event:

1. From the form view, select **tmrUpdate** from the form items combo box. The right combo box now displays the available events for the **ATimer** control.

2. When the timer is enabled, it fires the OnTimer event periodically. Select the **OnTimer** event. Click in the procedure space and begin typing:

```
!Retrieve the state of the dip switches and store in local variables
GTDEMO DipSwitch Get State(aDipSwitchChannel1, bDip1)
GTDEMO DipSwitch Get State(aDipSwitchChannel2, bDip2)
GTDEMO DipSwitch Get State(aDipSwitchChannel3, bDip3)
!update dip switch 1
if bDip1
   swDip1.Value=aswValueOn
else
   swDip1.Value=aswValueOff
endif
!update dip switch 2
if bDip2
   swDip2.Value=aswValueOn
else
   swDip2.Value=aswValueOff
endif
!update dip switch 3
if bDip3
   swDip3.Value=aswValueOn
else
   swDip3.Value=aswValueOff
endif
swDip3.Value=bDip3
```

3. Add some local variables to the OnTimer event. These are the variables which the dip switch state is saved to. All new variables will be of type Boolean and the names are **bDip1**, **bDip2**, **bDip3**.

4. Before you continue, you should test the newly created controls and events. Click the **FormIt!** Button on the toolbar or access it from the Debug Menu. This will launch the form, try changing the state of the dip switch and monitoring the response on the form.

The Load Statement

At this point, your form is complete. The next step is to write code to load the form from the program. Loading a form to display it is done using the **load** statement. The load statement has three parameters.

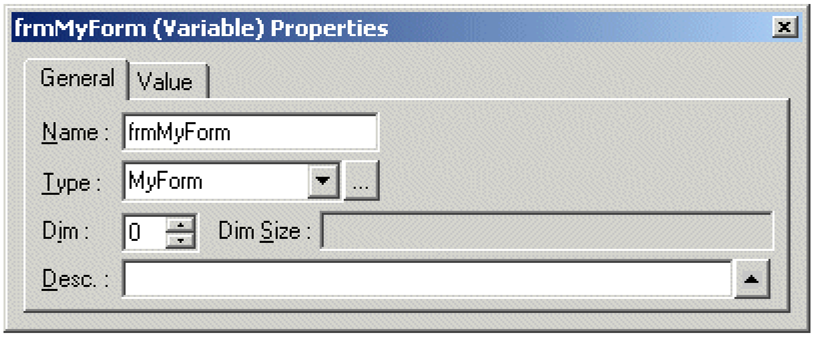
- The first parameter is the **form variable**, which is used to hold the form object. The variable type name should have the form name as appears below the Forms submodule.
- The second parameter is optional and indicates whether the form is created using the **modal** or **modeless modes**. When using **modal mode,** the load statement does not return to the caller until the form is unloaded. The form may be unloaded either by using the **unload** statement or when the user clicks the X button in the form title bar (if available). Modeless form returns immediately to the caller and the form runs in parallel to the code running after the load statement.
- The third and optional parameter is the form's parent window handle. A window handle is a 32-bit number used to uniquely identify the window in the current process. Each form has a handle that can be retrieved using the hWnd property. Passing 0 as a handle (or omitting the argument) uses the Windows desktop window. Forms created are always displayed on top of their parent. Additionally, when the parent is destroyed, the form is also destroyed. When a form is created using modal mode, the parent is disabled after the form is created and enabled, and is activated as the form closes.

Using the Form

In this example, load the form within a new test you will create in **MyProgram**. You will also create a variable that will have **MyForm** as a type.

▼ To use the form:

1. Define the form variable. Insert a new variable under the MyProgram variables submodule. Rename it to **frmMyForm**. Set its type to be **MyForm** as shown here:



frmMyForm (Variable) Properties

Name : frmMyForm

Type : MyForm

Dim : 0

Dim Size :

Desc. :

2. Insert a new Task in **MyProgram**. Name the task **Forms** and the test as **MyForm**.
3. In the MyForm test code view, type the following lines of code:

```
! creates the form in modal mode
Load frmMyForm, TRUE
! deletes the form object
frmMyForm=Nothing
```

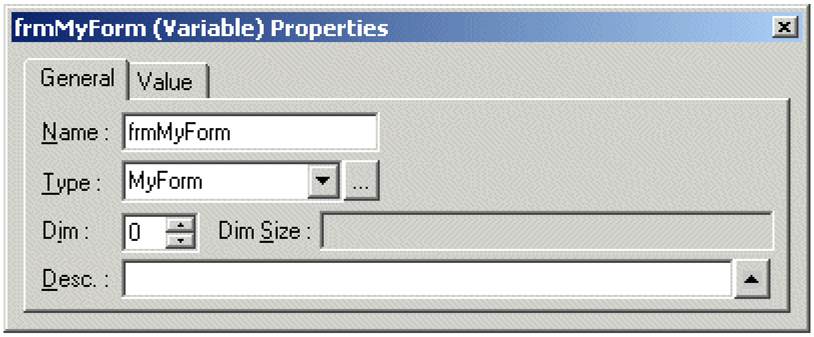
The first statement loads the form in modal mode and uses the windows desktop as a parent. The second statement deletes the form object releasing all resources associated with the object. This statement will be executed after the form window is destroyed. Note that before destroying the form object, you can still use form **public** variables and procedures (for example, frmMyForm.m_iAcquire if it was declared as public).

Using the Form

In this example, load the form within a new test you will create in **MyProgram**. You will also create a variable that will have **MyForm** as a type.

▼ To use the form:

1. Define the form variable. Insert a new variable under the MyProgram variables submodule. Rename it to **frmMyForm**. Set its type to be **MyForm** as shown here:



Name : frmMyForm
Type : MyForm
Dim : 0
Dim Size :
Desc. :

2. Insert a new Task in **MyProgram**. Name the task **Forms** and the test as **MyForm**.

3. In the MyForm test code view, type the following lines of code:

```
! creates the form in modal mode
Load frmMyForm, TRUE
! deletes the form object
frmMyForm=Nothing
```

The first statement loads the form in modal mode and uses the windows desktop as a parent. The second statement deletes the form object releasing all resources associated with the object. This statement will be executed after the form window is destroyed. Note that before destroying the form object, you can still use form **public** variables and procedures (for example, frmMyForm.m_iAcquire if it was declared as public).

Testing the Form

You can now execute the code to test your form.

▼ To run the form test code:

1. Make sure the test code is the active view. Select **Doit!** from the Debug Menu or from the Build/Run toolbar.

ATEasy provides additional tools to test the form. You can use the **Formit!** from the **Debug** menu or from the toolbar. The command executes a load statement on a temporary variable that ***ATEasy*** will create.

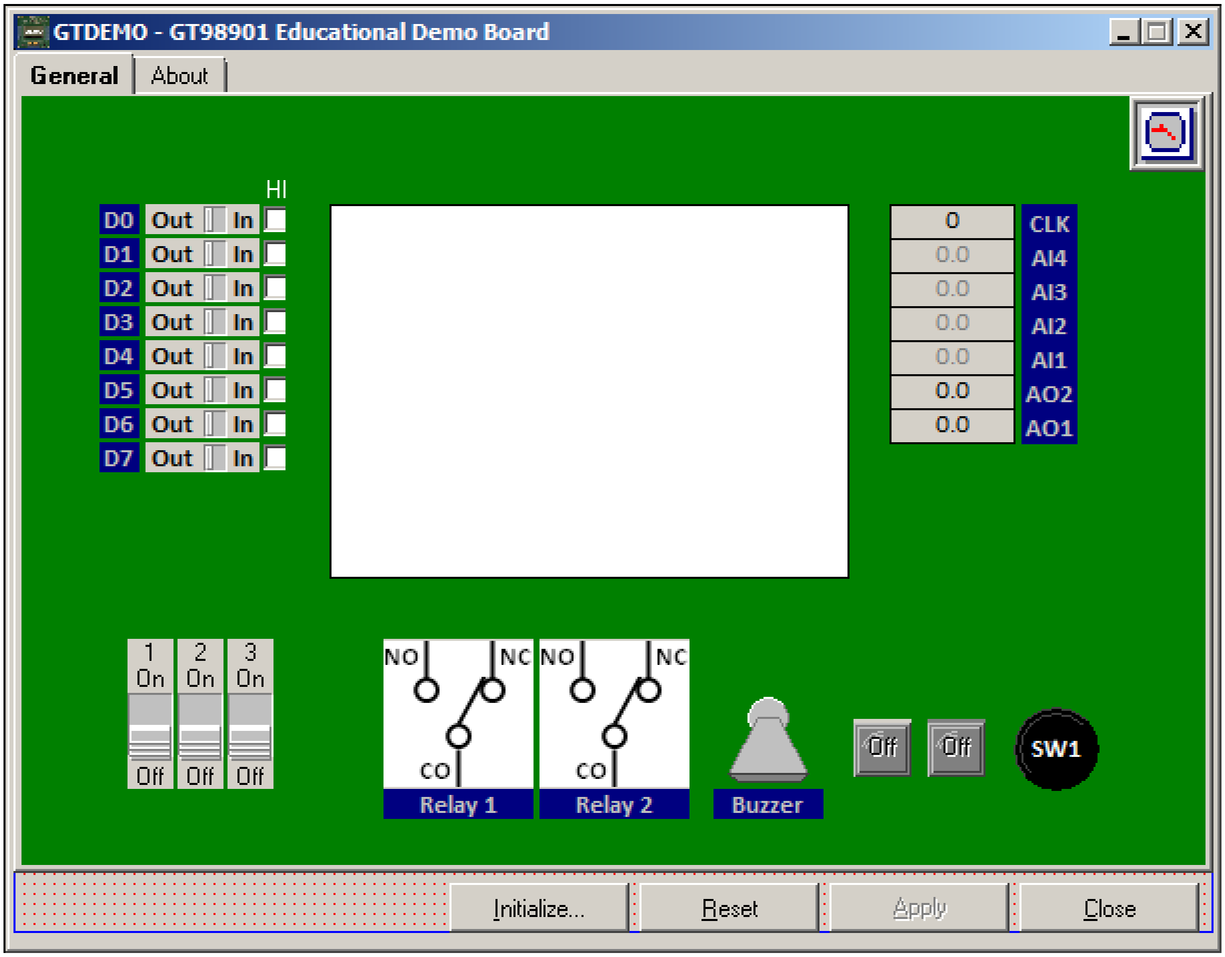
Using the GtDemo Panel as a Form-Building Example

In the official GtDemo.drv that is included with the GTDEMO software package is a complete software panel example. This form allows the user to operate each of the demo board's subsystems the control events contain code, which can be reused in projects of your own.

▼ To access the GtDemo.drv Panel:

1. Locate the GtDemo.drv in either the GTDEMO under Program Files directory or the ATEasy Drivers directory.
2. Open the GtDemo.drv in ATEasy.
3. In the GTDEMO driver's Forms submodule, double-click **PanelForm**.

The PanelForm that is included with the GtDemo ATEasy driver is shown below:



The controls and events that were added to the form are grouped by subsystem. Controls which belong to a common subsystem tend to function similarly. For example, the eight switches and checkboxes in the top-left side of the panel all control the digital input/output subsystem. Each of the eight switches perform the function of updating the currently programmed direction of the digital ports based on user-driver events. Double-clicking a control within the ATEasy form editor will open the code editor for the selected event and will allow the user to examine and alter the operation of the form.

▼ To view the DIO direction control code:

4. In the PanelForm form editor, double-click **Out/In** slider located to the right of the **D0** label.
5. Below the form editor, the code editor should have updated to represent the selected control, **swDIODirection0**, and the default control event, **OnChange**.
6. The OnChange event will execute each time the value of the swDIODirection0 switch is changed, or whenever the event is called explicitly in code. The code within allows the user to toggle the direction of a single digital port using the following code:

```
Driver Digital Get Direction(nDirection)
!Clear the target bit and set new value
ucDirection=nDirection and 0xFE
ucDirection=ucDirection or (Control.Value shl 0)
Driver Digital Set Direction(ucDirection)
```

Since the direction of all ports is written at once, changing the direction of a single port entails getting the direction of all the ports, then masking out and updating the target port. All of the digital direction switches will have similar code, but will have different masks and target values.

▼ To view the DIO data control code:

7. In the PanelForm form editor, double-click the checkbox located to the right of the **D0** label.
8. Below the form editor, the code editor should have updated to represent the selected control, **chkDIOValue0**, and the default control event, **OnChange**.
9. The code below executes when the checkbox is toggled:

```
Driver Digital Read Data(wValue)
ucValue=wValue and 0xFE
if Control.Value=achkChecked
      ucValue=ucValue or 0x1
endif
Driver Digital Write Data(ucValue)
```

This functions in the same manner as the digital direction control. The current data values are read from the microcontroller and the value to be updated is masked off and set or reset.

▼ To view the relay control code:

10. In the PanelForm form editor, double-click the image depicting an SPDT relay located above the **Relay 1** label.
11. Below the form editor, the code editor should have updated to represent the selected control, **swRelay1**, and the default control event, **OnChange**.

12. The OnChange event will execute each time the SPDT image is clicked. If you examine the properties of **swRelay1** you will see that the OnPicture and the OffPicture are non-standard. Each time the switch is toggled it will change which picture is being displayed. The OnChange code is listed here:

```
!Toggle value
if Control.Value=aswValueOff
   Driver Relay Switch Close(aRelayK1)
else
   Driver Relay Switch Open(aRelayK1)
endif
```

This code simply reads the current value of the control that the user clicked and updates the relay based on the new value. The other relay, the buzzer, and the LED switches all operate in the exact same manner.

▼ To view the code for updating and formatting the LCD display:

13. In the PanelForm form editor, double-click the large textfield in the center of the form that has several numbers 0 through 9 written in it..

14. Below the form editor, the code editor should have updated to represent the selected control, **tbLCD**, and the default control event, **OnChange**.

15. The OnChange event executes each a user changes the text in the textfield, by adding or deleting any of the characters contained within the control. The OnChange code is listed here:

```
lCursorPosition=Control.SelStart
sOrig=Control.Text
for lIndex=1 to 8
   lPos=Pos("\r\n", sOrig, lStart)
   if lPos=-1 or (lPos-lStart)>21
         sNew=sNew+Mid(sOrig, lStart, 21)+"\r\n"
         lStart=lStart+21
   else
         sNew=sNew+Mid(sOrig, lStart, lPos-lStart)+"\r\n"
         lStart=lPos+2
   endif
next
Control.Text=sNew
Control.SelStart=lCursorPosition
btnApply.Enabled=True
```

This code does not immediately update the LCD display on the GT98901. Instead it formats the text that the user has entered to the constraints of the LCD display. The display can only show 21 characters per line, so this code scans through the contents of the textfield control and places a CR/NL every 21 characters. It also enables the **Apply** button which will be used to send the data to update the physical display. The **Apply** button is also enabled when the user updates the AO1, AO2 or the CLK buttons.

▼ To view the code for updating changes via the Apply button:

16. In the PanelForm form editor, double-click the button labeled **Apply** at the bottom of the form.

17. Below the form editor, the code editor should have updated to represent the selected control, **btnApply**, and the default control event, **OnClick**.

18. The OnClick event executes each time a user clicks the Apply button. The OnClick code is listed here:

```
Driver Analog Output Write Single(aAnalogOutputChannel1,
Val(tbAO1.Text))
Driver Analog Output Write Single(aAnalogOutputChannel2,
Val(tbAO2.Text))
s=tbLCD.Text
lStart=0
for lIndex=1 to 8
   Driver Display Set LineNumber(lIndex)
   lPos=Pos("\r\n", s, lStart)
   if lPos=-1 or (lPos-lStart)>21
         Driver Display Set Text(Mid(s, lStart, 21))
         lStart=lStart+21
   else
         Driver Display Set Text(Mid(s, lStart, lPos-lStart))
         lStart=lPos+2
   endif
next
Control.Enabled=False
```

The Apply button only becomes active after a change has been made to the analog output channels or to the LCD display textfield. When clicked, it writes the data that is in the AO1 and AO2 textboxes to the GT98901. Then it parses the text in **tbLCD** and sends it line-by-line to the GT98901.

▼ To view the code for returning the board to a default state:

19. In the PanelForm form editor, double-click the button labeled **Reset** at the bottom of the form.
20. Below the form editor, the code editor should have updated to represent the selected control, **btnReset**, and the default control event, **OnClick**.
21. The OnClick event executes each the user clicks the Reset button. The OnClick code is listed here:

```
Driver System Reset()
swRelay1.Value=aswValueOn
swRelay2.Value=aswValueOn
swBuzzer.Value=aswValueOff
swGreen.Value=aswValueOff
swOrange.Value=aswValueOff
tbAO2.Text="0.0"
tbAO1.Text="0.0"
```

This code first resets the board using the GT98901's reset procedure. Then each of the configurable controls are changed to their default state manually.

▼ To view the code for reading back input data:

22. In the PanelForm form editor, double-click the timer control located in the top-right corner of the form.
23. Below the form editor, the code editor should have updated to represent the selected control, **tmrUpdate**, and the default control event, **OnTimer**.
24. Right-clicking the **tmrUpdate** control and viewing the properties window will show the interval for this control. The interval is the amount of time in milliseconds that will pass between **OnTimer** events. By default, it will be set to **500** which means that the software panel will update twice a second. The code for this event is very long but the operations that are performed are quite simple. Each subsystem is read using the appropriate GtDemo Get functions and the proper controls are updated with the latest data.

Chapter 12 - Additional Exercises

About Additional Tutorials

This chapter provides additional exercises which demonstrate the development of test applications. The exercises can be used at various points within this manual to supplement the material presented:

Topic	Description
Exercise 1 – Using the Application Wizard	Tutorial detailing the creation of a test application using the built-in ATEasy Application Wizard.
Exercise 2 – Creating Modules Manually	Creating a test application by adding the module files individually.
Exercise 3 – Writing Test Code	Introduction to the Tests sub-module of the Program module, used for the creation of tests.
Exercise 4 – Profile and Test Executive	An overview of the Test Executive and Profile software drivers and how they can be added to projects.
Exercise 5 – Flow-Control Statement	Tutorial detailing the use of ATEasy's language syntax to simplify coding.
Exercise 6 – Module Events	Introduction to the Events sub-module and its function in a test application.
Exercise 7 – Using Procedures	Introduction to the Procedures and Variables sub-module and the order of access for public variables.
Exercise 8 – Creating an ATEasy Driver	Tutorial detailing how to write an ATEasy driver module when starting from nothing.
Exercise 9 – Using Commands	Introduction to utilizing the Commands sub-module.
Exercise 10 – Importing an External DLL Library	Introduction to adding external library to the Libraries sub-module.
Exercise 11 – Form Creation and Use	Introduction to the Forms sub-module, the creation of custom Form class, and the loading of form instances.

Exercise 1 – Using the Application Wizard

Objectives

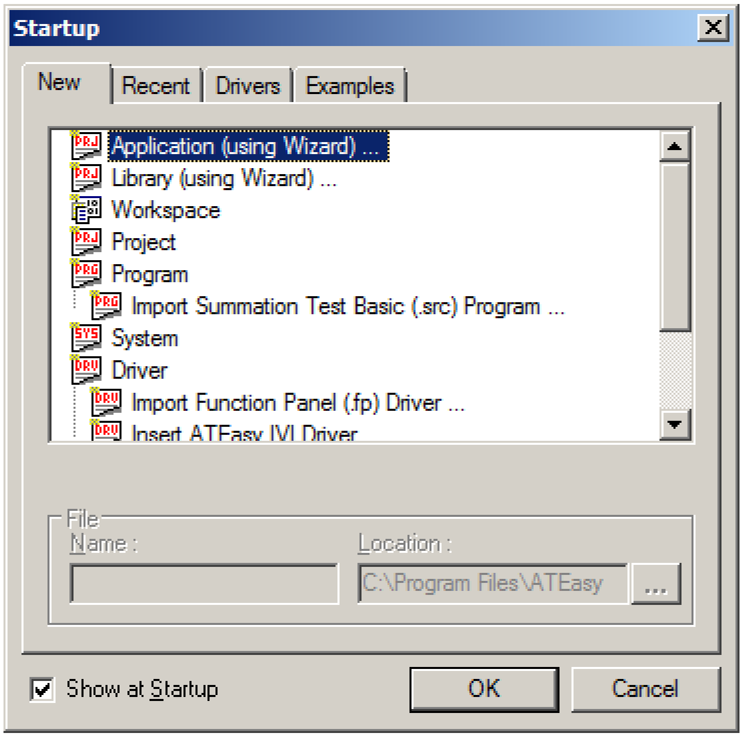
Create a test application using the ATEasy Application Wizard.

Run your newly created application.

In this exercise, we will create a complete ATEasy test application using the ATEasy Application Wizard. An ATEasy application consists of project, program, system and driver files. The wizard will quickly create these files and optionally include additional software drivers to give the user a starting point for application development

▼ Use the Application Wizard to create an ATEasy Test Application

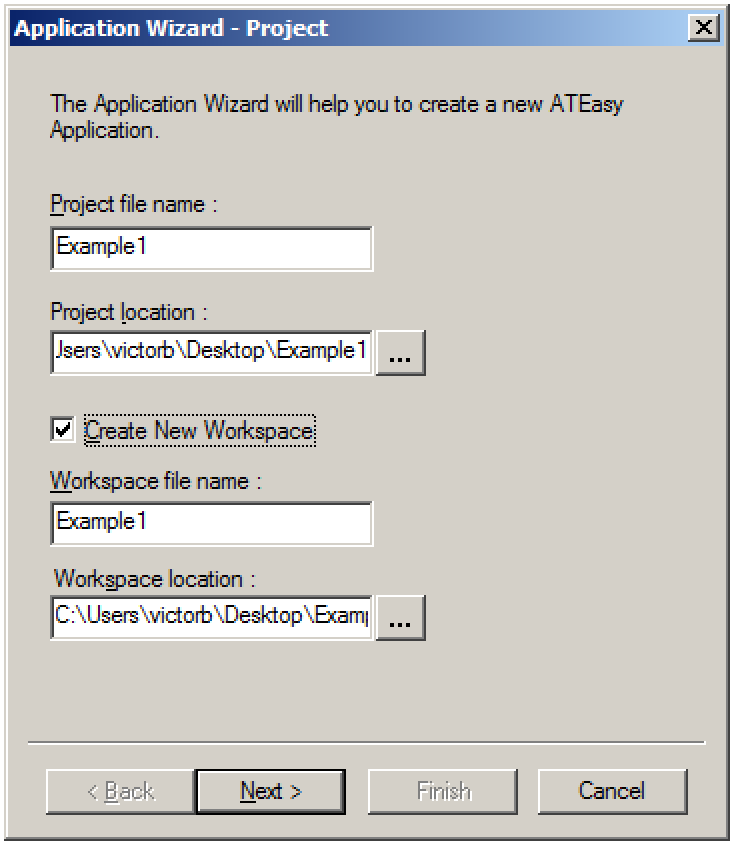
1. Create a folder on the Desktop called **Example1**.
2. Open ATEasy from the Desktop icon or from the start menu: **Programs | ATEasy**.
3. Depending on the workspace configuration, the startup window will appear automatically and show you the recently opened workspace. **Click** the **New** tab. If the startup window is not open, select **File | New** from the file menu.


Startup
New
Recent
Drivers
Examples
Application (using Wizard) ...
Library (using Wizard) ...
Workspace
Project
Program
Import Summation Test Basic (.src) Program ...
System
Driver
Import Function Panel (.fp) Driver ...
Insert ATEasy IVI Driver
File
Name :
Location :
...
Show at Startup
OK
Cancel

Location : C:\Program Files\ATEasy

☑ Show at Startup

4. Highlight Application (using Wizard) … and click OK.
5. For the Project file name, enter Example1.
6. For the Project location, enter the location of the Example1 folder that you created on the Desktop in step 1. If you type in
7. Click the **Create New Workspace** checkbox. This will copy the file name and location from the project fields.



Application Wizard - Project

The Application Wizard will help you to create a new ATEasy Application.

Project file name : Example1

Project location : Jsers\victorb\Desktop\Example1

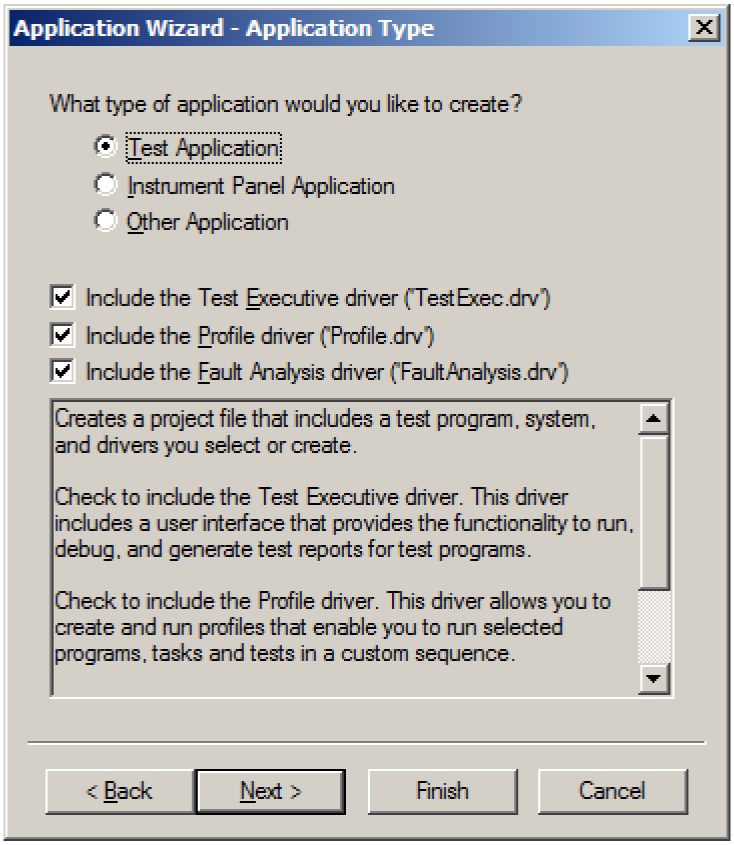
☑ Create New Workspace

Workspace file name : Example1

Workspace location : C:\Users\victorb\Desktop\Examp

The next screen will ask which type of application you want to create. Most of the exercises will require **Test Application** which creates a program, system and drivers with the intention of designing a test for a UUT.

8. You are also provided options to insert software drivers such as the Test Executive, Fault Analysis, and Profile drivers. Leave all of these checked and click **Next**.



What type of application would you like to create?

● Test Application
○ Instrument Panel Application
○ Other Application

☑ Include the Test Executive driver ("TestExec.drv")
☑ Include the Profile driver ("Profile.drv")
☑ Include the Fault Analysis driver ("FaultAnalysis.drv")

Creates a project file that includes a test program, system, and drivers you select or create.

Check to include the Test Executive driver. This driver includes a user interface that provides the functionality to run, debug, and generate test reports for test programs.

Check to include the Profile driver. This driver allows you to create and run profiles that enable you to run selected programs, tasks and tests in a custom sequence.

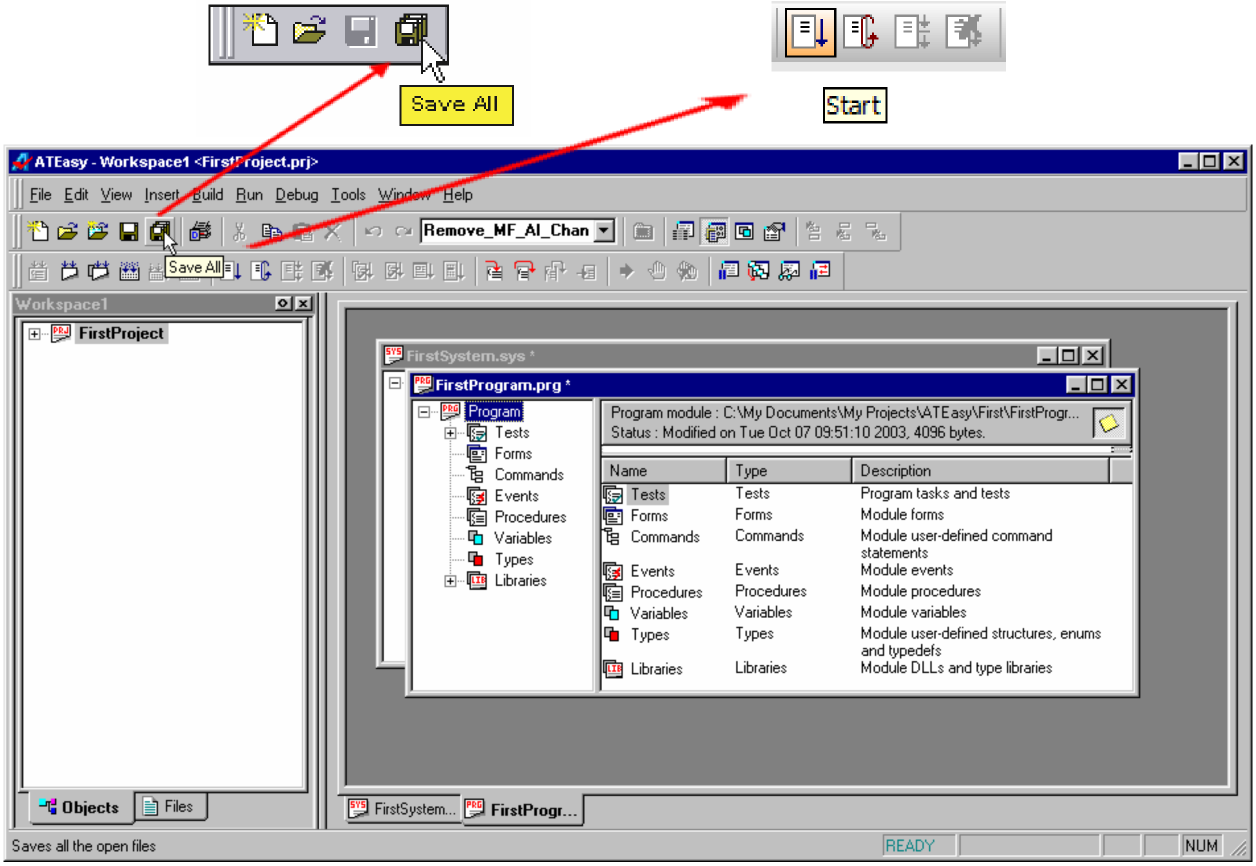
9. The next window, **Application Wizard – Program**, allows you to change the **Program** name and location, leave these unchanged and click **Next.**

10. **Application Wizard – System** allows you to change the name and location of the ATEasy System file, leave these unchanged and click **Next**.

11. The **Select Drivers** window lets you review the already included drivers and insert other drivers if desired. We have no drivers to add so click **Finish**.

12. A summary will be presented for your approval. Click **OK**.

13. The resulting framework and layout will be similar to that shown below. Click the **Save All** icon to save the recently create ATEasy application files.



Program module : C:\My Documents\My Projects\ATEasy\First\FirstProgr...
Status : Modified on Tue Oct 07 09:51:10 2003, 4096 bytes.

Name	Type	Description
Tests	Tests	Program tasks and tests
Forms	Forms	Module forms
Commands	Commands	Module user-defined command statements
Events	Events	Module events
Procedures	Procedures	Module procedures
Variables	Variables	Module variables
Types	Types	Module user-defined structures, enums and typedefs
Libraries	Libraries	Module DLLs and type libraries

Saves all the open files

Exercise Summary

This project is a complete test application; albeit one that does not do very much. You can click the Start button and a test application with Test Executive will launch. A single **Untitled Test** will reside within your program and if you click the Test Executive's **Start** button, the empty test will be run and added to the log. Since we have not programmed PASS/FAIL criteria for this test, it will not have a PASS / FAIL status.

Exercise 2 – Creating Modules Manually

Objectives

Create a test application manually by inserting each module individually.

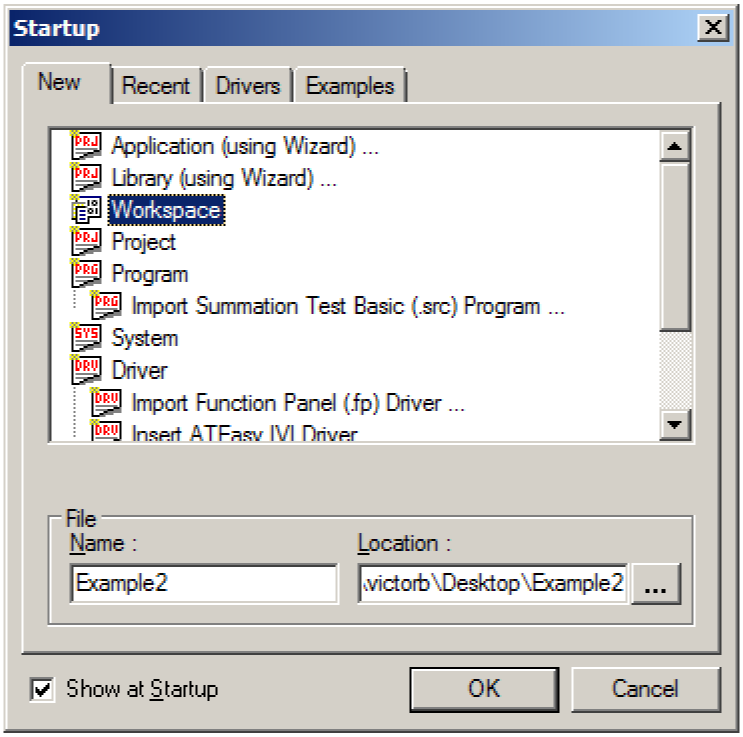
In this exercise, we will create a complete ATEasy test application module by module. The Application Wizard does an excellent job of creating a canned test application quickly, but a developer will often need to create non-standard test application, such as one that has multiple program modules in one project or multiple projects in a single workspace.

▼ Create an ATEasy Application Manually

1. Create a folder on the Desktop called **Example2**.

2. Depending on the workspace configuration, the startup window will appear automatically and show you the recently opened workspace. Click the **New** tab. If the startup window is not open, select **File | New** from the file menu.

3. Select **Workspace** from the dialog, enter **Example2** for the name and select the **Example2** folder that was created on the Desktop for the location. Press the **OK** button and an empty workspace will be created.

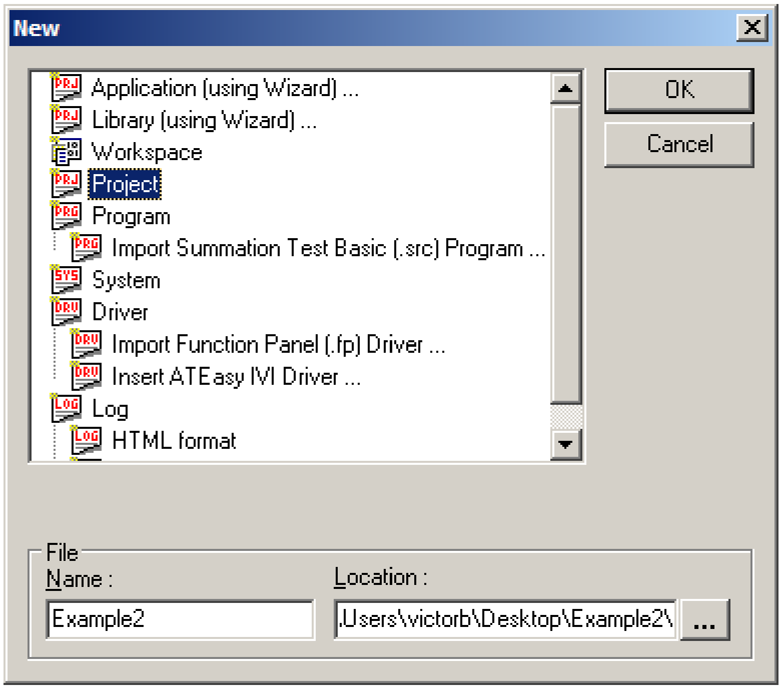

Startup
New
Recent
Drivers
Examples
Application (using Wizard) ...
Library (using Wizard) ...
Workspace
Project
Program
Import Summation Test Basic (.src) Program ...
System
Driver
Import Function Panel (.fp) Driver ...
Insert ATEasy IVI Driver
File
...
OK
Cancel

Name : Example2

Location : victorb\Desktop\Example2

☑ Show at Startup

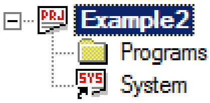
4. Select **File | New** from the file menu. Select **Project** from the dialog, enter **Example2** for the name and point the location to the folder that was created on the Desktop. Press the **OK** button and an Example2 project will be added to your workspace.


New
Application (using Wizard) ...
Library (using Wizard) ...
Workspace
Project
Program
Import Summation Test Basic (.src) Program ...
System
Driver
Import Function Panel (.fp) Driver ...
Insert ATEasy IVI Driver ...
Log
HTML format
OK
Cancel
File
...

Name : Example2

Location : .Users\victorb\Desktop\Example2\

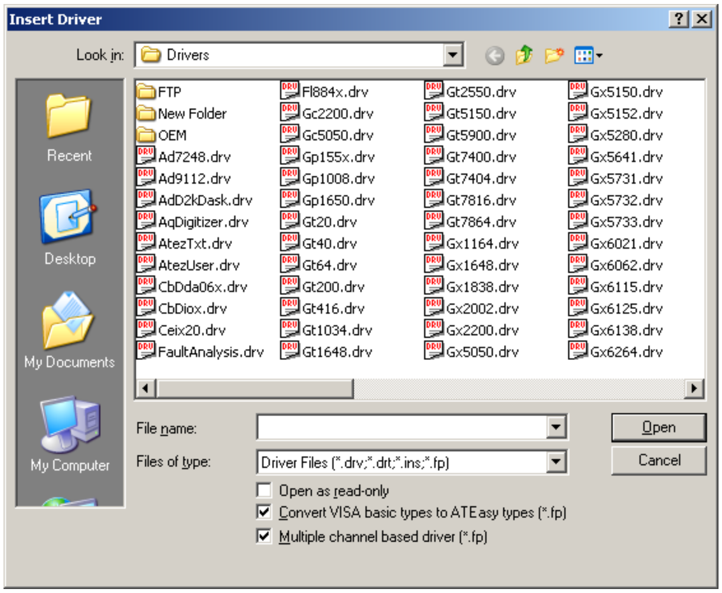
5. Expand the **Example2** project item within the workspace to reveal the System shortcut and the Programs folder.



6. Right-click the Programs folder and select **New Program**. Click **Save** after the Program has been added to the project. This will prompt you to select a file name and location. Save **Example2** to the Example2 desktop folder.

7. Right-click the System shortcut and select **New System**. Click **Save** after the System has been added. Save **Example2** to the Example2 desktop folder.

8. Expand the System shortcut to reveal the Drivers folder Right-click the **Drivers** folder and select **Insert Driver Below**. This allows the user to insert an existing driver into an ATEasy System. The Insert Driver dialog will appear.



Look in: Drivers

File name:

Files of type: Driver Files (*.drv;*.drt;*.ins;*.fp)

☐ Open as read-only

☑ Convert VISA basic types to ATEasy types (*.fp)

☑ Multiple channel based driver (*.fp)

9. The default location opened is the ATEasy Drivers folder (**`Program Files\ATEasy`**`[ (x86)]`**`\Drivers`**`).` This folder contains ATEasy drivers for all MTS products and some popular instruments from other manufacturers. Using the **Insert Driver** dialog, insert the following drivers:

   Profile.drv

   FaultAnalysis.drv

   TestExec.drv

Exercise Summary

This project, like the one created in Exercise 1, is a complete test application and it will behave in the same manner if you the **Start** button to launch the application. The techniques practiced here will be utilized more often in future applications where modification of existing projects is the intent rather that new development.

Exercise 3 – Writing Test Code

Objectives

Create Tests in the Program module.

Examine various test properties.

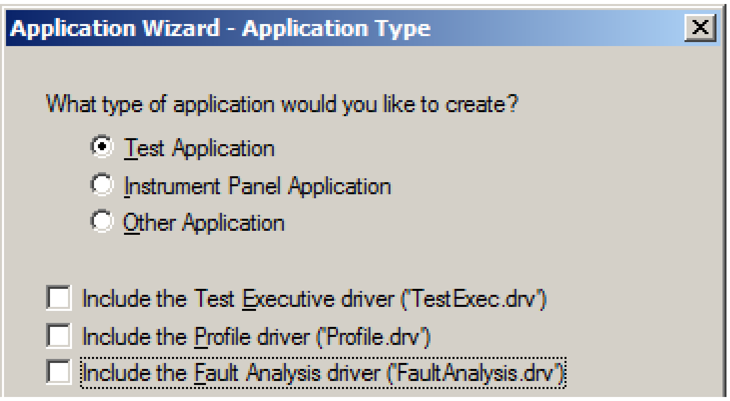
Write test code.

Use the TestResult variable.

We will create an ATEasy test application and insert the provided GtDemo ATEasy driver to control the GT98901 Demo Board. The demo board has two DACs and four ADCs which we will wire together to perform a voltage test. The specified voltage will be assigned to the demo board in the test code and the expected voltages will be assigned to the test via the test properties. After the test code is written and the test properties are set, we will examine how ATEasy uses this information to execute the test application and generate a log. For this exercise, the GT98901 Demo Board must be plugged into the controller prior to opening ATEasy. We will wire the contact of AOut1 to AIn1 and AOut2 to AIn2.

▼ Create the Task / Test Structure

1. Create a folder on the Desktop called **Example3**.
2. Use the Application Wizard to create a basic test application. The name of the files should be **Example3** and set the file location to the Example3 folder on the desktop. During the application wizard, you will be given an option of selecting whether or not to include software drivers. Uncheck all of the optional drivers for this exercise:



Application Wizard - Application Type

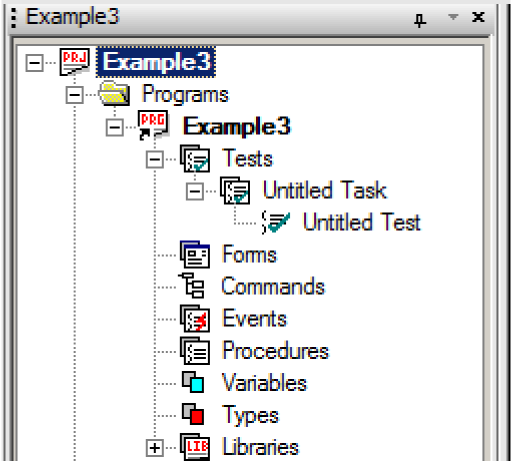
What type of application would you like to create?

● Test Application
○ Instrument Panel Application
○ Other Application

□ Include the Test Executive driver ("TestExec.drv")
□ Include the Profile driver ("Profile.drv")
□ Include the Fault Analysis driver ("FaultAnalysis.drv")

3. Click **Finish** to complete the application wizard.

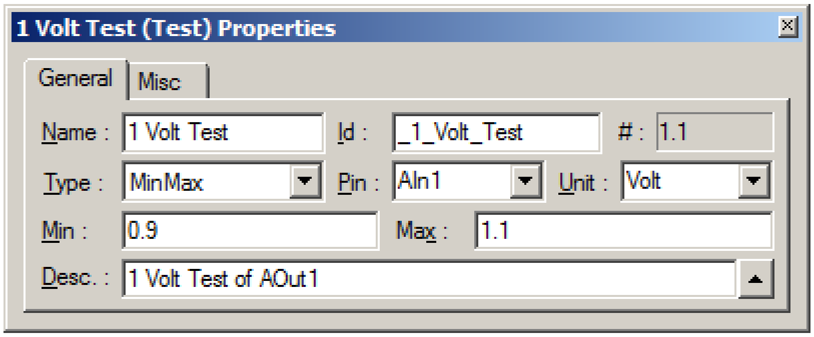
4. Expand the **Example3 program** module and then expand the **Tests** submodule.



5. Left-click the **Untitled Task** and press F2 to edit it. Rename this task to "**AO1 Voltage Task**".

6. Left-click the **Untitled Test** and press F2 to edit it. Rename this test to "**1 Volt Test**".

7. Right-click **1 Volt Test** and select **Properties** to open the properties editor. Set the properties to the following:



1 Volt Test (Test) Properties

Name : 1 Volt Test
Id : _1_Volt_Test
: 1.1
Type : MinMax
Pin : AIn1
Unit : Volt
Min : 0.9
Max : 1.1
Desc. : 1 Volt Test of AOut1

8. We need to create four more tests within **AO1 Voltage Task**, this can be achieved by copy-pasting the **1 Volt Test** and selecting to **Duplicate**. Alternately, you can right-click the **1 Volt Test** and select **Insert Test After** to create a new Untitled Test.

9. Give the next four tests (2-5) the following properties:

   Name **2 Volt Test**
   Type **Min/Max**
   Pin **AIn1**
   Unit **Volt** (you either type this or select from the combo box list)
   Min **1.8**
   Max **2.2**
   Desc **2 Volt Test of AOut1**

   Name **3 Volt Test**
   Type **Min/Max**
   Pin **AIn1**
   Unit **Volt** (you either type this or select from the combo box list)
   Min **2.7**
   Max **3.3**
   Desc **3 Volt Test of AOut1**

   Name **4 Volt Test**
   Type **Min/Max**
   Pin **AIn1**
   Unit **Volt** (you either type this or select from the combo box list)
   Min **3.6**
   Max **4.4**
   Desc **4 Volt Test of AOut1**

   Name **5 Volt Test**
   Type **Min/Max**
   Pin **AIn1**
   Unit **Volt** (you either type this or select from the combo box list)
   Min **4.5**
   Max **5.5**
   Desc **5 Volt Test of AOut1**

10. Create another Task and set of five Tests. You can do this by right-clicking the **AO1 Voltage Task** and selecting **Insert Task After**. This will create a new Untitled Task and Untitled Test. Alternately, you can copy and paste AO1 Voltage Task. The new task should be named **AO2 Voltage Task**. Create the next five tests as follows:

Name	**6 Volt Test**
Pin	**AIn2**
Unit	**Volt** (you either type this or select from the combo box list)
Type	**Tolerance**
Value	**6**
Min	**0.6**
Max	**0.6**
Desc	**6 Volt Test of AOut2**

Name	**7 Volt Test**
Pin	**AIn2**
Unit	**Volt** (you either type this or select from the combo box list)
Type	**Tolerance**
Value	**7**
Min	**0.7**
Max	**0.7**
Desc	**7 Volt Test of AOut2**

Name	**8 Volt Test**
Pin	**AIn2**
Unit	**Volt** (you either type this or select from the combo box list)
Type	**Tolerance**
Value	**8**
Min	**0.8**
Max	**0.8**
Desc	**8 Volt Test of AOut2**

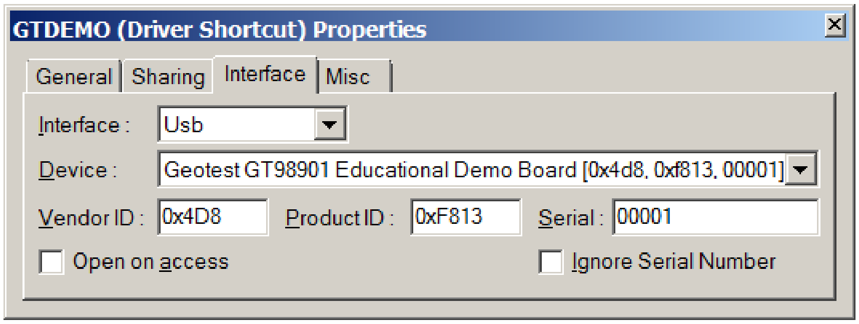
Name	**9 Volt Test**
Pin	**AIn2**
Unit	**Volt** (you either type this or select from the combo box list)
Type	**Tolerance**
Value	**9**
Min	**0.9**
Max	**0.9**
Desc	**9 Volt Test of AOut2**

Name	**10 Volt Test**
Pin	**AIn2**
Unit	**Volt** (you either type this or select from the combo box list)
Type	**Tolerance**
Value	**10**
Min	**1**
Max	**1**
Desc	**10 Volt Test of AOut2**

▼ Add the GtDemo driver and configure

11. Add the **GTDEMO** driver by right-clicking the Drivers folder in the System module and selecting **Insert Driver Below**. Select the driver called **GtDemo.drv** or **GT98901.drv**. The driver is located in **C:\ProgramFiles\Marvin Test Solutions\GtDemo**
12. Right-click on the **GTDEMO** driver and select **Properties** to open the GTDEMO (Driver Shortcut) Properties window.
13. In the Properties editor, click on the **Interface** tab.
14. Change the Interface type from **None** to **Usb**.

The GT98901 may have been registered under any of several different Device Driver names within the Windows Device Manager. Click the device dropbox and select the item labeled **USBTMC** or **Geotest GT98901 Educational Demo Board** or **USB Test and Measurement**.



GTDEMO (Driver Shortcut) Properties

Interface : Usb

Device : Geotest GT98901 Educational Demo Board [0x4d8, 0xf813, 00001]

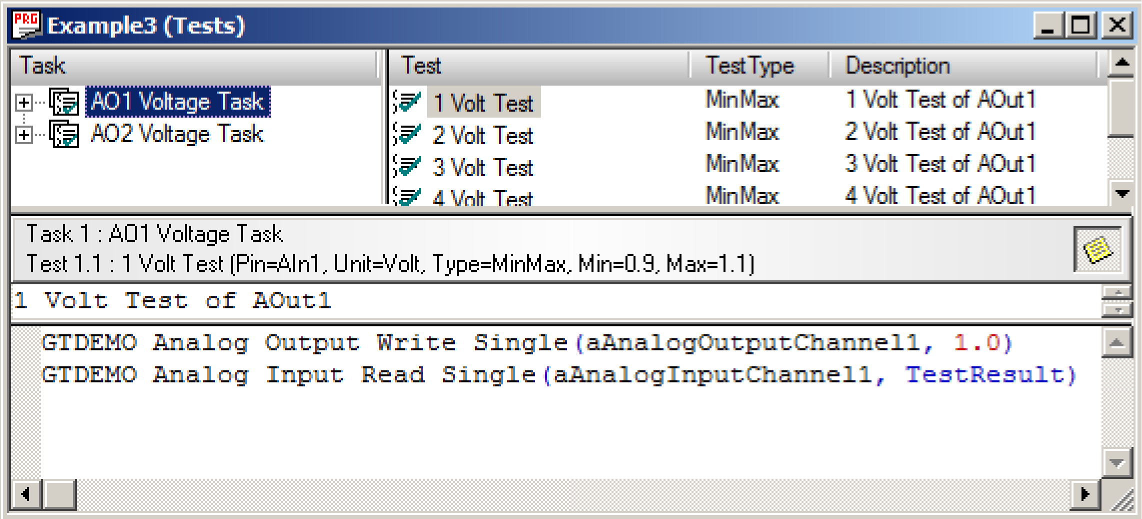
Vendor ID : 0x4D8 Product ID : 0xF813 Serial : 00001

□ Open on access □ Ignore Serial Number

▼ Write the test code

15. From the workspace navigator, **double-click** the **1 Volt Test**. This will open the code editor for the 1 Volt Test and move the cursor into position to begin typing.
16. Use the Insert menu to insert a command to control the GT98901. Select **Insert | Driver Command | GTDEMO | Analog | Output | Write | Single** from the menu bar. This inserts the command code into the code editor. If the command requires parameters, which it does, a parameter suggestion box will open.
17. The first parameter that you are adding is the analog output channel that you want to update. Highlight **aAnalogOutputChannel1** and hit Enter to insert. Press the **comma** key (**,**) to view the next parameter.
18. The second parameter to add is the target voltage. Type **1.0** and a closed parenthesis to complete this command.
19. Use the Insert menu to insert a second command. Select **Insert | Driver Command | GTDEMO | Analog | Input | Read | Single**.
20. The first parameter that you are adding is the analog input channel that you want to make a measurement on. Highlight **aAnalogInputChannel1** and hit enter to insert. Press the **comma** key (**,**) to view the next parameter.
21. The second parameter will be the variable that will be used to save the measurement that is being made from the specified input channel. Type **TestResult** and a closed parenthesis to complete the command.

22. This concludes the creation of the first test, which should appear similar to the following:



Test	Test Type	Description
1 Volt Test	MinMax	1 Volt Test of AOut1
2 Volt Test	MinMax	2 Volt Test of AOut1
3 Volt Test	MinMax	3 Volt Test of AOut1
4 Volt Test	MinMax	4 Volt Test of AOut1

Task 1 : AO1 Voltage Task
Test 1.1 : 1 Volt Test (Pin=AIn1, Unit=Volt, Type=MinMax, Min=0.9, Max=1.1)

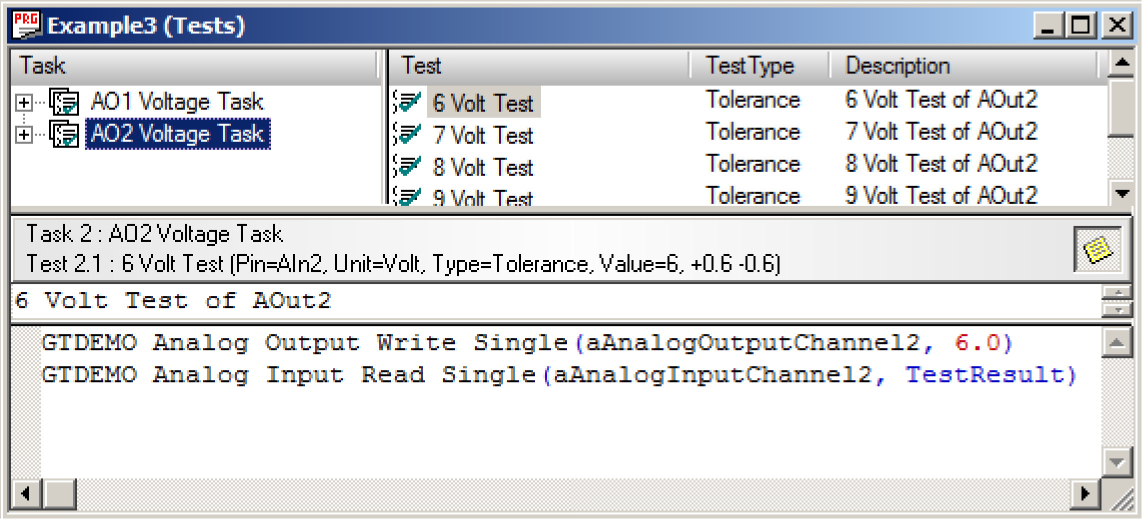
```
1 Volt Test of AOut1
```

```
GTDEMO Analog Output Write Single(aAnalogOutputChannel1, 1.0)
GTDEMO Analog Input Read Single(aAnalogInputChannel1, TestResult)
```

Since the intention is to exercise the full range of voltages that AOut1 is capable of generating, we will continue by populating the 2 Volt Test and so on.

▼ Update Tests 6-10

23. Follow the procedure in steps 16 through 22 to set the voltage of the analog outputs and read back measurements with the analog inputs of the other nine tests. To edit test code for a particular test, **double-click** the test in the Tests sub-module of the program module. Remember that in AO2 Voltage Task we will be generating voltages from aAnalogOutputChannel2 (not AO1) and making measurements on aAnalogInputChannel2 (not AI1), as below:



Test	Test Type	Description
6 Volt Test	Tolerance	6 Volt Test of AOut2
7 Volt Test	Tolerance	7 Volt Test of AOut2
8 Volt Test	Tolerance	8 Volt Test of AOut2
9 Volt Test	Tolerance	9 Volt Test of AOut2

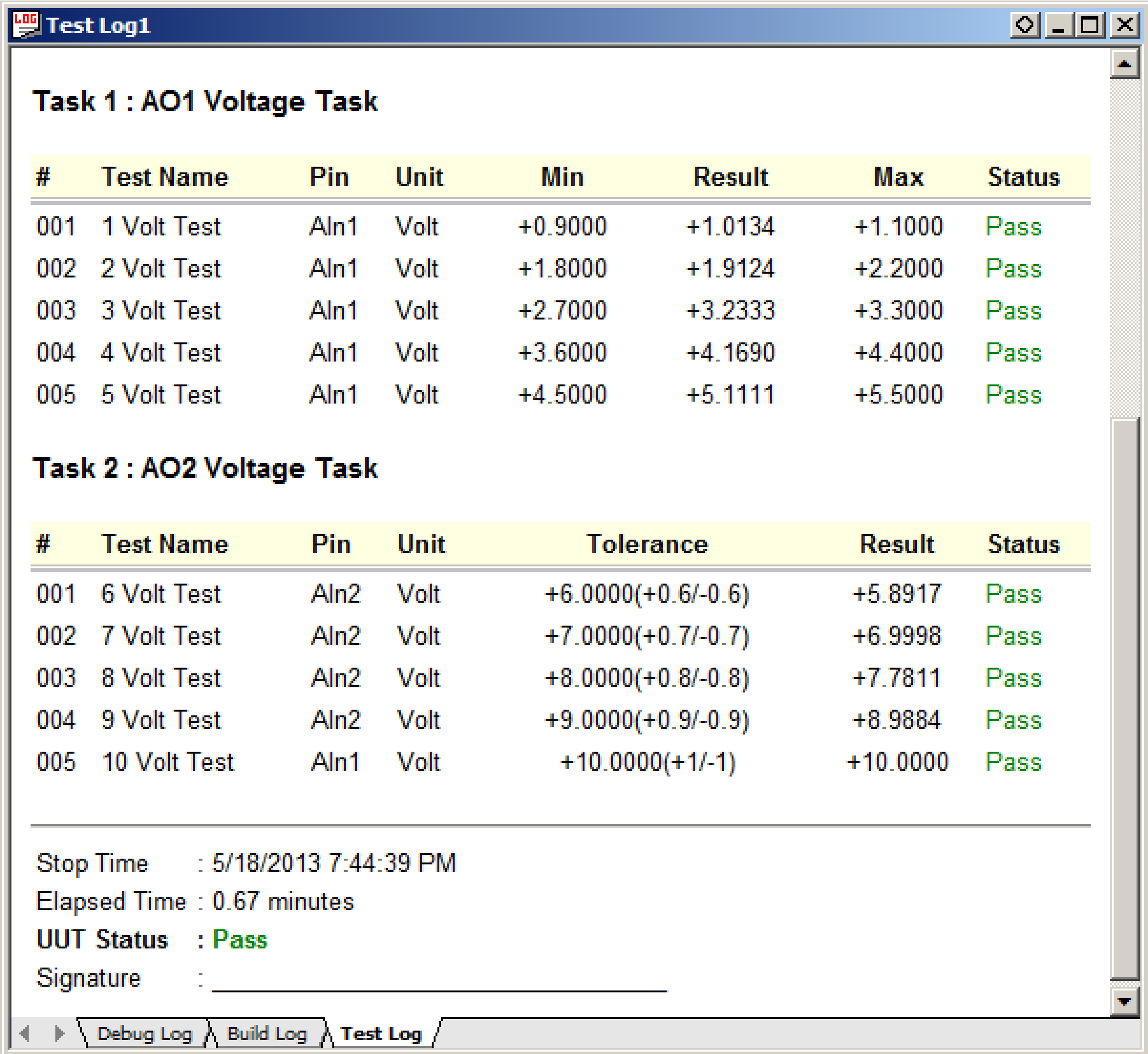
Task 2 : AO2 Voltage Task
Test 2.1 : 6 Volt Test (Pin=AIn2, Unit=Volt, Type=Tolerance, Value=6, +0.6 -0.6)

```
6 Volt Test of AOut2
```

```
GTDEMO Analog Output Write Single(aAnalogOutputChannel2, 6.0)
GTDEMO Analog Input Read Single(aAnalogInputChannel2, TestResult)
```

▼ Running the tests

24. Now would be the best time to physically wire the connection between the DAC and ADC ports. AOut1 should be wired to AIn1 and AOut2 should be wired to AIn2.
25. Once all the tests have been written and the connections made, press the **Start** button to run the voltage tests. Since we have excluded the test executive from this project, the output will appear in the Log window within ATEasy. Run it a few times and review the results.



Task 1 : AO1 Voltage Task

#	Test Name	Pin	Unit	Min	Result	Max	Status
001	1 Volt Test	AIn1	Volt	+0.9000	+1.0134	+1.1000	Pass
002	2 Volt Test	AIn1	Volt	+1.8000	+1.9124	+2.2000	Pass
003	3 Volt Test	AIn1	Volt	+2.7000	+3.2333	+3.3000	Pass
004	4 Volt Test	AIn1	Volt	+3.6000	+4.1690	+4.4000	Pass
005	5 Volt Test	AIn1	Volt	+4.5000	+5.1111	+5.5000	Pass

Task 2 : AO2 Voltage Task

#	Test Name	Pin	Unit	Tolerance	Result	Status
001	6 Volt Test	AIn2	Volt	+6.0000(+0.6/-0.6)	+5.8917	Pass
002	7 Volt Test	AIn2	Volt	+7.0000(+0.7/-0.7)	+6.9998	Pass
003	8 Volt Test	AIn2	Volt	+8.0000(+0.8/-0.8)	+7.7811	Pass
004	9 Volt Test	AIn2	Volt	+9.0000(+0.9/-0.9)	+8.9884	Pass
005	10 Volt Test	AIn1	Volt	+10.0000(+1/-1)	+10.0000	Pass

Stop Time : 5/18/2013 7:44:39 PM
Elapsed Time : 0.67 minutes
UUT Status : Pass
Signature : ______________________________

Summary

This exercise introduced the tasks and test operations that will take place within the Tests submodule of the program module: adding new tests, configuring those tests, and writing test code. We have also seen the basic ATEasy test log used for the first time, without the additional utility granted by including the Test Executive driver.

Exercise 4 – Profile and Test Executive

Objectives

Add the Profile and Test Executive to the system.

Define custom profiles using the Profile driver.

Review Misc properties of Driver Shortcuts.

We will pick up where we left off with Exercise 3 by including the Test Executive and Profile drivers. We will briefly explore the functionality and features that the Test Executive adds to the project and then we will use the Profile driver to create custom subset of tests from the complete test set. Finally, we will look at the Misc properties of a Driver Shortcut to see how a driver's functionality can be altered through configuration.

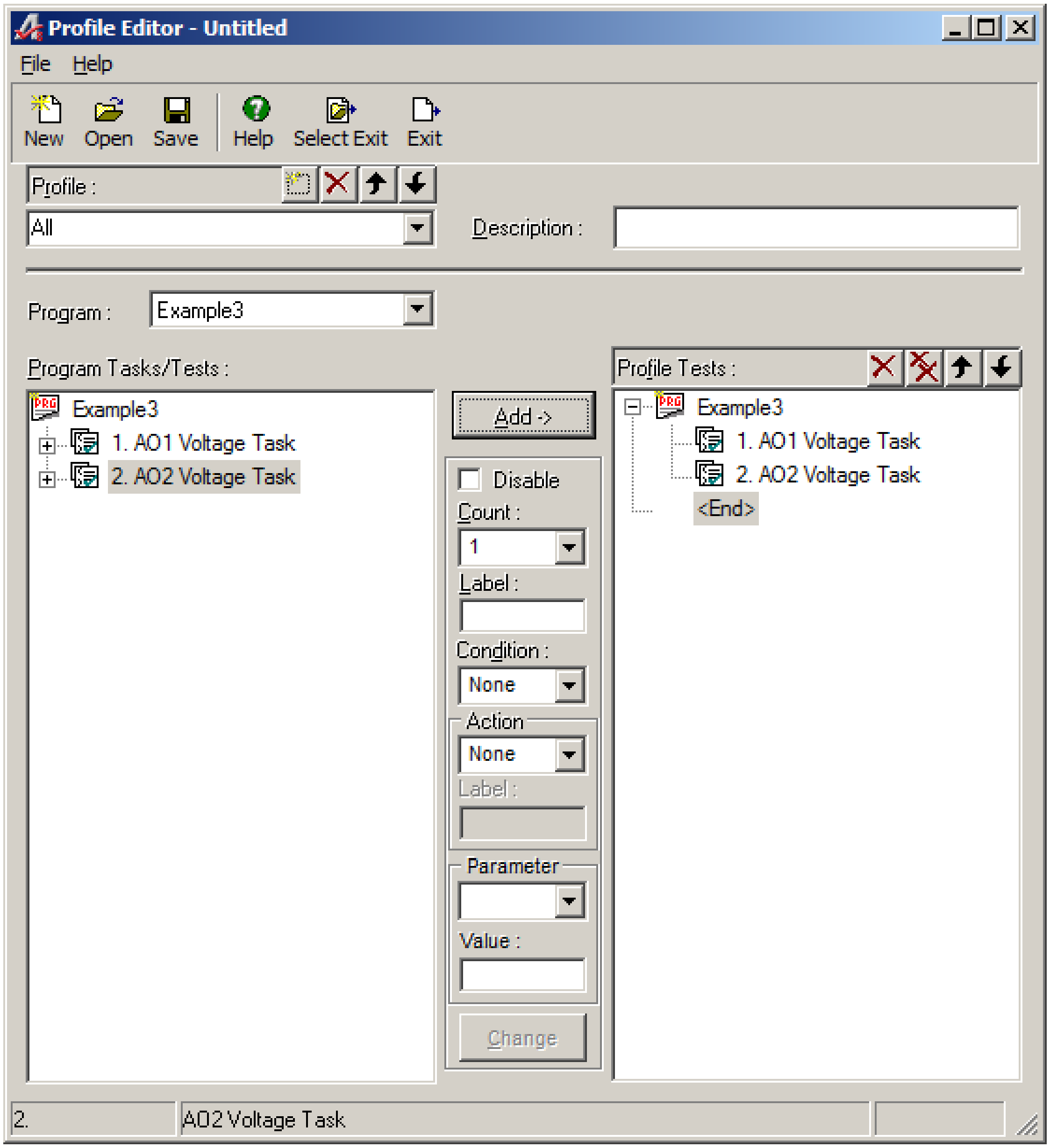
▼ Add the Profile and Test Executive drivers

1. Open the existing **Example3** project.
2. Expand the System shortcut to reveal the Drivers folder Right-click the **Drivers** folder and select **Insert Driver Below**. This allows the user to insert an existing driver into an ATEasy System. Using the **Insert Driver** dialog, insert the following drivers:
    - Profile.drv
    - TestExec.drv

▼ Create the All profile

3. Hit F5 to Start the **TestExecutive**. Note the difference in how the log of ATEasy and the TestExecutive look.
4. From the menu bar of the **Test Executive**, select **Program | Edit Profile** to launch the Profile Editor, a graphical interface used to create/edit profiles.
5. After opening the Profile Editor, click the **Save** button to create a disk copy of your new Profile. When prompted, save the file as **Example4.prf** in the Desktop **Example3** folder.
6. A default Profile called **Profile 1** is created for new Profiles by default. Rename Profile 1 to **All**. This will be the name of the first profile we are creating, one which will run all of the tests in the test set. Add the following to the **Description** field: **Runs all of the tests.**
7. From the list of Program Tasks/Tests, click on the first Task: **AO1 Voltage Task**. Then click the **Add ->** button. This will add the task to the current profile's test list.

8. Click on the second Task: **AO2 Voltage Task** and click **Add ->**. This will add the second task to the All profile's test list. Task 1 and Task 2 together comprise all of the tests in the **Example3** program, so this profile is truly executing all of the tests.



Profile : All

Description :

Program : Example3

Program Tasks/Tests :

- Example3
  - 1. AO1 Voltage Task
  - 2. AO2 Voltage Task

Profile Tests :

- Example3
  - 1. AO1 Voltage Task
  - 2. AO2 Voltage Task
- <End>

□ Disable

Count : 1

Label :

Condition : None

Action: None

Label :

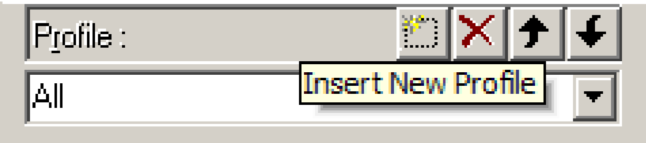
Parameter:

Value :

2. AO2 Voltage Task

▼ Create the Some profile

9. To start creating a new profile, click the **Insert New Profile** button. It is located near the profile name and looks like a yellow asterisk and a dotted square.



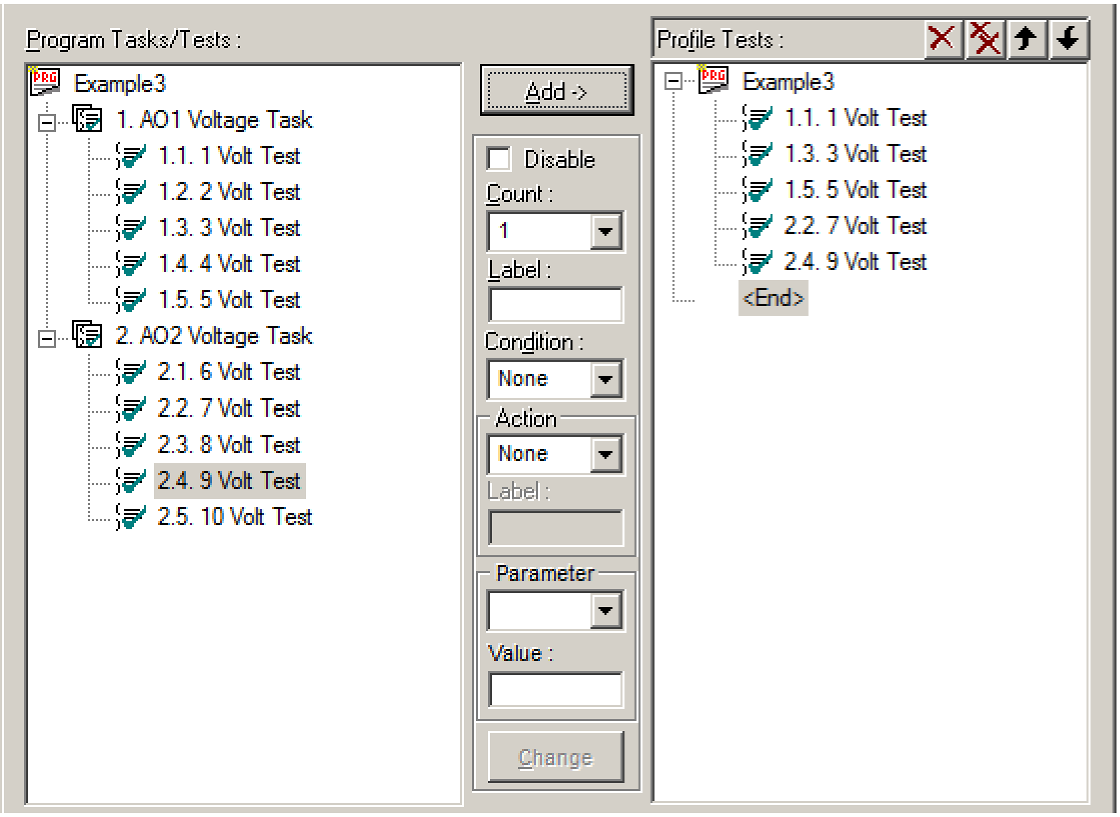
Profile :

All

10. Rename the new **Profile 2** to **Some**. This profile will run only half of the complete test set. Add the following text to the **Description** field: **Runs some of the tests.**

11. Expand the tasks in the Program Tasks/Tests tree.

12. Add every other test to the current profile starting with **Test 1.1**.



Program Tasks/Tests :

- Example3
  - 1. AO1 Voltage Task
    - 1.1. 1 Volt Test
    - 1.2. 2 Volt Test
    - 1.3. 3 Volt Test
    - 1.4. 4 Volt Test
    - 1.5. 5 Volt Test
  - 2. AO2 Voltage Task
    - 2.1. 6 Volt Test
    - 2.2. 7 Volt Test
    - 2.3. 8 Volt Test
    - 2.4. 9 Volt Test
    - 2.5. 10 Volt Test

☐ Disable

Count : 1

Label :

Condition : None

Action : None

Label :

Parameter :

Value :

Profile Tests :

- Example3
  - 1.1. 1 Volt Test
  - 1.3. 3 Volt Test
  - 1.5. 5 Volt Test
  - 2.2. 7 Volt Test
  - 2.4. 9 Volt Test
  - <End>

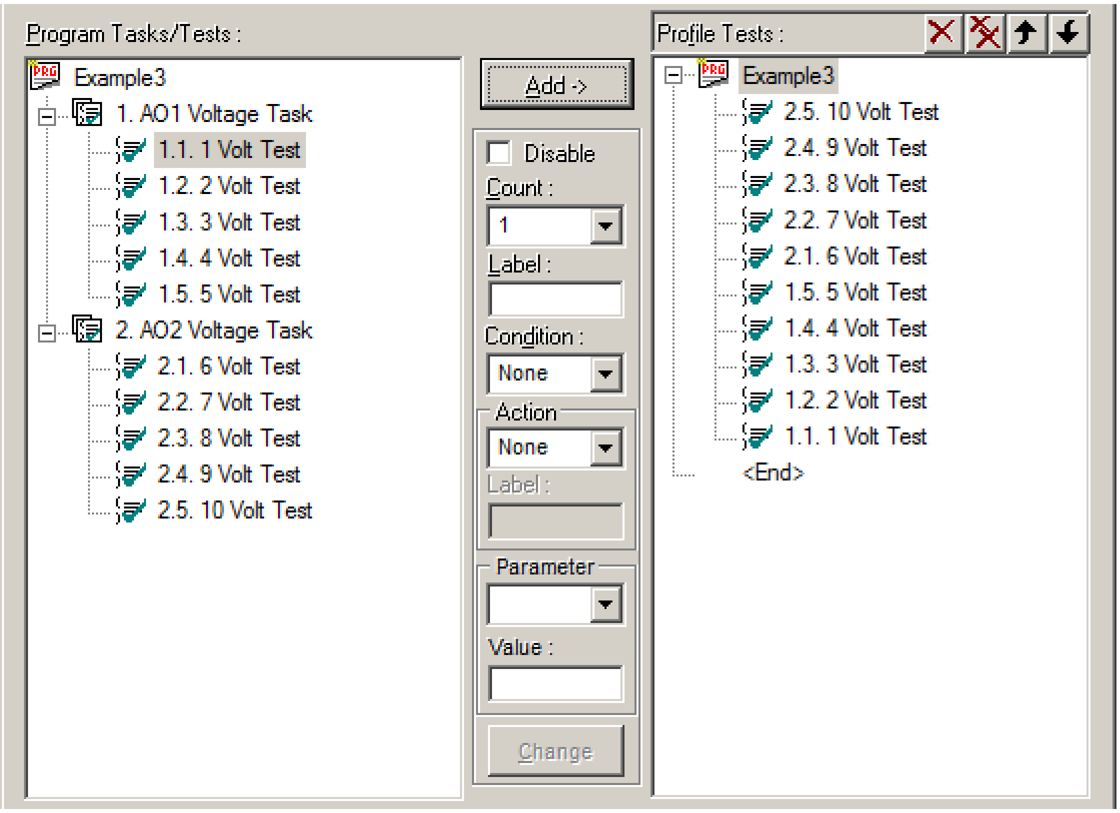
▼ Create the Order profile

13. Click the **Insert New Profile** button.

14. Rename the new **Profile 3** to **Order**. This profile will run all of the tests, but it will run them out of order. Add the following text to the Description field: Runs the tests out of order.

15. Expand the tasks in the Program Tasks/Tests tree.

16. Add every test starting with test 2.5 and working back to test 1.1.



Program Tasks/Tests :

- Example3
  - 1. AO1 Voltage Task
    - 1.1. 1 Volt Test
    - 1.2. 2 Volt Test
    - 1.3. 3 Volt Test
    - 1.4. 4 Volt Test
    - 1.5. 5 Volt Test
  - 2. AO2 Voltage Task
    - 2.1. 6 Volt Test
    - 2.2. 7 Volt Test
    - 2.3. 8 Volt Test
    - 2.4. 9 Volt Test
    - 2.5. 10 Volt Test

□ Disable

Count : 1

Label :

Condition : None

Action: None

Label :

Parameter

Value :

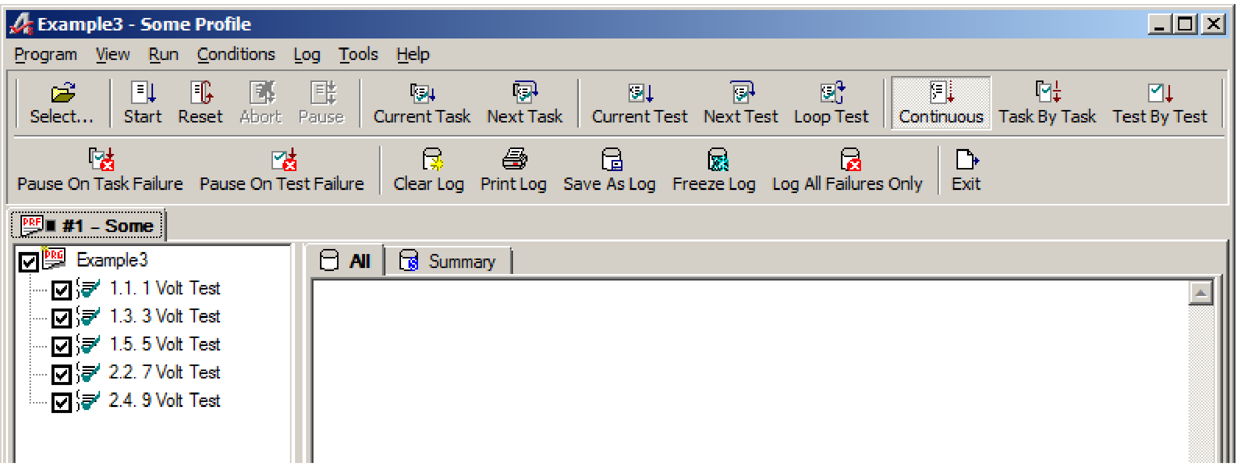
Profile Tests :

- Example3
  - 2.5. 10 Volt Test
  - 2.4. 9 Volt Test
  - 2.3. 8 Volt Test
  - 2.2. 7 Volt Test
  - 2.1. 6 Volt Test
  - 1.5. 5 Volt Test
  - 1.4. 4 Volt Test
  - 1.3. 3 Volt Test
  - 1.2. 2 Volt Test
  - 1.1. 1 Volt Test
  - <End>

This concludes the profile creation. Click **Save** and then click **Exit**.

▼ Running the newly created profiles

17. Once all the profiles have been created and the profile editor is exited, you can load a profile by accessing **Program | Select Profile** from the menu bar. Use the dropdown list to select which profile to load: **All**, **Some**, or **Order**. Once the desired profile is chosen, press **Select** to load the selected profile to the Test Executive. Then run the profile and observe the results.


Example3 - Some Profile
Program View Run Conditions Log Tools Help
Select... Start Reset Abort Pause Current Task Next Task Current Test Next Test Loop Test Continuous Task By Task Test By Test
Pause On Task Failure Pause On Test Failure Clear Log Print Log Save As Log Freeze Log Log All Failures Only Exit
#1 – Some
Example3
1.1. 1 Volt Test
1.3. 3 Volt Test
1.5. 5 Volt Test
2.2. 7 Volt Test
2.4. 9 Volt Test
All
Summary

When a profile is loaded, it will be noted in the Test Executive caption as well as the UUT tab. The Tests list will show the profile's subset of tests.

When TestExec.drv is linked into the Drivers folder of the active project, it is automatically enabled so that subsequent builds will be able to use its features. If desired, the Test Executive software driver can be left in the system, but programmatically disabled. This is done via the driver parameters.

▼ Disabling the Test Executive

18. To edit the TestExec driver parameter, you will need to **close the running application**.

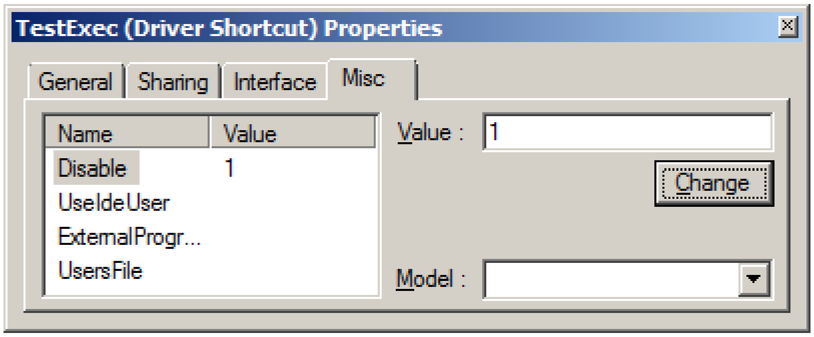
19. Open the **Properties Editor** for the **TestExec** driver.

20. Click on the **Misc** tab to view the Driver Shortcut parameters. The TestExec driver parameters function as follows:

    - Disable - When set to 1, the test executive is disabled.
    - UseIdeUser - When set to 1, the test executive will use the currently logged in ATEasy user to log into the test executive. This only functions if multiple user support is enabled.
    - ExternalProgramsFile Path an INI file containing Project information. Enables External Program Executables Support when set.
    - UsersFile - Path to a USR file. If this parameter is set, then multiple user support will be enabled.

Many of these features are beyond the scope of the driver, but should demonstrate how Driver Parameters are useful in allowing the user to configure their own usage with an ATEasy driver.

21. Set the Disable parameter to 1 and start the project. It should output to the IDE log window instead of opening a Test Executive.



Name	Value
Disable	1
UseIdeUser	
ExternalProgr...	
UsersFile	

Value : 1

Model :

Summary

This exercise demonstrated the features and usage of both the Test Executive and the Profile drivers. It introduced profile files (PRF) which can be loaded into applications at run-time. Finally, we looked at the Driver parameters and their usefulness for configuring drivers.

Exercise 5 – Flow-Control Statements

Objectives

Use ATEasy flow control statements

Check for errors using CheckIt!

Execute code using DoIt!

Toggle some of the subsystems of the GT98901 driver

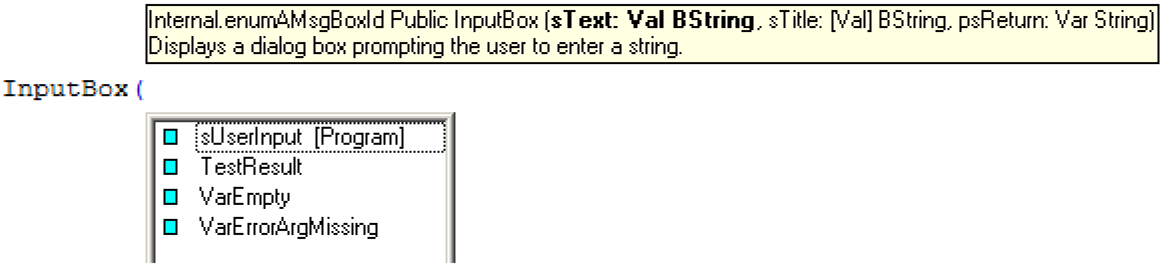
In this exercise, we will create a new ATEasy application and use flow-control statements to toggle the GT98901's LEDs or buzzers based on user input. This should help to familiarize the user with the ATEasy syntax and introduce the ATEasy version of common programming language elements.

1. Create a new project using the application wizard.
2. Create a new folder on the Desktop called **Example5**.
3. Use the Application Wizard to create a new test application. Name it **Example5** save it to the **Example5** desktop directory. When prompted, uncheck all of the software driver such as Profile.drv, TestExec.drv, and FaultAnalysis.drv.
4. Add an instance of the **GTDEMO** driver to the project.
5. Configure the Interface properties of the GTDEMO board to use the appropriate interface: **USBTMC** or **Geotest GT98901 Educational Demo Board** or **USB Test and Measurement**.

▼ Prompt the user for input

6. Rename the Untitled Task to **Toggling Task**.
7. Rename the Untitled Test to **Prompt User for Mode**.
8. **Right-click** the Variables sub-module of the Example5 Program module and select **Insert Variable Below**. A dummy Variable1 will be created. Open the properties editor for this variable and rename Variable1 to **sUserInput** and change the data type to **String**. This variable will be used to hold user input.
9. **Right-click** the sUserInput variable and select **Insert Variable After**. Rename this variable to **iIndex**. Leave the data type as long. This variable will be used to keep track of the for-loops.
10. **Double-click** the test Prompt User for Mode in the Workspace Navigator to open the code editor for this test.

11. We will be using the **InputBox()** procedure to prompt the user to select the demo board component that will be toggled. Type **InputBox(** into the code editor. Once you type the open parenthesis, the parameter suggestion/information windows will open and provide the expected parameter data type, description and suitable variables to use as the parameter.



Internal.enumAMsgBoxId Public InputBox (**sText: Val BString**, sTitle: [Val] BString, psReturn: Var String)
Displays a dialog box prompting the user to enter a string.

```
InputBox(
```

InputBox() displays a dialog box that includes a message, an input field, and an OK and Cancel button. The three parameters of the **InputBox** procedure are:

sText	String that describes the data we are requesting from the user.
sTitle	String that is displayed in the caption of the dialog box.
psReturn	Returns the contents of the text box.

12. Type the following user prompt for the sText parameter, including quotes: **"Enter 1 for buzzer,\r2 for the orange LED,\r3 for the green LED"**. The \r in the string is an escape character for carriage return and it is being used to format the prompt. After finishing the string literal, type comma (**,**) to move to the next parameter.

13. Type the following caption as the second parameter: **"Please select a mode"**. After typing this, type a comma '**,**' to move to the last parameter.

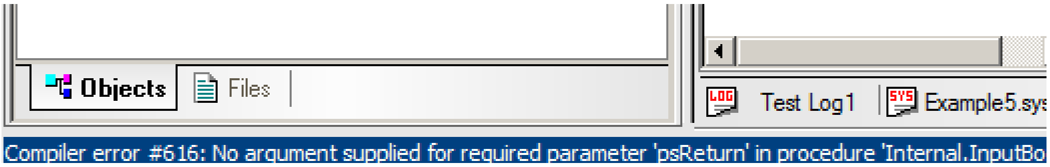
14. The final parameter will be a variable that will be assigned the contents of the text field after the user clicks the OK button and dismisses the dialog box. This is the final parameter, type a close parenthesis '**)**' to finish the procedure call.

CheckIt! is a utility within the ATEasy environment that is used to check the syntax of the selected code. It is much faster to CheckIt! to find an error than to wait for errors to be found during compilation. When clicked, **CheckIt!** will analyze the currently active code editor or the highlighted code.

▼ Using CheckIt! to check for errors

15. **Double-click** the **Prompt User for Mode** test if it is not currently in focus. Use the menu bar to activate **Build | CheckIt!**

If an error is found, a caret will appear pointing to the offending line of code and the status bar at the bottom of the screen will display a descriptive error message.



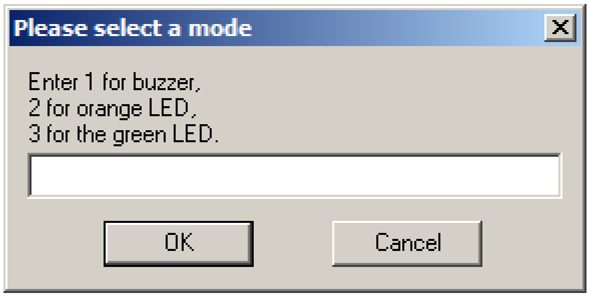
Compiler error #616: No argument supplied for required parameter 'psReturn' in procedure 'Internal.InputBo

16. The error message indicates that our procedure call indicates a missing parameter. Go back to your InputBox procedure call and type **sUserInput** as your last parameter.

DoIt! is a utility that is used to execute small snippets of code to verify its operation without the necessity of compiling and running through the application until the code is encountered. Like CheckIt!, it will execute the currently active code editor or the highlighted code.

▼ Using DoIt! to execute code

17. **Double-click** the **Prompt User for Mode** test if it is not currently in focus. Use the menu bar to activate **Debug | DoIt!** or use the keyboard shortcut **CTRL+D**. This will execute the InputBox procedure and present the user with a prompt.



Please select a mode

Enter 1 for buzzer,
2 for orange LED,
3 for the green LED.

▼ Create the select statement

18. Create another test under Prompt User for Mode. Name this new test **Toggle Selected Device**.

19. **Left-click** in the code editor within Toggle Selected Device so that the cursor appears within. The location of the cursor is the location that inserted code will be placed. From the menu bar, select **Insert | Flow-Control Statement | Select – Case – EndSelect**. A prototype for the Select statement should be inserted into the code editor below.

20. Next to the inserted select is a set of parentheses, which need to be filled with the expression that we are evaluating. The expression we are evaluating is the user input from the input box, so put **sUserInput** inside the parentheses.

21. The four cases we are expecting are “1”, “2”, or “3”. Next to the **case** keywords, type in these string literals with the quotes included.

22. There is a **case else** statement left. The code under case else will be executed if any input other than “1”, “2” or “3” is entered. This will including the cases when the user entered nothing into the input box or the case where the user clicked the cancel button. We will notify the user in this case. After the case else statement, hit **Enter** and type the following on the next line:

```
MsgBox("No valid mode selected")
```

23. This will pop up a dialog box with a message for the user and an OK button.

▼ Toggle the Buzzer

24. On the line after **case "1"**, we will add the code to toggle the demo board's buzzer ON and OFF three times, finishing with OFF. Position the cursor on the line below case "1". From the menu bar, select **Insert | Flow-Control Statement | For – Next | For – to – Next**. This will insert the prototype for an incrementing for-loop.

25. The first set of parentheses needs to be filled with the index variable and the starting valid. Type **iIndex=0** within these parentheses.

26. The second set of parentheses needs to be filled with the final value of the index variable. Type **6** within these parentheses. The for-loop will execute seven times.

27. Place the following code inside the for-loop to toggle the buzzer:

    ```
    GTDEMO Buzzer Set State(iIndex mod 2)
    ```

    This will turn the buzzer off when iIndex modulus 2 is zero and on when iIndex modulus 2 is one.

28. After turning the buzzer on or off, add the line:

    ```
    Sleep(100)
    ```

    This line puts the main thread of the application to sleep for 100 milliseconds, creating a 100 millisecond pause before continuing program execution. While this thread is sleeping, other threads and Windows processes will be able to execute.

▼ Toggle the LEDs

29. On the line after **case "2"**, we will add the code to toggle the demo board's orange LED ON and OFF. Start by adding a for-loop that spans **iIndex = 0 to 6**.

30. We will use the same modulus technique to toggle the LEDs. This time an **IF** statement will be used instead of putting the modulus as a parameter of the **GTDEMO** command. After opening the for-loop, use the menu bar and select **Insert | Flow-Control Statement | If-Then-Else-EndIf**.

31. After the If statement structure has been inserted, erase the **elseif** line as it will not be necessary for the purposes of this exercise.

32. Within the parentheses of **if ( ) then**, put the index modulus evaluation, **iIndex mod 2=1**.

33. On the next line (executed if the evaluation is true), type the following:

    ```
    GTDEMO Led Set State On(aLedOrange)
    ```

    As you might expect, this command turns on the orange LED.

34. After the **else** statement, type the following code to turn off the orange LED.

    ```
    GTDEMO Led Set State Off(aLedOrange)
    ```

35. The final step necessary is to add the delay that will allow the code to pause long enough to allow the viewer to see the LED toggling. Add a **Sleep(100)** before the **next** statement that concludes the for-loop.

36. The green LED code will follow the same process as the orange LED. Follow steps 26 through 32 with case "3" making the necessary modifications to the GTDEMO commands.

Reviewing the code

At this point, **CheckIt!** can be used to check the syntax of the Toggle Selected Device test. If there are any compiler errors found, a caret will point to the offending line and the error description will be placed in the status bar. Use this information to try to discover and resolve the issue.

Working code for this test can be viewed below. Your code may differ and still accomplish the goal of this exercise:

```
select (sUserInput)
case "1"
   for (iIndex=0) to (6) do
         GTDEMO Buzzer Set State(iIndex mod 2)
         Sleep(100)
   next
case "2"
   for (iIndex=0) to (6) do
         if (iIndex mod 2=1)
               GTDEMO Led Set State On(aLedOrange)
         else
               GTDEMO Led Set State Off(aLedOrange)
         endif
         Sleep(100)
   next
case "3"
   for (iIndex=0) to (6) do
         if (iIndex mod 2=1)
               GTDEMO Led Set State On(aLedGreen)
         else
               GTDEMO Led Set State Off(aLedGreen)
         endif
         Sleep(100)
   next
case else
   MsgBox("No valid mode selected")
endselect
```

Summary

This exercise demonstrated the ATEasy syntax of several flow control statements and how they can be used to simplify code creation. It also introduced the CheckIt! and DoIt! code testing and code execution utilities.

Exercise 6 – Module Events

Objectives:

Create event procedures

Add print statements to mark the location of event entry points

Write code to perform recurrent analysis of the PASS/FAIL status of tests

Use the ATEasy Help books

In this exercise, we will first add **Print** statements to event procedures in the Program and System modules. This will allows us to see the order of execution of program and system events. After we have located our entry points, we will use ATEasy branching statements alter the way our code executes. Specifically, we will jump out of a task if a test failure is encountered.

▼ Create a new project using the application wizard.

1. Create a new folder on the Desktop called **Example6**.
2. Use the Application Wizard to create a new test application. Name it **Example6** save it to the **Example6** desktop directory. When prompted, uncheck all of the software driver such as Profile.drv, TestExec.drv, and FaultAnalysis.drv.

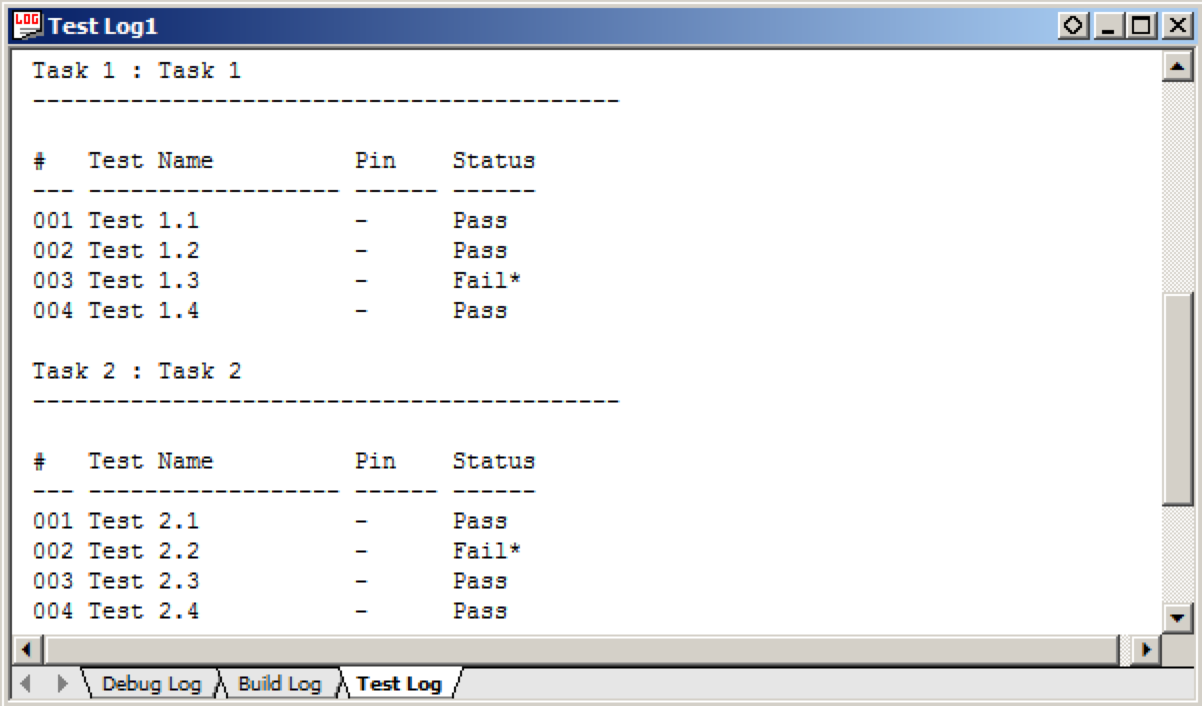
▼ Create the Task/Test structure

3. Create 2 tasks. Each task should have 4 tests each. Rename the task and test so they follow this structure:

    1. Task 1
       1.1 Test 1.1
       1.2 Test 1.2
       1.3 Test 1.3
       1.4 Test 1.4
    2. Task 2
       2.1 Test 2.1
       2.2 Test 2.2
       2.3 Test 2.3
       2.4 Test 2.4

4. Change the properties of all the tests so that their test type is **Other**.
5. In the code for each of the test, explicitly set the test status to PASS with the following code: **TestStatus=PASS**.
6. Go back and change the test status to FAIL for one test from each task.

7. Press the **Start** button to run the program. Check the log output to verify that the test ran correctly.



```
Task 1 : Task 1
------------------------------------------

#   Test Name          Pin    Status
--- ------------------ ------ ------
001 Test 1.1           -      Pass
002 Test 1.2           -      Pass
003 Test 1.3           -      Fail*
004 Test 1.4           -      Pass

Task 2 : Task 2
------------------------------------------

#   Test Name          Pin    Status
--- ------------------ ------ ------
001 Test 2.1           -      Pass
002 Test 2.2           -      Fail*
003 Test 2.3           -      Pass
004 Test 2.4           -      Pass
```

▼ Add the program module events

8. From the Workspace Navigator, **double-click** the Events submodule of the Program module. This will open a new window. At the top of the window is a drop-down list which contains all of the events that belong to the program module.

9. **Click** the drop-down list and select **OnInit()**.

10. In the code editor below, enter text to notify the user that this event has executed. Make sure that you are specific about the module and the event procedure being called. For example:

```
Print "Program OnInit() event has executed."
```

11. Put a similar print statement in the following Program events:

- OnInitTask()
- OnInitTest()
- OnEndTest()
- OnEndTask()
- OnEnd()

Remember to update the print statement to refer to the event that is being called.

▼ Add the system module events

12. From the Workspace Navigator, **double-click** the Events submodule of the System module. This will open the system events editor.
13. In the code editor, put a **Print** statement that declares that the event is being fired. Make sure that it references that this event is coming from the System module. For example:
14. Print “System OnInit() event has executed.”
15. Put a similar print statement in the following System events:

    OnInitSystem()

    OnInitProgram()

    OnInitTask()

    OnInitTest()

    OnEndTest()

    OnEndTask()

    OnEndProgram()

    OnEndSystem()

It is easier to read and analyze the test log in this next section if the test log is printed in plain text rather than HTML, so before running our application we will update or verify our test log output format.

▼ Observe the results in the test log

16. From the file menu, select **Tools | Options**. This will open the options dialog, which is used to set several miscellaneous options concerning the ATEasy development environment.
17. **Click** the **Log** tab.
18. **Uncheck** the option “Use HTML Format As Default”. This will disable HTML output, effectively setting the output to Plain Text only.
19. **Click Close** to dismiss the Options dialog.
20. Press the **Start** button to run your application.

Now, with the print statements in the module events, we can see a visual representation of when events occur within our application's lifecycle with respect to automatic test log generation that ATEasy performs. The log shown below gives a summary of the results that you should see. The events are preceded by a '>>>'. The application being run only has one task with two tests.

```
>>>System OnInit() event has executed.
>>>System OnInitSystem() event has executed.
      Application Header is appended to the log.
>>>System OnInitProgram() event has executed.
>>>Program OnInit() event has executed.
      Program Header is appended to the log.
>>>System OnInitTask() event has executed.
>>>Program OnInitTask() event has executed.
      Task 1 Header is appended to the log.
>>>System OnInitTest() event has executed.
>>>Program OnInitTest() event has executed.
      Test 1.1 code is executed here !
>>>Program OnEndTest() event has executed.
>>>System OnEndTest() event has executed.
      Test 1.1 test results are appended to the log.
>>>System OnInitTest() event has executed.
>>>Program OnInitTest() event has executed.
      Test 1.2 code is executed here !
>>>Program OnEndTest() event has executed.
>>>System OnEndTest() event has executed.
      Test 1.2 test results are appended to the log.
>>>Program OnEndTask() event has executed.
>>>System OnEndTask() event has executed.
>>>Program OnEnd() event has executed.
>>>System OnEndProgram() event has executed.
      Application Footer is appended to the log.
>>>System OnEndSystem() event has executed.
>>>System OnEnd() event has executed.
```

The application, program, and task headers are automatically appended at certain stages. For instance, the application header is appended just prior to the program module initialization and the program header is appended just after the program module has completed initialization. Since OnInitTest() is called prior to each test, it can be used to regularly perform tasks such as instrument manipulation, log appending, or variable initialization. Similarly, OnEndTest() is called just after the test evaluation, so it can be used to modify the test log output before it gets appended to the test log or the TestStatus can be used for conditional branching as we will explore in the rest of this example.

We will next use the event modules to provide a more functional purpose. Using **OnEndTest()**, we will examine the **TestStatus** variable (which would have been updated based on the previous test). If the previous test failed, we will jump to the next Task. If we are currently at the last task, then the application will exit.

▼ Write code to branch on test failure

21. Add the following code to the Program module's OnEndTest event after the print statement:

```
If TestStatus = FAIL
      If Task.Index = 0
            Task EndEvents 2
      Else
            ExitProgram
      Endif
Endif
```

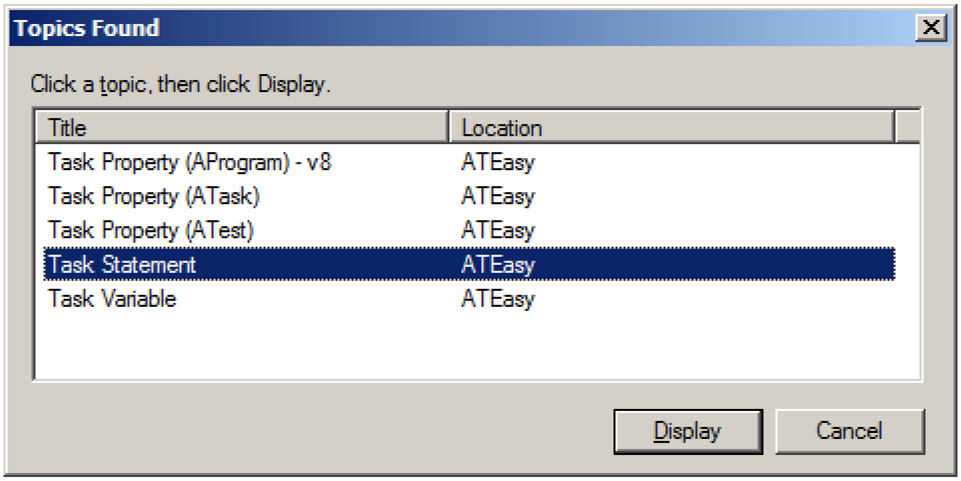
This code checks to see if **TestStatus** is FAIL. If the last test failed, then we will execute the rest of the code. In this case, we will see which task we are using. If Task.Index = 0, this indicates the first task of the program. So, if the **TestStatus** failed and the **Task.Index** = 0, then we will execute the line **Task EndEvents 2**, which jumps to task number 2. If **TestStatus** is FAIL and Task.Index does not equal zero, then we will execute **ExitProgram** which will exit the current program, effectively ending the application.

▼ Getting more information about a reserved keyword or Internal procedure

We used two statements in the **OnEndTest** event that you may be less familiar with: **Task** and **ExitProgram**. Since these statements are part of the ATEasy programming language, there are help articles devoted to explaining their usage in detail.

22. Position the cursor over the **Task** statement in the line **Task EndEvents 2**. While the cursor is blinking on Task, press F1. The ATEasy Help program will open and search for the the phrase Task.

23. There is some ambiguity as task can refer to many things in ATEasy. When prompted, choose **Task Statement** and hit **Display**.



Topics Found

Click a topic, then click Display.

Title	Location
Task Property (AProgram) - v8	ATEasy
Task Property (ATask)	ATEasy
Task Property (ATest)	ATEasy
Task Statement	ATEasy
Task Variable	ATEasy

The topic page for the **Task** statement discusses, in detail, the usage of the **Task** statement, it's optional parameters and provides examples of its usage within the code editor. The ATEasy help covers other statements such as the **Print** statement. It covers Internal variables such as **TestResult** and **TestStatus** and Internal procedures such as **SizeOf** and **InputBox**. Use the help files as a resource to help decipher unfamiliar code or to refresh your memory.

Summary

This exercise demonstrated the use of the events sub-module. In addition to program and system events, the driver module also contains a set of event procedures that can be used. With this example, we have also seen that events can be used to write code in one location and know that it will execute in known points during your application.

Exercise 7 – Using Procedures

Objectives

Create procedures.

Test each procedure.

Create variables and parameters where appropriate.

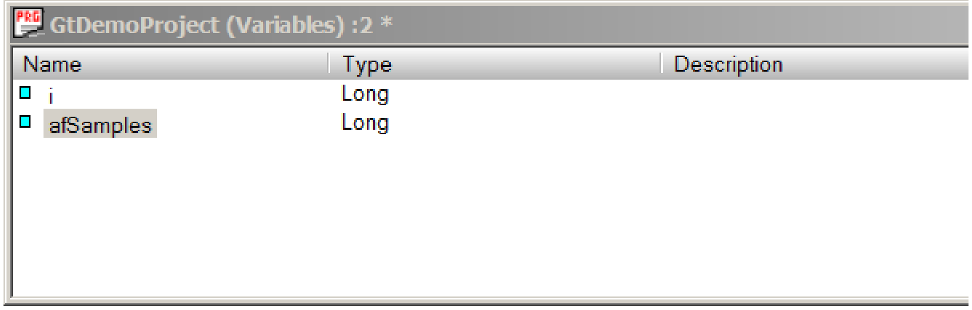
In this exercise, we will first create several procedures within the program module. These procedures will be used to create arrays that represent waveforms. These arrays would be required if we wanted to program an arbitrary waveform generator.

▼ Create a new project using the application wizard.

1. Create a new folder on the Desktop called **Example7**.
2. Use the Application Wizard to create a new test application. Name it **Example7** save it to the **Example7** desktop directory. When prompted, uncheck all of the software driver such as Profile.drv, TestExec.drv, and FaultAnalysis.drv.

▼ Create the program module variables

3. Double-click on the **Variables** sub-module of the program module in the Workspace window. A Variables view opens displaying three columns: Name, Type, and Description.
4. **Right-click** on the variables view and select **Insert Variable Below** from the context menu. A new variable is created and displayed in the view. An edit box displays allowing you to rename the variable name
5. Type the name: **i**.
6. **Right-click** on the **i** variable and select **Insert Variable After** . A new variable is created and inserted after **i**. Rename it by typing **afSamples**.



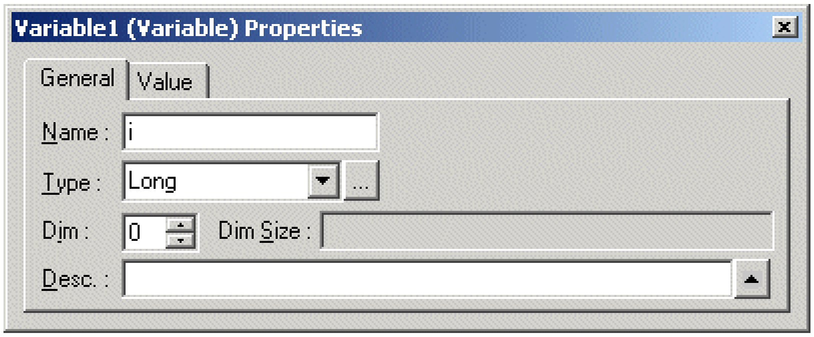
GtDemoProject (Variables) :2 *

Name	Type	Description
i	Long	
afSamples	Long	

In ATEasy 9.0+, changing the variable name to afSamples will automatically change the data type to Float[1].

Setting the Variable Properties

7. **Right-click** on the **i** variable and select **Properties**. Alternately, you can double-click on the variable icon.



Name : i

Type : Long

Dim : 0 Dim Size :

Desc. :

8. Click in the **Desc** field and type **Loop Counter** for the description.

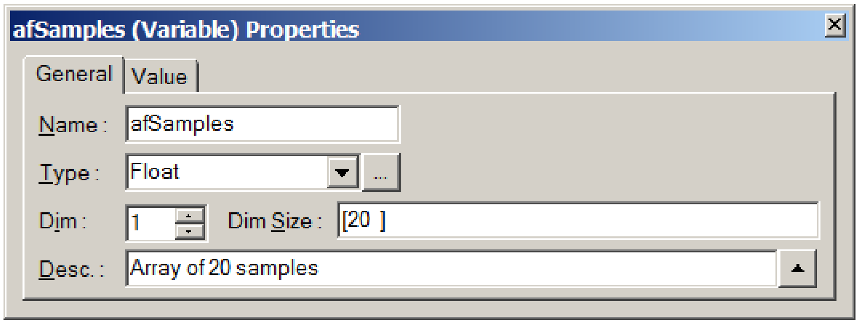
9. Now click on the **afSamples** variable. Note that you did not have to close the Properties window; the properties window updates and displays the currently active object.

10. Update the **Desc** for afSamples to **Array of 20 samples**

11. Click the drop-down arrow next to the Type combo box and select **Float** as the type.

12. Use the tickers to set **Dim** to **1**. This will change afSamples from a scalar variable to a 1-dimensional array.

13. Finally, set the **Dim Size** to [**20**]. This makes **afSamples** a 1-dimensional array of 20 elements.



Name : afSamples

Type : Float

Dim : 1 Dim Size : [20]

Desc. : Array of 20 samples

Using Variable Shorthand

Using the properties editor is a very thorough method of configuring variable properties. But after familiarizing yourself with the proper creation of variables, you can use an ATEasy variable shorthand to enter the name, type, and comments quickly. When you double-click on the variable name, you enter a rename mode. From this mode, you can enter the shorthand which is the name of the variable, a colon **:** the data type, an exclamation mark **!** and the comment. The syntax of the shorthand is:

```
NAME:DATA TYPE!COMMENT
i:Long!Loop Counter
afSamples:Float[20]!Array of 20 samples
```

From now on, you will see a variable shorthand listed for parameters/variables. This is the text that will produce the desired variable.

Define the CalculateRMS Procedure

14. **Right-click** the procedures sub-module from the program module and select **Insert Procedure Below**. A new blank Procedure1 will be created.

15. From the Properties Editor, change the **Name** of the new procedures to **CalculateRMS**.

16. Change the **Returns** type to **Double**. This will change the procedure from a subroutine to a function. It will have to return a value of type double to conclude the function call.

17. Type the following description: **Calculates the root mean square of the values of an array.**

Create local variables and parameters for CalculateRMS

18. **Right-click** in the variables pane of the procedure view. This is the pane with the heading **Name, Type,** and **Description.** It appears below the procedure description panel. Select **Insert Parameter/Variable After**. A new variable named **Variable1** is inserted.

19. This variable will be the array that is passed into the procedure and evaluated. Rename this variable to **pafData**.

20. Repeat steps 18 and 19 to define the following variables: **lSize**, **d**, and **i**. When you have completed, you should have four variables defined as type **Val Long**.

21. Open the properties editor for **pafData**. Set the following properties:

Name:	pafData
Parameter:	Val
Type:	Float
Dim:	1
Desc.:	Array to calc RMS
Shorthand:	pafData:Val Float[]!Array to calc RMS

22. Repeat step 19 for the **lSize** as follows:

Name:	lSize
Parameter:	[Val]
Optional:	*Yes*
Type:	Long
Dim:	0
Desc.:	Number of elements, use the whole array if omitted.

Check the **Optional** check box. This allows the user to pass or not pass an argument here. Set the initial value from the **Value** property page for this parameter as **-1**. If the caller will not provide this argument the value of lSize will be -1.

Shorthand:	lSize:[Val] Long = -1!Number of elements, use the whole array if omitted

23. Repeat step 19 for **d**, a local variable, as follows:

Name:	d
Parameter	None
Type	Double

Dim	0
Desc.	Sum of elements of the array
Shorthand:	d:Double!Sum of elements of the array

24. Repeat step 19 for **i**, another local variable, as follows:

Name:	i
Parameter	None
Type:	Long
Dim:	0
Desc.:	Loop counter
Shorthand:	i:Long!Loop counter

▼ Write the CalculateRMS Procedure Code

25. Type the following in the procedure code area beneath the variables view:

```
! handle optional parameter
! if -1 or not passed then calculate the array size
if lSize=-1
   ! calc num of elements
   lSize=(sizeof pafData)/(sizeof float)
endif
if lSize<=0
   return 0
endif
! square the elements of the array and add them to d
for i=0 to lSize-1 do
   d=d+pafData[i]*pafData[i]
next
! calculate the mean
d=d/lSize
! return the square root of the mean
return sqrt(d)
```

1. Run **CheckIt!** on this procedure to ensure that it was transcribed correctly.

▼ Test the CalculateRMS procedure

26. Rename the Untitled Task to **Procedure Tests**.

27. Rename the Untitled Test to **CalculateRMS Test**.

28. Change the test type to **Tolerance**, the value to **3.3**, the plus to **0.1** and the minus to **0.1**.

29. Now enter the following code into the test code editor:

```
! set array values 1..5
for i=0 to 4
   afSamples[i]=i+1
```

```
next

! calc RMS of array into TestResult
TestResult=CalculateRMS(afSamples, 5)
```

30. You can test your code by selecting the **TestIt!** Command from the **Debug** menu. This command will run only this test. After the test runs, take a look at the test log and verify that the test you just wrote has a PASS status.

31. Within the CreateSineWave procedure, create the following procedure variables:

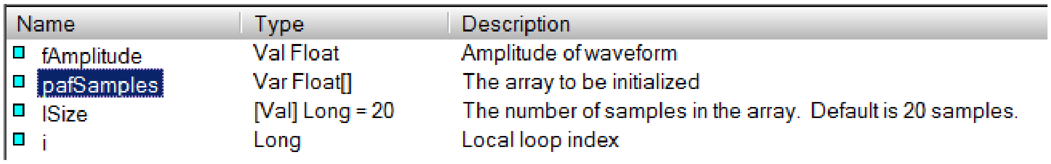
Name:	fAmplitude
Parameter:	Val
Type:	Float
Dim:	0
Desc:	Amplitude of waveform
Shorthand:	fAmplitude:Val Float!Amplitude of waveform

Name:	pafSamples
Parameter:	Var
Type:	Float
Dim:	1
Desc:	The array to be initialized
Shorthand:	pafSamples:Var Float[]!The array to be initialized

Name:	lSize
Parameter:	[Val]
Optional:	*Yes*
Type:	Long
Dim:	0
Default Value:	20
Desc:	The number of samples in the array, default 20
Shorthand:	lSize:[Val] Long = 20!The number of samples in the array, default 20

Name:	i
Parameter:	None
Type:	Long
Dim:	0
Desc:	Local loop index
Shorthand:	i:Long!Local loop index

The variables view for this procedure should now look similar to the following:



Name	Type	Description
□ fAmplitude	Val Float	Amplitude of waveform
□ pafSamples	Var Float[]	The array to be initialized
□ lSize	[Val] Long = 20	The number of samples in the array. Default is 20 samples.
□ i	Long	Local loop index

32. Add the following code in the procedure code editor below:

```
For i=0 to lSize-1
   pafSamples[i]=fAmplitude*sin((2*PI*i)/lSize)
next
```

Create a new procedure to create a triangle wave

Since the triangle wave creation is going to be similar to the sinusoidal wave generation, we can use the CreateSineWave procedure as a basis for CreateTriangleWave.

33. **Right-click** CreateSineWave and select **Copy**. Then **right-click CreateSineWave** again and select **Paste**. A dialog will pop up asking what your intention is.

34. Select **Duplicate**. A duplicate procedure named **CreateSineWave1** will be added to the program module.

35. Using the Properties window, change the name of the new procedure to **CreateTriangleWave** and change the description to **Populates an array with a triangular waveform**.

36. Make sure you are focuses on CreateTriangleWave and erase the code in the code editor. Replace it with the following:

```
For i=0 to lSize-1
   pafSamples[i]=abs((2*fAmplitude*i/lSize)-fAmplitude)
next
```

Create tests for the CreateSineWave test

37. **Right-click** the **CalculateRMS Test** and select **Insert Test After**.

38. Rename the new Untitled Test to **CreateSineWave Test**.

39. Put the following code into the code editor for **CreateSineWave** test:

```
CreateSineWave(5, afSamples, 20)
! calc RMS of array into TestResult
TestResult=CalculateRMS(afSamples, 20)
```

This will create a 20-point sinusoidal waveform with amplitude of 5 and loads it into **afSamples**. Then the RMS of the array is calculated.

40. Edit the **CreateSineWave test** properties and set some PASS/FAIL characteristics. Remember that the expected RMS for a sine wave is amplitude divided by square root of two.

▼ Create a test for the CreateTriangleWave procedure

41. **Right-click** the **CreateSineWave** test and select **Insert Test After**.

42. Rename the new Untitled Test to **CreateTriangleWave Test**.

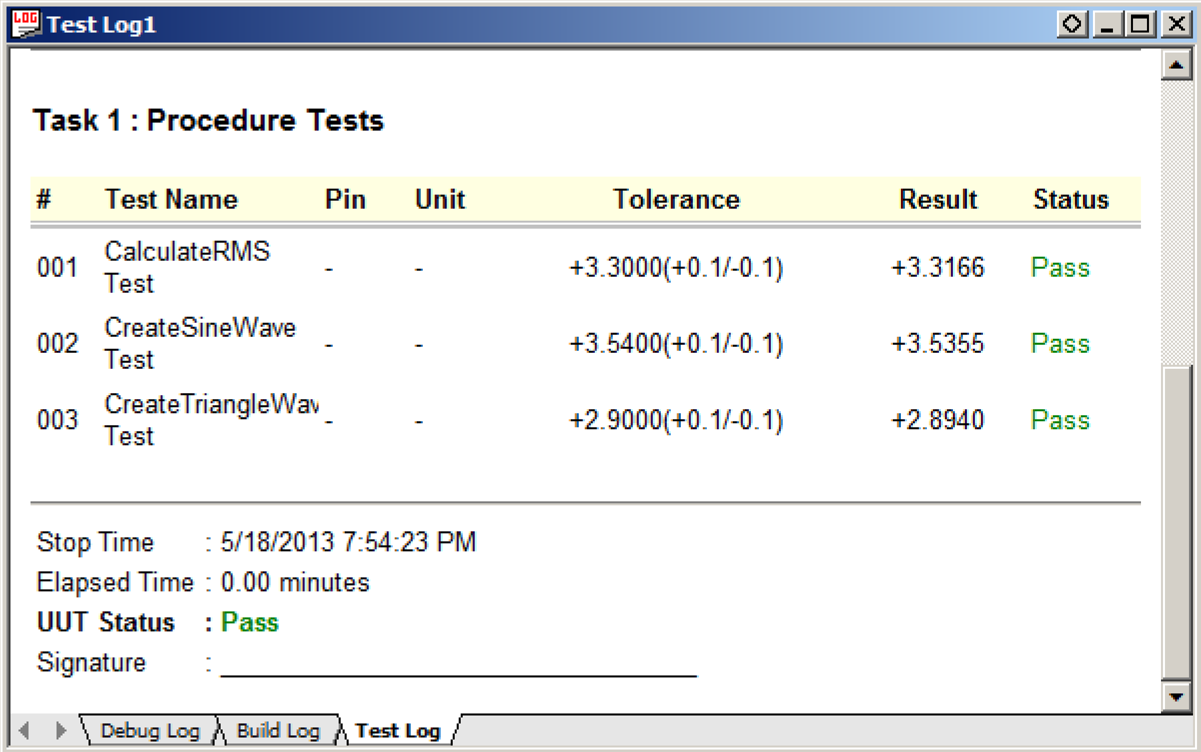
43. Put the following code into the code editor for CreateTriangleWave test:

```
CreateTriangleWave(5, afSamples)
! calc RMS of array into TestResult
TestResult=CalculateRMS(afSamples, 20)
```

This will create a 20-point triangular waveform with amplitude of 5 and loads it into afSamples. Then the RMS of the array is calculated.

44. Edit the **CreateTriangleWave test** properties and set some PASS/FAIL characteristics. Remember that the expected RMS for a triangle wave is amplitude divided by square root of three.

At this point, you can **Start** the application and you should see that all three tests PASS.



Task 1 : Procedure Tests

#	Test Name	Pin	Unit	Tolerance	Result	Status
001	CalculateRMS Test	-	-	+3.3000(+0.1/-0.1)	+3.3166	Pass
002	CreateSineWave Test	-	-	+3.5400(+0.1/-0.1)	+3.5355	Pass
003	CreateTriangleWav Test	-	-	+2.9000(+0.1/-0.1)	+2.8940	Pass

Stop Time : 5/18/2013 7:54:23 PM
Elapsed Time : 0.00 minutes
UUT Status : Pass
Signature : ________________________

Summary

This exercise demonstrated the creation and use of procedures and variables, including both program variables and procedure variables and also looked at how we can use some of the debugging tools within the ATEasy environment. This code created for this exercise will be used again later as the arrays are programmed into the burst memory of the DACs.

Exercise 8 – Creating an ATEasy Driver

Objectives

Create a driver interface.

Configure the driver in the system layer.

Define I/O tables for the digital subsystem.

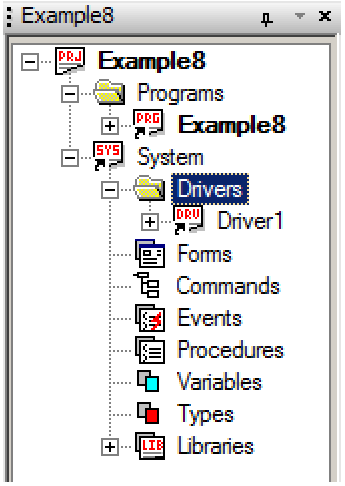
Debug using Monitor View.

In this exercise, we will create an ATEasy driver module from scratch. This driver will be a reusable module that contains instructions to manipulate the digital subsystem of the GT98901. This involves the creation of IOProcedures and the monitoring of text-based communication via specific interfaces such as USB, RS-232, and Winsock.

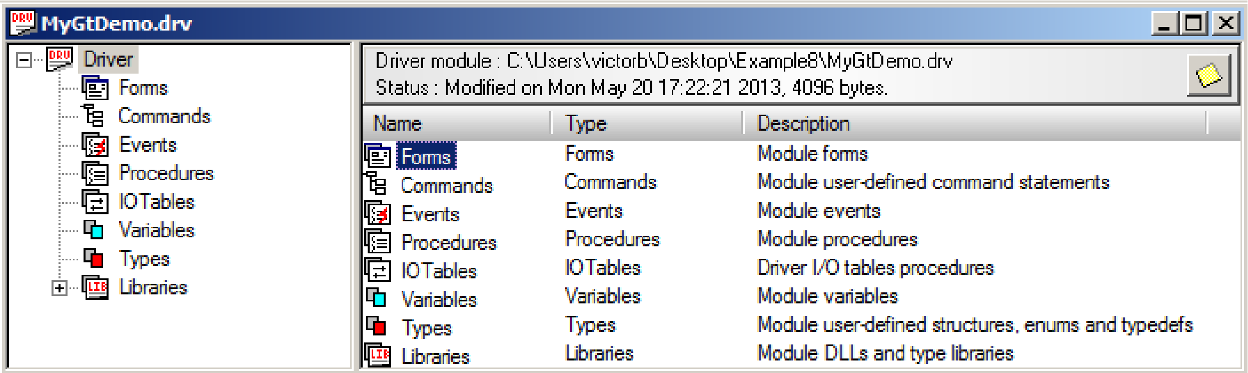
▼ Create a new test application and driver

1. Create a new directory of the desktop called **Example8**. Then use the ATEasy **Application Wizard** to create a new test application. Name it **Example8** and save it to the **Example8** directory on the Desktop. **Uncheck** all software drivers.

2. Within the system module, **right-click** on the **Drivers** folder and select **New Driver**. An empty driver module will be created called **Driver1**:



3. **Double-click** the **Driver1** entry in the Workspace Navigator. This will open the driver module in the MDI workspace.

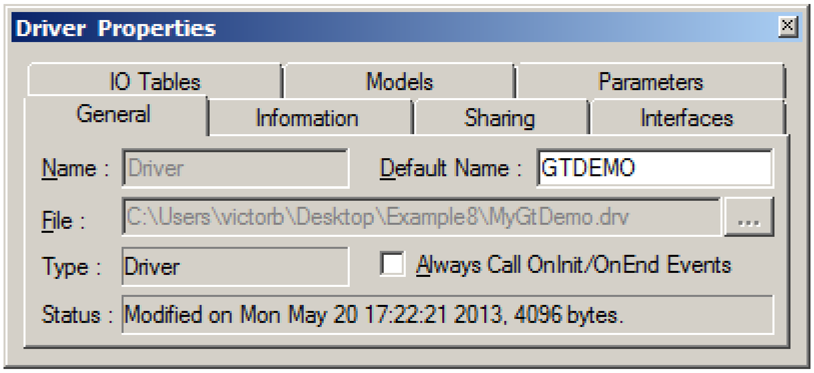


Driver module : C:\Users\victorb\Desktop\Example8\MyGtDemo.drv
Status : Modified on Mon May 20 17:22:21 2013, 4096 bytes.

Name	Type	Description
Forms	Forms	Module forms
Commands	Commands	Module user-defined command statements
Events	Events	Module events
Procedures	Procedures	Module procedures
IOTables	IOTables	Driver I/O tables procedures
Variables	Variables	Module variables
Types	Types	Module user-defined structures, enums and typedefs
Libraries	Libraries	Module DLLs and type libraries

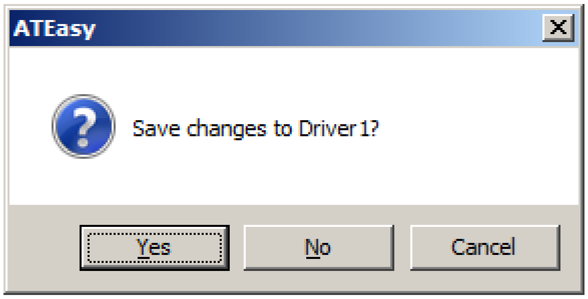
4. Open the **Properties Editor** for MyGtDemo by **right-clicking** the entry **Driver** and selecting **Properties**.

5. Change the **Default Name** of the driver to **GTDEMO**. With this default name set, each time this driver is instanced and added to a system it will attempt to claim the name **GTDEMO**. If the name is already used by another symbol, it will use the name GTDEMO1, GTDEMO2, etc.



Name : Driver
Default Name : GTDEMO
File : C:\Users\victorb\Desktop\Example8\MyGtDemo.drv
Type : Driver
☐ Always Call OnInit/OnEnd Events
Status : Modified on Mon May 20 17:22:21 2013, 4096 bytes.

6. Select **File | Save All** from the file menu. This will prompt you to save your new driver file. Select **Yes** from the prompt.



Save changes to Driver 1?

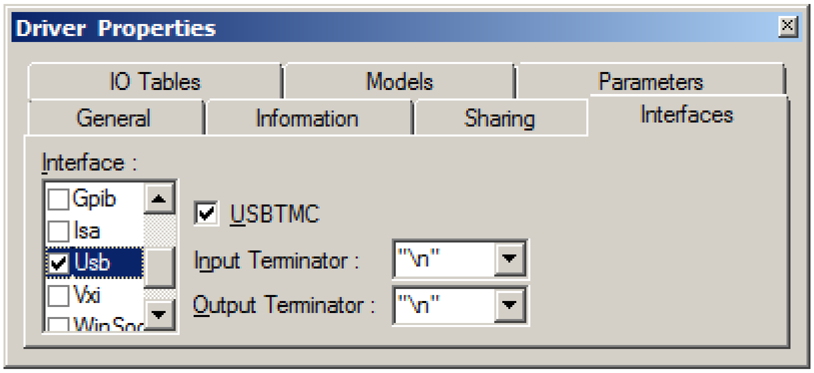
7. Save the file in your **Example8** directory on the Desktop. Name it **MyGtDemo.drv**

▼ Configure the Driver Interface

8. **Open** the **Properties Editor** for the **GTDEMO** driver (not the shortcut) you just created if it is not already open.

9. **Select** the **Interfaces** tab.

10. By default, the None interface is selected. Since we know that the GT98901 uses the USB interface and no other. **Check** the **USB** option. Leave the rest of the settings as default.



Driver Properties

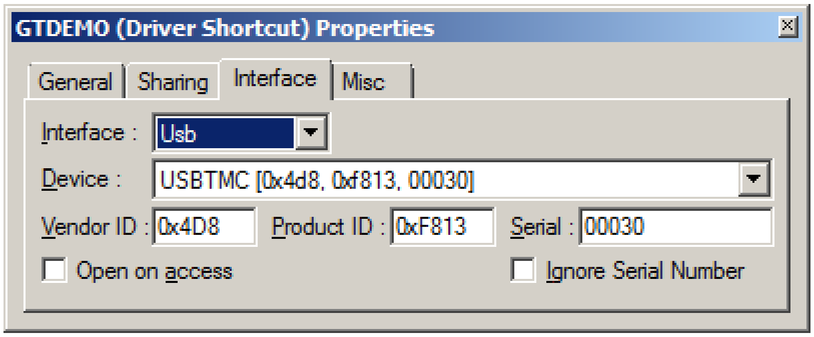
Interface :
- □ Gpib
- □ Isa
- ☑ Usb
- □ Vxi
- □ WinSoc

☑ USBTMC

Input Terminator : "\n"

Output Terminator : "\n"

11. Now that the interface has been set, you can select the **GTDEMO** (Driver Shortcut) Properties to set the interface to your connected GTDEMO board. Remember that your Device identifier may be **USBTMC** or **USB Test and Measurement** or **GT98901 Educational Demo Board**.



GTDEMO (Driver Shortcut) Properties

Interface : Usb

Device : USBTMC [0x4d8, 0xf813, 00030]

Vendor ID : 0x4D8 Product ID : 0xF813 Serial : 00030

□ Open on access □ Ignore Serial Number

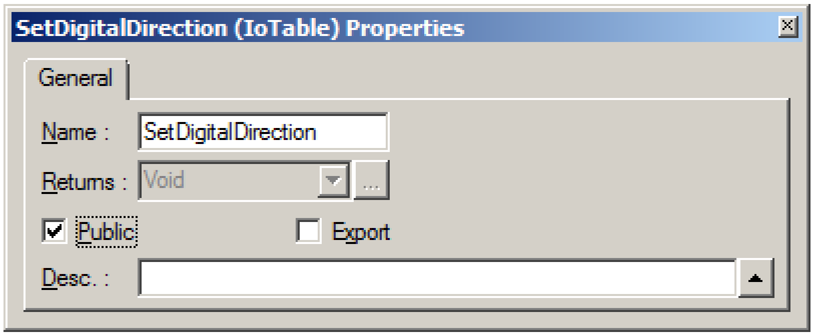
▼ Creating an IOProcedure to set the direction of the digital pins

12. **Expand** the **GTDEMO** Driver module in the Workspace Navigator.

13. **Right-click** the **IOTables** sub-module and select **Insert IOTable Below**. A new IOProcedure called **IOTable1** will be created within the IOTables sub-module.

14. Open the Properties Editor for the new **IOTable1** procedure. Change the **Name** to **SetDigitalDirection**.

15. By default, IOTable Procedure are not set to public. The only statements that are public by default are user-defined commands. For this exercise, we will be setting our IOProcedures to public as we create them.



SetDigitalDirection (IoTable) Properties

Name : SetDigitalDirection

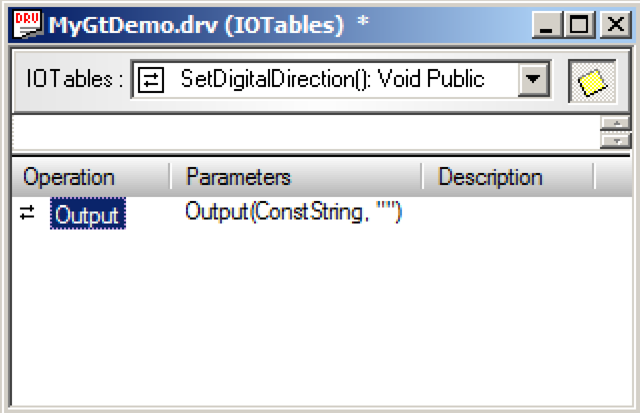
Returns : Void

☑ Public □ Export

Desc. :

16. **Double-click** the **SetDigitalDirection** IOTable procedure to open the object view in the MDI work area. Under the description area is a field where we can enter IOOperations.

17. **Right-click** the operation area and select **Insert IoOperation After**. This will add a new operation to the SetDigitalDirection procedure.



MyGtDemo.drv (IOTables) *

IOTables : SetDigitalDirection(): Void Public

Operation	Parameters	Description
Output	Output(ConstString, "")	

18. Open the properties editor for the **Output** operation. In this window, you can change the Operation to one of these types:

Output: Append some ASCII characters to the output buffer.

Input: Take data from the input buffer and assign it to a parameter.

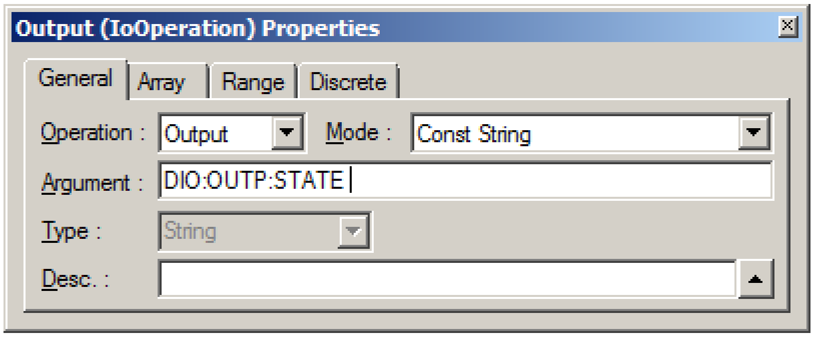
Send: Send the contents of the output buffer to the target device.

Receive: Receive data from the target device and add it to the input buffer.

Delay: Places a delay for X seconds within this IOTable Procedure.

Trigger: Sends a trigger signal to the device (for GPIB and VXI).

19. The first operation will output the SCPI command used to set the digital pin direction. This command can be found in the SCPI commands appendix of this manual. Type the SCPI command into argument field follow by a single space: **"DIO:OUTP:STATE "**. We add the **white space** after the SCPI command because we will next be adding a parameterwhich specifies which digital channels are output and which are input.



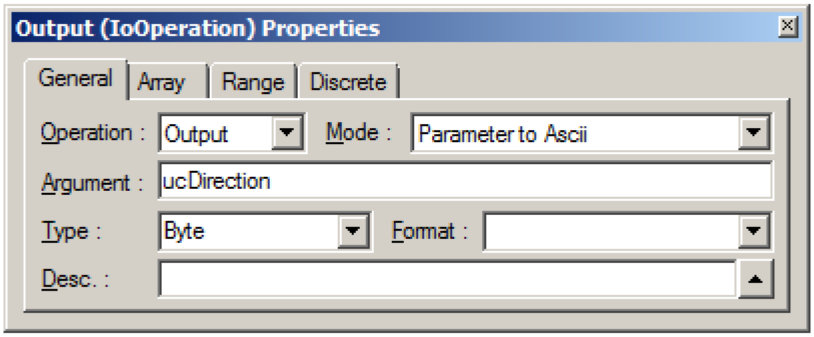
Operation : Output
Mode : Const String
Argument : DIO:OUTP:STATE
Type : String
Desc. :

20. **Right-click** on the first Output operation and select **Insert IoOperation After**.

21. For the second Output operation, we will allow the use to pass in the value as a parameter. Change the **Mode** of the second Output operation to **Parameter to Ascii.**

22. Change the **Argument** to **ucDirection**. This parameter name designation will help the caller to know what the intention of this parameter is.

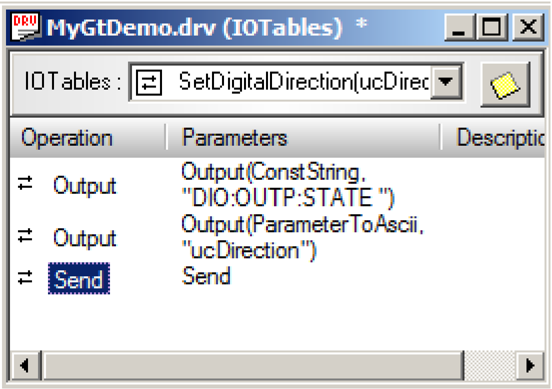
23. Finally, change the **Type** to **Byte** as specified by the SCPI listing.



Operation : Output
Mode : Parameter to Ascii
Argument : ucDirection
Type : Byte
Format :
Desc. :

24. **Right-click** on the second Output operation and select **Insert IoOperation After**.

25. Now that we have constructed our message to the instrument, we can send it to the GT98901. Open the Properties Window for the last IoOperation in the list. Change the **Operation** type to **Send**.



IOTables : SetDigitalDirection(ucDirec

Operation	Parameters	Descriptic
⇄ Output	Output(ConstString, "DIO:OUTP:STATE ")	
⇄ Output	Output(ParameterToAscii, "ucDirection")	
⇄ Send	Send	

Query operations such as GetDigitialDirection tend to have the following structure:

Output a query.

Send the query to the device.

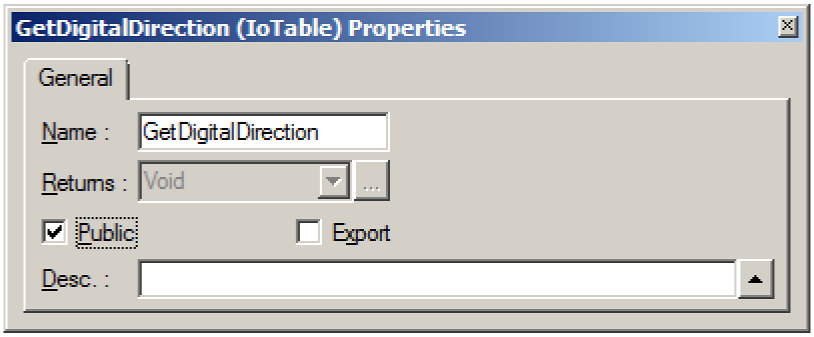
Receive a response from the device.

Input the response into a parameter.

▼ Creating an IOProcedure to read the direction of the digital pins

26. **Right-click** the completed **SetDigitalDirection** IOTable Procedure and select **Insert IoTable After** to create a new IOTable procedure.

27. Change the **Name** of this procedure to **GetDigitalDirection**. Make sure you set it to public as well.



GetDigitalDirection (IoTable) Properties

General

Name : GetDigitalDirection

Returns : Void

☑ Public ☐ Export

Desc. :

28. **Right-click** in the **GetDigitalDirection** object view area and select **Insert IoOperation After** to add a new IOOperation.

29. Leave the Operation type as Output and set the argument to **DIO:OUTP:STATE?** which is the SCPI command to query the **GT98901** for the direction of the digital pins.

30. **Right-click** the Output operation and select **Insert IoOperation After**.

31. Change the second output operation type to **Send**.

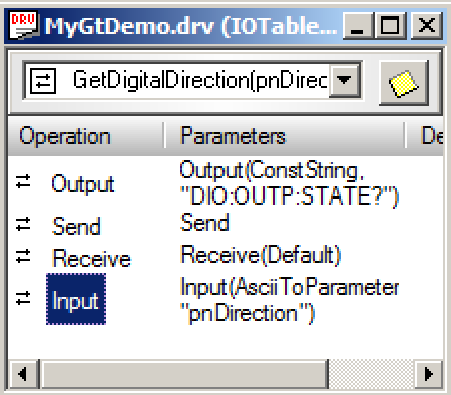
32. **Right-click** the Send operation and select **Insert IoOperation After**.

33. This next operation will attempt to read data from the target device into the input buffer. Change this newest IOOperation **Type** to **Receive**.

34. **Right-click** the Receive IOOperation and select **Insert IOOperation After.**

35. The final IOOperation will read data from the input buffer and load it into a parameter. Change the newest **Operation** to **Input**. Change the **Mode** to **Ascii to Parameter.**

36. Change the **Argument** to **pnDirection** and change the **Type** to **Short**.



Operation	Parameters	De
Output	Output(Const String, "DIO:OUTP:STATE?")	
Send	Send	
Receive	Receive(Default)	
Input	Input(AsciiToParameter "pnDirection")	

▼ Creating an IOProcedure to write to the digital pins

37. **Right-click** the completed GetDigitalDirection IOTable Procedure and select **Insert IoTable After** to create a new IOTable procedure.

38. Change the **Name** of this procedure to **SetDigitalData**. Make sure you set it to public as well.

39. **Double-click** the **SetDigitalData** IOTable procedure to open the object view in the MDI work area.

40. **Right-click** the operation area and select **Insert IoOperation After**. This will add a new operation to the SetDigitalDirection procedure.

41. Open the properties editor for the **Output** operation. Change the **Argument** to **"DIO:OUTP "**. Make sure that there is a white space after the SCPI command DIO:OUTP because we will next be adding a parameter which specifies which digital channels to write a low value and which to write a high value.

42. **Right-click** on the first Output operation and select **Insert IoOperation After**.

43. For the second Output operation, we will allow the use to pass in the value as a parameter. Change the **Mode** of the second Output operation to **Parameter to Ascii.**

44. Change the **Argument** to **ucData**. This parameter name designation will help the caller to know what the intention of this parameter is.

45. Finally, change the **Type** to **Byte** as specified by the SCPI listing.

46. **Right-click** on the second Output operation and select **Insert IoOperation After**.

47. Now that we have constructed our message to the instrument, we can send it to the GT98901. Open the Properties Window for the last IoOperation in the list. Change the **Operation** type to **Send**.

▼ Creating an IOProcedure to read from the Digital Pins

48. **Right-click** the completed **SetDigitalData** IOTable Procedure and select **Insert IoTable After** to create a new IOTable procedure.

49. Change the **Name** of this procedure to **GetDigitalData**. Make sure you set it to public as well.

50. **Right-click** in the **GetDigitalData** object view area and select **Insert IoOperation After** to add a new IOOperation.

51. Leave the Operation type as **Output** and set the argument to **DIO:INP?** which is the SCPI command to query the read the current data off the digital pins.

52. **Right-click** the Output operation and select **Insert IoOperation After**.

53. Change the second output operation type to **Send**.

54. **Right-click** the Send operation and select **Insert IoOperation After**.

55. This next operation will attempt to read data from the target device into the input buffer. Change this newest IOOperation **Type** to **Receive**.

56. **Right-click** the Receive IOOperation can select **Insert IoOperation After.**

57. The final IOOperation will read data from the input buffer and load it into a parameter. Change the newest **Operation** to **Input**.

58. Change the **Argument** to **pnData** and change the **Type** to **Short**.

We will now test each of the IOProcedures that we just created. Since we will be writing and reading digital lines, we will first need to loopback some of the digital lines. Before you continue, wire the demo board as follows:

▼ Test the new IOTable Procedures

```
D0 to   D4
D1 to   D5
D2 to   D6
D3 to   D7
```

59. In the **Example8** program module, navigate to the Tests sub-module and rename the Untitled Task to **IOProcedure Tests** and rename the Untitled Test to **Set/Get DigitalDirection Test**.

    The **SetDigitalDirection** IOTable procedure that we created earlier accepts a one-byte parameter. Each of the bits in the parameter refers to a digital channel. For instance, bit 0 refers to D0, bit 1 refers to D1, etc. Setting a low value for a bit means that that the associated channel will be set to output, a high value means that the associated channel should be set to input. For this first test, we will use the **SetDigitalDirection** IOTable procedure to set digital ports zero through three to output and digital ports four through seven to input. Then we will use the **GetDigitalDirection** IOTable procedure to verify that the GT98901 has updated properly.

60. In the code editor for this first test, type the following:

```
!Set D0-D3 direction to output
!Set D4-D7 direction to input
GTDEMO.SetDigitalDirection(0xF0)
GTDEMO.GetDigitalDirection(TestResult)
```

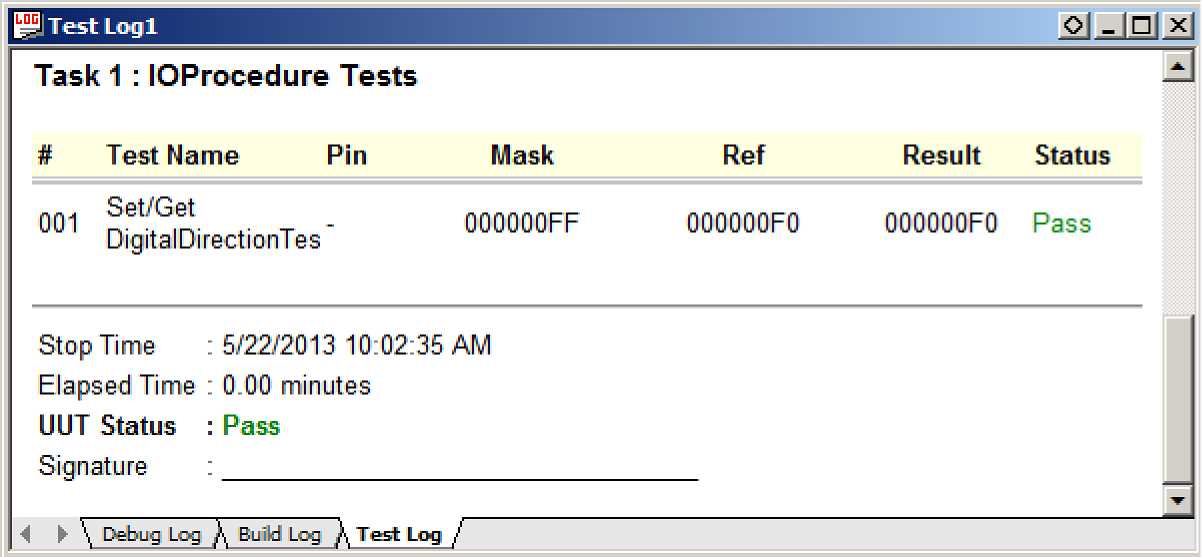
61. We should expect **GetDigitalDirection** to return a value equivalent to 0xF0. Open the **Properties Editor** for the **Set/Get DigitalDirection Test**.

62. Set the **Type** of this test to **RefX** since we are expecting a hexadecimal response.

63. Since we are only interested in the first byte of any response, set the **Mask** to **0xFF**. This will mask off any other bytes of the response.

64. Type the value that we are expecting, **0xF0**, into the **Ref** field.

65. Use **TestIt!** from the **Debug** menu to try it out.



Test Log1

Task 1 : IOProcedure Tests

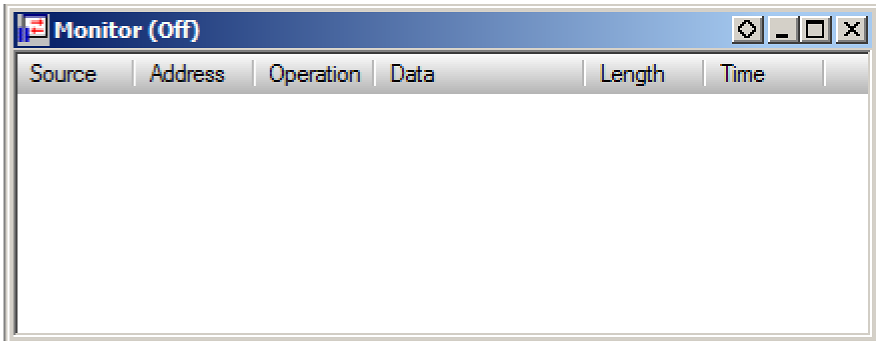
#	Test Name	Pin	Mask	Ref	Result	Status
001	Set/Get DigitalDirectionTes	-	000000FF	000000F0	000000F0	Pass

Stop Time : 5/22/2013 10:02:35 AM
Elapsed Time : 0.00 minutes
UUT Status : Pass
Signature : ______________________________

Monitor View is used to view the communication that takes place between ATEasy and the devices it is controlling through ports such as RS-232, USB, and WinSock (TCP/IP and UDP). Since we are communicating through USB to set instructions to the GT98901, we will explore Monitor View now.

▼ Using Monitor View

66. Select **View | Monitor** from the menu bar to open the Monitor window. This window may be floating or docked when it opened, depending on the last way it was used.

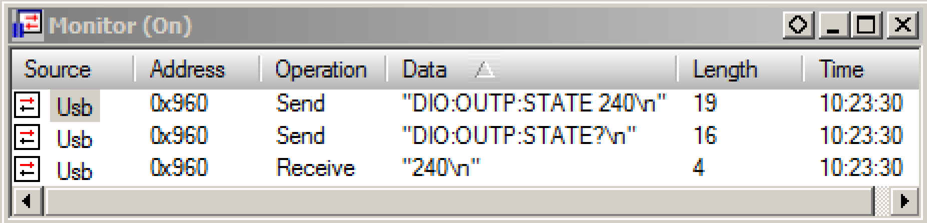


Monitor (Off)

Source	Address	Operation	Data	Length	Time

67. **Right-click** on the Monitor View and select **Start Logging** to begin monitoring for text communications.

68. **Select** your **Set/Get DigitalDirection Test** again to make it the selected item and use **TestIt!** again. This time, the communication will be appended to the monitor view as it send and received.



Source	Address	Operation	Data	Length	Time
Usb	0x960	Send	"DIO:OUTP:STATE 240\n"	19	10:23:30
Usb	0x960	Send	"DIO:OUTP:STATE?\n"	16	10:23:30
Usb	0x960	Receive	"240\n"	4	10:23:30

In the Monitor window, we can see that our **SetDigitalDirection**(0xF0) resulted in the instruction that was sent: “DIO:OUTP:STATE 240\n”, then we queried the instrument for its current direction: “DIO:OUTP:STATE?\n”, and finally received the response “240\n” which is equivalent to 0xF0.

▼ Test the Remaining Procedures

69. In the Example8 program module, add a new test to your **IOProcedure Tests** task and rename the Untitled Test to **Set/Get DigitalData 0x55**.

70. In the code editor for this first test, type the following:

```
GTDEMO.SetDigitalData(0x5)
GTDEMO.GetDigitalData(TestResult)
```

71. We should expect **GetDigitalData** to return a value equivalent to 0x55. This is because we will be outputting 0x5 on the low nibble looped to a 0x5 on the high nibble. Open the **Properties Editor** for the Set/Get DigitalData Test.

72. Set the **Type** of this test to **RefX** since we are expecting a hexadecimal response.

73. Since we are only interested in the first byte of any response, set the **Mask** to **0xFF**. This will mask off any other bytes of the response.

74. Type the value that we are expecting, **0x55**, into the **Ref** field.

75. In the **Example8** program module, add a new test to your **IOProcedure Tests** task and rename the Untitled Test to **Set/Get DigitalData 0xAA**.

76. In the code editor for this test, type the following:

```
GTDEMO.SetDigitalData(0xA)
GTDEMO.GetDigitalData(TestResult)
```

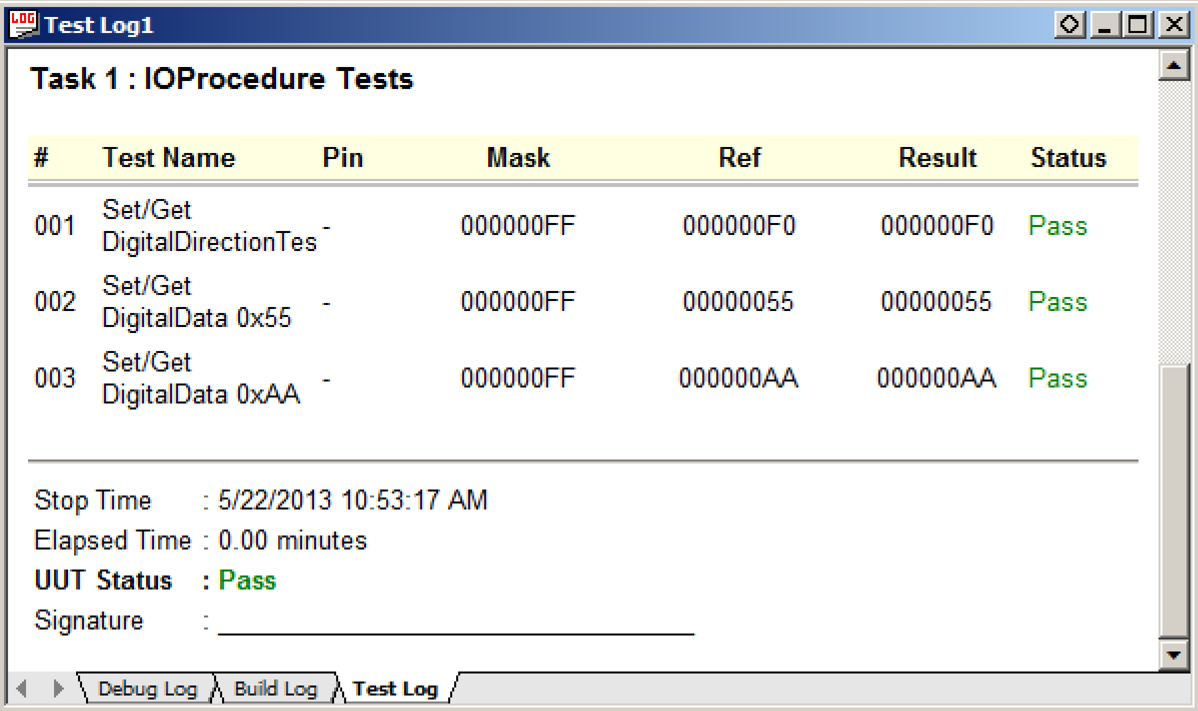
77. We should **expect GetDigitalData** to return a value equivalent to 0xAA. Open the **Properties Editor** for the **Set/Get DigitalData Tes**t.

78. Set the **Type** of this test to **RefX** since we are expecting a hexadecimal response.

79. Since we are only interested in the first byte of any response, set the **Mask** to **0xFF**. This will mask off any other bytes of the response.

80. Type the value that we are expecting, **0xAA**, into the **Ref** field.

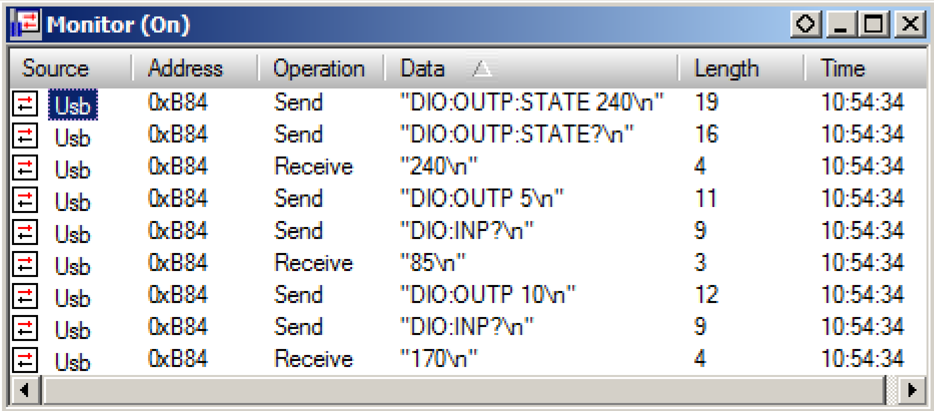
81. Run your application and review the results in the test log and Monitor window.



Task 1 : IOProcedure Tests

#	Test Name	Pin	Mask	Ref	Result	Status
001	Set/Get DigitalDirectionTes	-	000000FF	000000F0	000000F0	Pass
002	Set/Get DigitalData 0x55	-	000000FF	00000055	00000055	Pass
003	Set/Get DigitalData 0xAA	-	000000FF	000000AA	000000AA	Pass

Stop Time : 5/22/2013 10:53:17 AM
Elapsed Time : 0.00 minutes
UUT Status : Pass
Signature : ______________________________



Source	Address	Operation	Data	Length	Time
Usb	0xB84	Send	"DIO:OUTP:STATE 240\n"	19	10:54:34
Usb	0xB84	Send	"DIO:OUTP:STATE?\n"	16	10:54:34
Usb	0xB84	Receive	"240\n"	4	10:54:34
Usb	0xB84	Send	"DIO:OUTP 5\n"	11	10:54:34
Usb	0xB84	Send	"DIO:INP?\n"	9	10:54:34
Usb	0xB84	Receive	"85\n"	3	10:54:34
Usb	0xB84	Send	"DIO:OUTP 10\n"	12	10:54:34
Usb	0xB84	Send	"DIO:INP?\n"	9	10:54:34
Usb	0xB84	Receive	"170\n"	4	10:54:34

▼ Optional extra exercise

To ensure the GT98901 is always starting from a known state, it might be good to start this application from a RESET state. Create another IOTable procedure to execute the *RST operation. Check Appendix B of this manual to examine syntax and function.

Summary

This exercise demonstrated the creation of IOTable procedures and how to use the Monitor window to troubleshoot your IOTables.

Exercise 9 – Using Commands

Objectives

Create driver module commands.

Attach Procedures to commands.

Test from within the program module.

This exercise expands upon Exercise 8 by linking the IOTable procedures to user-defined commands.. The commands created will then be used in the program module.

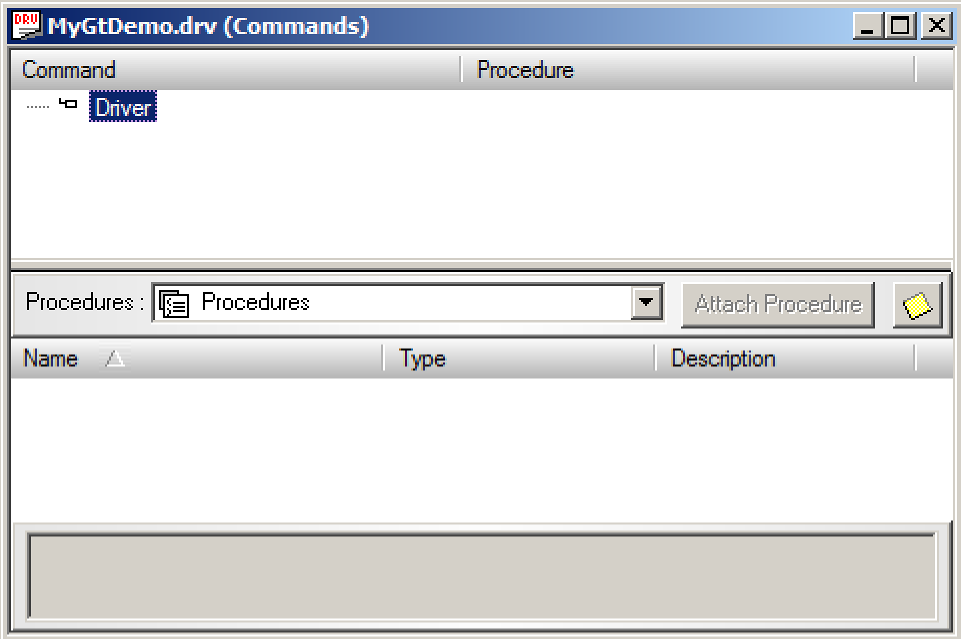
▼ Open the Exercise 8 Workspace

1. This exercise builds upon the MyGtDemo.drv that we created in exercise 8. Open the **Example8.wsp.**

We will start by attaching our created IOTable procedures directly to commands. The commands will be used to directly manipulate the GT98901 so MyGtDemo.drv is where we will insert the commands, not Program or System.

▼ Create a command tree to execute IOTable procedures

2. **Double-click** the Commands sub-module in your MyGtDemo.drv module. This opens the Commands object view in the MDI work area.



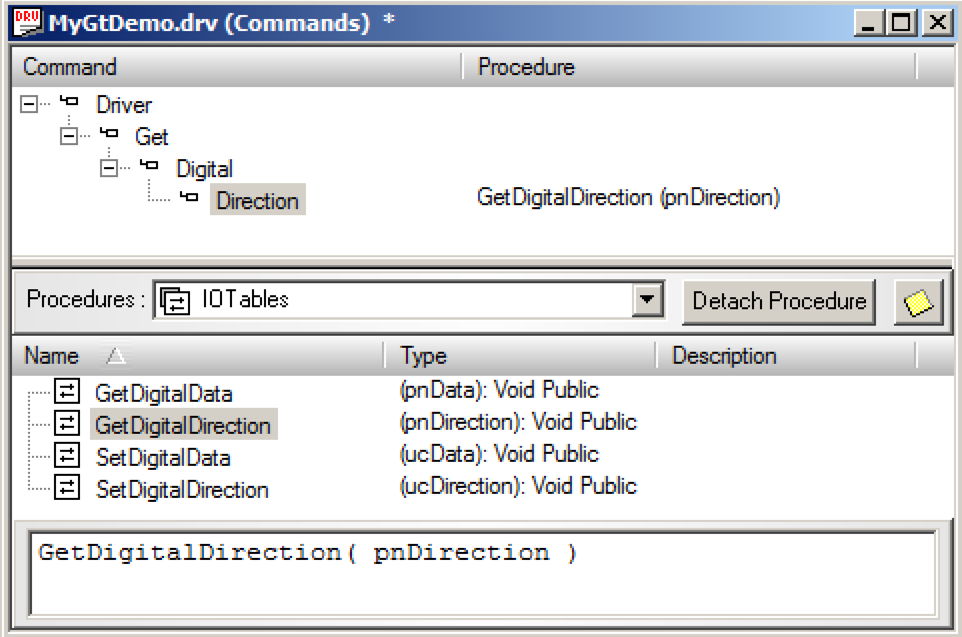
The Driver node is the root node. Every command statement called with start with the root node which is the name of the driver module (GTDEMO, in our case).

3. **Right-click** the Driver node and select **Insert Command Below**. This will create a command called Untitled1 which is a child of the root node.
4. Change the name of the new node to **Get**.
5. **Right-click** the Get node and select **Insert Command Below**. This will create a command called Untitled1 which is a child of the root node.
6. Change the name of the new node to **Digital**.
7. **Right-click** the Digital node and select **Insert Command Below**. This will create a command called Untitled1 which is a child of the root node.
8. Change the name of the new node to **Direction**.

Next, we will attach the GetDigitalDirection IOTable procedure to the bottom node of this branch.

▼ Attach IOTable to a Command

9. Select **IOTables** from the **Procedures** drop-down list.
10. **Click** on the **GetDigitalDirection** IOTable.
11. Then **click** on the lowest node of the branch, **Direction**.
12. **Click** the **Attach Procedure** button.



Command	Procedure
Direction	GetDigitalDirection (pnDirection)

Name	Type	Description
GetDigitalData	(pnData): Void Public	
GetDigitalDirection	(pnDirection): Void Public	
SetDigitalData	(ucData): Void Public	
SetDigitalDirection	(ucDirection): Void Public	

```
GetDigitalDirection( pnDirection )
```

This finalizes the command. When the user types the statement **GTDEMO Get Digital Direction**, the associated GetDigitalDirection IOTable procedure will be run.

13. **Right-click** the Direction node and select **Insert Command After**. Name the new node **Data**.
14. **Attach** the **GetDigitalData** procedure to the **Data** node.

Now we have a second command in our driver. When the user types **GTDEMO Get Digital Data**, the associated GetDigitalData IOTable procedure will be run. Next, we will add some Set commands to complement our Get commands.

▼ Adding Set Commands

15. **Right-click** the Get node and select **Insert Command After**. This will create the Set node as a child of the same parent as the Get node. Name the new node **Set**.

16. **Right-click** the Set node and select **Insert Command Below**. Change the name of the new node to **Digital**.

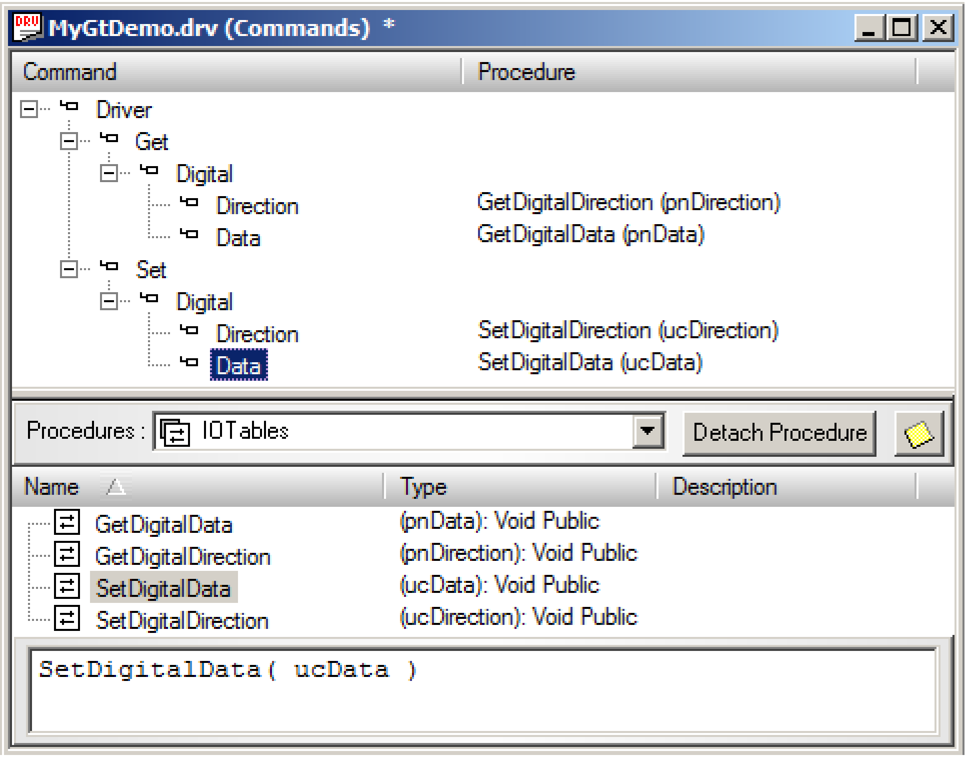
17. **Right-click** the Digital node and select **Insert Command Below**. Change the name of the new node to **Direction**.

18. **Attach** the **SetDigitalDirection** procedure to the **Direction** node.

19. **Right-click** the Direction node and select **Insert Command After**. Change the name of the new node to **Data**.

20. **Attach** the **SetDigitalData** procedure to the **Data** node.

These last few steps add the **GTDEMO Set Digital Direction** and **GTDEMO Set Digital Data** command statements to the driver.



Command	Procedure
Driver	
Get	
Digital	
Direction	GetDigitalDirection (pnDirection)
Data	GetDigitalData (pnData)
Set	
Digital	
Direction	SetDigitalDirection (ucDirection)
Data	SetDigitalData (ucData)

Name	Type	Description
GetDigitalData	(pnData): Void Public	
GetDigitalDirection	(pnDirection): Void Public	
SetDigitalData	(ucData): Void Public	
SetDigitalDirection	(ucDirection): Void Public	

```
SetDigitalData( ucData )
```

▼ Test your newly created commands in the program module

21. **Right-click** on the IOProcedure Tests and select **Insert Task After**. Change the name of the new Untitled Task to **Command Tests**.

22. Change the **Name** of the Untitled Test to **Set / Get Digital Direction**. This test will function identically to the IOProcedure version. The difference will be that commands are called rather than IOTable procedures.

23. Open the Properties Editor for the Set / Get Digital Direction test. Change the **Type** to **RefX**. Change the **Mask** to **0xFF**. Change the **Ref** to **0xF0**.

24. Add the following code to the test:

```
GTDEMO Set Digital Direction(0xF0)
GTDEMO Get Digital Direction(TestResult)
```

25. **Right-click** on the Set / Get Digital Direction test and select **Insert Test After**. Change the name of the new Untitled Test to **Set / Get Digital Data 0x5**.

26. Open the Properties Editor for the **Set / Get Digital Data 0x5** test. Change the **Type** to **RefX**. Change the **Mask** to **0xFF**. Change the **Ref** to **0x55**.

27. Add the following code to the test:

```
GTDEMO Set Digital Data(0x5)
GTDEMO Get Digital Data(TestResult)
```

28. **Right-click** on the Set / Get Digital Data 0x5 test and select **Insert Test After**. Change the name of the new Untitled Test to **Set / Get Digital Data 0xA**.

29. Open the Properties Editor for the Set / Get Digital Direction test. Change the **Type** to **RefX**. Change the **Mask** to **0xFF**. Change the **Ref** to **0xAA**.

30. Add the following code to the test:

```
GTDEMO Set Digital Data(0xA)
GTDEMO Get Digital Data(TestResult)
```

At this point, you can press the Start button to run your application. The IOTable procedure task and the Commands task should run back to back and function in the same manner.

Summary

This exercise demonstrated the creation and use of commands within the driver module. This driver can now be instanced and used in other project. Since commands are set to public by default, these commands can be used to control which procedures your user has access to and which are private.

Exercise 10 – Importing an External DLL Library

Objectives:

Create a driver for an existing instrument.

Create Initialize and CheckError procedures.

Set driver properties.

Create wrapper procedures for low-level library procedures.

Create commands.

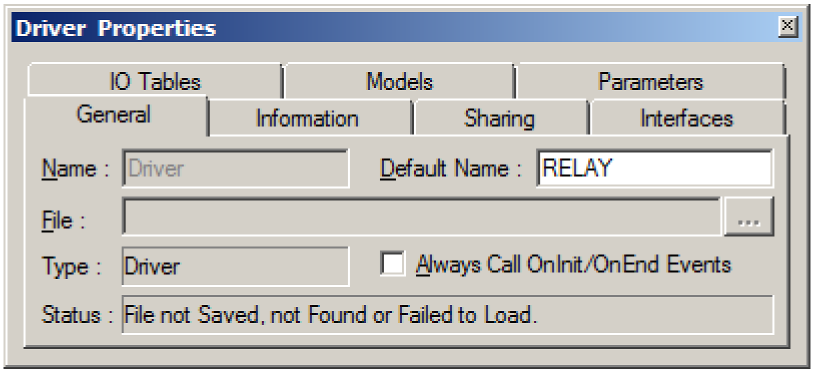
In this exercise, we will create an ATEasy driver to control the switching subsystem of the GT98901 board, which consists of two SPDT relays. We will get a head start on driver development by starting with a DLL library and header file. Then we will create wrapper procedures and helper functions to keep track of the driver and error checking, which will allow us to write program module code that is simple and clutterless.

▼ Create a New Application and Driver

1. Create a new directory of the desktop called **Example10**. Then use the ATEasy **Application Wizard** to create a new test application. Name it **Example10** and save it to the **Example10** directory on the Desktop. **Uncheck** all software drivers.

2. Within the system module, **right-click** on the **Drivers** folder and select **New Driver**. An empty driver module will be created called Driver1.

3. Open the **Driver Properties** for your new **Driver1** and set the **Default Name** to **RELAY**.



Name : Driver
Default Name : RELAY
File :
Type : Driver
☐ Always Call OnInit/OnEnd Events
Status : File not Saved, not Found or Failed to Load.

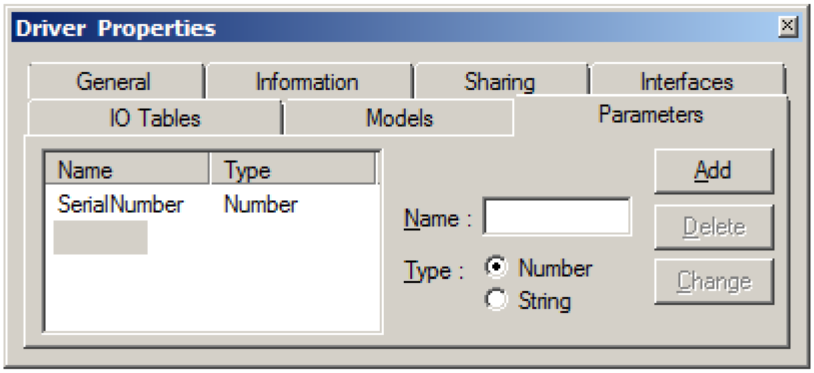
4. Click **Save All** or select **File | Save All** from the menu bar. You will be prompted to save your new driver. Save in in the **Example10** folder on the desktop and name it **GtDemo-Switch.drv**.

Because there is no external interface, we do not use the Interfaces tab to configurate our instrument. This instrument can be identified by its serial number which will be a unique identifier. In the creation of a versatile driver, we must provide the user a method to differentiate between duplicate instances of the same instrument within a system. For this example, we will add a parameter that allows the user to specific the Serial number.

▼ Configure driver parameters

5. Open the **Driver Properties** window and select the **Parameters** page.

6. Type **SerialNumber** into the **Name** field, choose the **Type** as **Number** and click **Add**. Observe that the parameter was added to the parameter's list.



Name	Type
SerialNumber	Number

Name :

Type : ● Number ○ String

Now we will add the external library to this driver, adding an external library such as a DLL in the driver module will add DLL procedures that are available for use within that driver module. Since they are not automatically set to public, they are not immediately available in the program module.

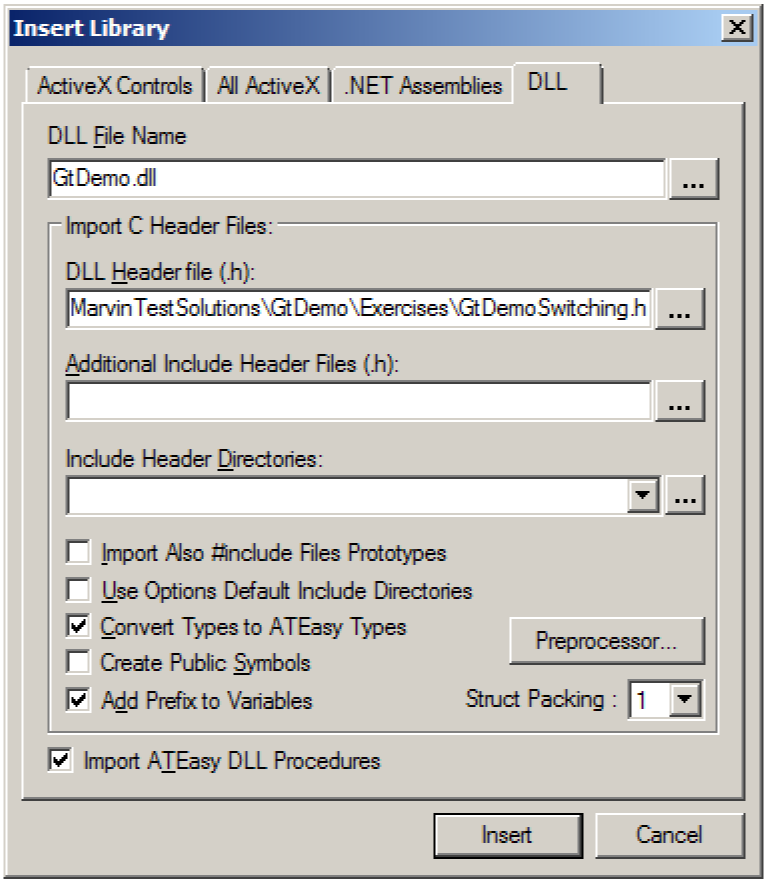
▼ Import the External Library and C Header File

7. Expand the RELAY driver. **Right-click** the **Libraries** sub-module and select **Insert Library Below**. This will open the Insert Library dialog.

8. **Select** the **DLL** page.

9. **Click** the ellipse button next to the DLL File Name field. This will open a file dialog allowing you to select the DLL file to import. Use the File Open dialog to select **C:\Windows\SysWOW64\GtDemo.dll** or **C:\Windows\System32\GtDemo.dll**

10. **Click** the ellipse button next to the DLL Header file (.h) field. This will open a file dialog allowing you to select the Header file to import. Use the File Open dialog to select **`C:\Program Files[ (x86)]\MarvinTestSolutions\GtDemo\Exercises\GtDemoSwitching.h`**



DLL File Name

GtDemo.dll

Import C Header Files:

DLL Header file (.h):

MarvinTestSolutions\GtDemo\Exercises\GtDemoSwitching.h

Additional Include Header Files (.h):

Include Header Directories:

□ Import Also #include Files Prototypes

□ Use Options Default Include Directories

☑ Convert Types to ATEasy Types

□ Create Public Symbols

☑ Add Prefix to Variables

Struct Packing : 1

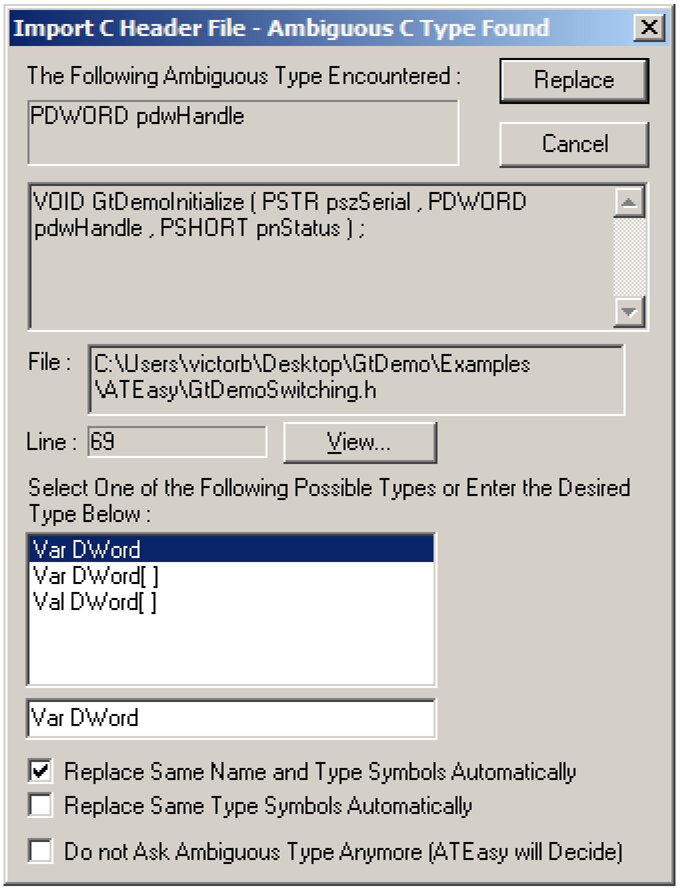
☑ Import ATEasy DLL Procedures

11. Click **Insert**.If next screen does not pop up right click the **GtDemo** driver and select **Import C Header File (.h)…**

When performing a Header file import, ATEasy attempts to convert the native C/C++ data types to ATEasy basic data types by default or creates typedef / aliases if chosen. Because there is a conversion process, ATEasy will prompt the user if ambiguous data types are encountered. For instance, in C++ arrays and variables passed by reference are both designated in the function prototype by a pointer to a data type. Since ATEasy explicitly differentiates between arrays and pointers, it will prompt the user when it encounters this situation.

12. The first ambiguity encountered will be PDWORD *pdwHandle*. This variable is the handle to the demo board that is returned by the Initialize procedure. Because it is a pointer to a DWord, ATEasy needs to know if we are trying to make this parameter a scalar passed by reference or an array. ATEasy guesses that “Var DWord” is the most likely data type to use for the import. Since this is correct, click **Replace**. If the import wizard encounters PDWORD pdwHandle again within this Header file, it will automatically perform the same replacement operation. This is because Replace Same Name and Type Symbols Automatically is currently checked.

    See the diagram on the next page.



Import C Header File - Ambiguous C Type Found

The Following Ambiguous Type Encountered :

PDWORD pdwHandle

VOID GtDemoInitialize (PSTR pszSerial , PDWORD pdwHandle , PSHORT pnStatus) ;

File : C:\Users\victorb\Desktop\GtDemo\Examples\ATEasy\GtDemoSwitching.h

Line : 69

Select One of the Following Possible Types or Enter the Desired Type Below :

Var DWord
Var DWord[ ]
Val DWord[ ]

Var DWord

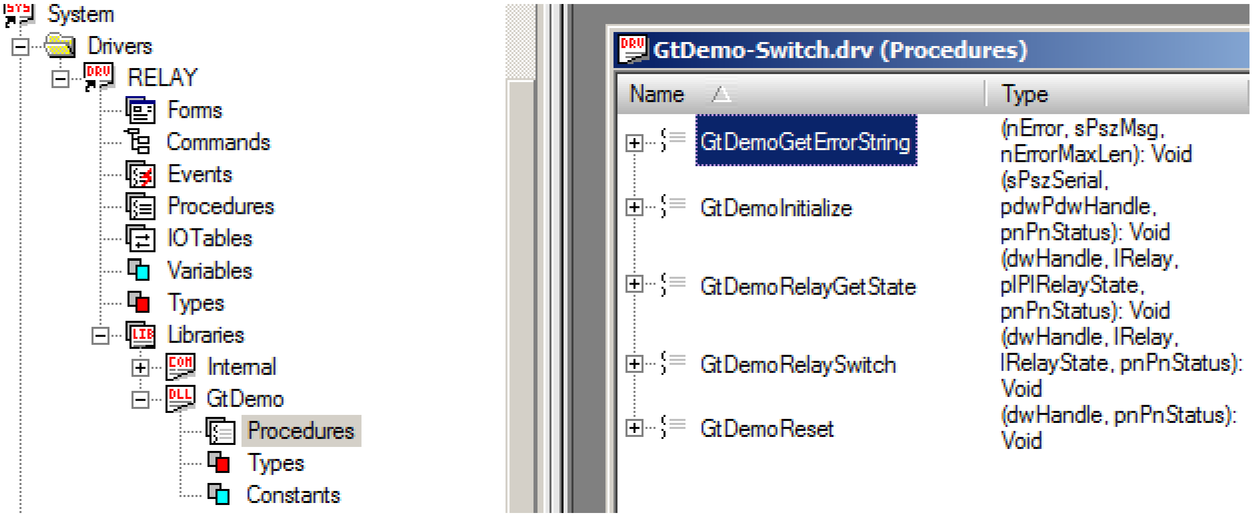
☑ Replace Same Name and Type Symbols Automatically

☐ Replace Same Type Symbols Automatically

☐ Do not Ask Ambiguous Type Anymore (ATEasy will Decide)

13. It will encounter another ambiguity with PSHORT *pnStatus* the returned error code variable, which is again a scalar value. If you are not sure which type to use, it would be prudent to review the instrument manufacturer's software function reference manual. Instead of going through each ambiguity individually, **click** the checkbox next to **Do not Ask Ambiguous Type Anymore**. Then click **Replace**.

14. A new **GtDemo** library will now be added to your **MyGtDemo.drv** driver module. Expand the **GtDemo** library and double-click the procedures sub-module within to view the procedure that were imported using the wizard.



GtDemo-Switch.drv (Procedures)

Name	Type
GtDemoGetErrorString	(nError, sPszMsg, nErrorMaxLen): Void
GtDemoInitialize	(sPszSerial, pdwPdwHandle, pnPnStatus): Void
GtDemoRelayGetState	(dwHandle, lRelay, plPlRelayState, pnPnStatus): Void
GtDemoRelaySwitch	(dwHandle, lRelay, lRelayState, pnPnStatus): Void
GtDemoReset	(dwHandle, pnPnStatus): Void

▼ Add driver module variables

15. **Create** these variables within the driver's Variables sub-module:

Name:	m_dwHandle
Type:	DWord
Desc:	Module variable keeps track of the driver handle

The **CheckError** procedure is a helper function that will be called after each DLL procedure call. Since it will not be exposed to the user via a command, it will allow us to write code to automate error checking without troubling the user.

▼ Create the CheckError procedure

16. In your driver module, **right-click** the Procedures sub-module and select **Insert Procedure Below**.

17. Change the **Name** of Procedure1() to **CheckError**.

18. Add the following variables to the **CheckError** procedure:

Name:	nStatus
Parameter:	Val
Type:	Short
Desc:	Error Code to be checked
Shorthand:	nStatus:Val Short!Error Code to be checked

Name:	sError
Parameter:	None
Type:	String
Fixed Size:	256
Shorthand:	sError:String:256

19. Next, we will write code to evaluate the **nStatus** parameter that is passed in and report the error to the user if an issue is found. Traditionally, a function will return a 0 if no error has occurred, a positive number for a warning, and a negative number for an error. Our code below will check only for errors by acting only when nStatus is less than 0. If an error is found, we retrieve the error message using the **GtDemoGetErrorString** API and then use the error statement to prompt the user. Add the following code to the code section of the CheckError procedure:

```
If nStatus < 0 Then
   GtDemoGetErrorString(nStatus, sError, 256)
   Error nStatus, sError
EndIf
```

An initialization procedure initializes a software driver for a specified instrument and returns a handle that can be used in subsequent calls to the software driver.

▼ Write the Initialization code

20. **Right-click** the CheckError procedure in the driver's procedure sub-module and select **Insert Procedure After**.
21. Change the **Name** of the new Procedure to **Initialize.** Change the **Desc** to **Initialize the driver for the board**.
22. Add the following variables to the Initialize procedure:

Name:	sSerial
Parameter:	[Val]
Optional:	Checked
Type:	String
Default Value:	“”
Desc:	Serial Number. If empty, taken from the driver config

    Shorthand: sSerial:[Val]String=””!Serial number. If empty, taken from driver config

Name:	nStatus
Parameter:	None
Type:	Short

    Shorthand: nStatus:Short

23. Next we add code to perform various checks. Since the **sSerial** can be passed in as a parameter, we don't want to immediately use the Driver parameter as the value. Because of this, we need to check to see if **nSerial** is set to anything other than its default value. Add this code:

    ```
    If sSerial = “”
       sSerial=Driver.Parameters(“SerialNumber”)
    EndIf
    If sSerial = “”
       Error -1, “Driver Parameter ‘SerialNumber’ not set to the proper
       number”
    EndIf
    ```

24. If our program executes past this code without error, then we know that we have some non-zero value entered for the **sSerial** variable. Next, we try to call the initialize API that was imported with the DLL. If it is successful, we will assign the handle back to the driver's module variable, **m_nHandle**. We will also check for errors. Add this code into the Initialize procedure after the existing code:

    ```
    GtDemoInitialize(sSerial, m_dwHandle, nStatus)
    CheckError(nStatus)
    ```

We will finish by writing three more procedures that exercise functions of the relay board.

▼ Write Functional Code

25. Create another procedure within your driver module and call it **Switch**. Add the description **Switch the state of the relay**.

26. Add the following variables to the Switch procedure:

Name: nRelay
Parameter: Val
Type: Short
Desc: Relay number. 1 for K1, 2 for K2
Shorthand: nRelay:Val Short!Relay number. 1 for K1, 2 for K2

Name: nRelayState
Parameter: Val
Type: Short
Desc: Relay state. 0 for Open, 1 for Closed
Shorthand: nRelayState:Val Short!Relay state. 0 for Open, 1 for Closed

Name: nStatus
Parameter: None
Type: Short
Shorthand: nStatus:Short

27. Add this code:

```
GtDemoRelaySwitch(m_dwHandle, nRelay, nRelayState, nStatus)
CheckError(nStatus)
```

28. Create another procedure within your driver module and call it **GetState**. Add the description **Gets the state of the selected relay**.

29. Add the following variables to the GetState() procedure:

Name: nRelay
Parameter: Val
Type: Short
Desc: Relay number. 1 for K1, 2 for K2
Shorthand: nRelay:Val Short!Relay number. 1 for K1, 2 for K2

Name: plRelayState
Parameter: Var
Type: Long
Desc: Returned relay state. 0 for Open, 1 for Closed
Shorthand: plRelayState:Var Long!Returned relay state. 0 for Open, 1 for Closed

Name:	nStatus
Parameter:	None
Type:	Short
Shorthand:	nStatus:Short

30. Add this code:

```
GtDemoRelayGetState(m_dwHandle, nRelay, plRelayState,
nStatus)
CheckError(nStatus)
```

31. Create a final procedure within your driver module and call it _**Reset**. The underscore character at the beginning is necessary since "Reset" is a reserved keyword. Add the description **Resets the GTDEMO board**.

32. Add the following variables to the _Reset() procedure:

Name:	nStatus
Parameter:	None
Type:	Short
Shorthand:	nStatus:Short

33. Add this code:

```
GtDemoReset(m_dwHandle, nStatus)
CheckError(nStatus)
```

We will attach all of the procedures to commands except our helper function, **CheckError**. First, create the command tree with four simple nodes that branch from the root node and then attach the proper procedures.

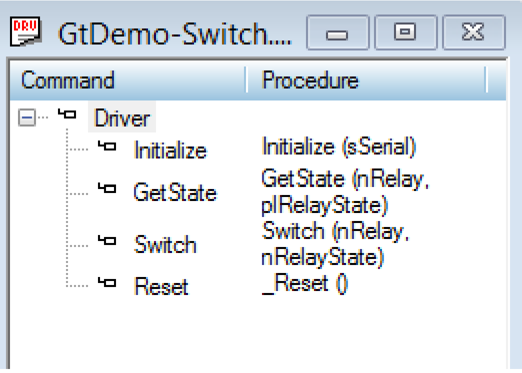
▼ Create driver commands for the procedures

34. **Double-click** the Commands sub-module to open the **Command** object in the MDI client area.

35. **Right-click** the Driver node and select **Insert Command Below**. Name the new command Initialize.

36. Use **Attach Procedure** to attach **Initialize()** to the Initialize node.

37. Repeat steps 33-35 to create a **GetState**, **Switch**, and **Reset** node.



GtDemo-Switch....

Command	Procedure
Driver	
Initialize	Initialize (sSerial)
GetState	GetState (nRelay, plRelayState)
Switch	Switch (nRelay, nRelayState)
Reset	_Reset ()

The final step courtesy that we will add is the code which will actually make the board initialize automatically. This is done in the events module.

▼ Add automatic initialization

38. **Double-click** the Events sub-module to open the Events object in the MDI client area.

39. **Select** the **OnInit** event from the events drop-down list.

40. In the code editor for the **OnInit** event, add the following:

```
Driver Initialize()
```

This concludes the creation of the new driver. The end user of your driver only needs to enter the serial number in the Driver Shortcut's miscellaneous parameters and insert commands that we created.

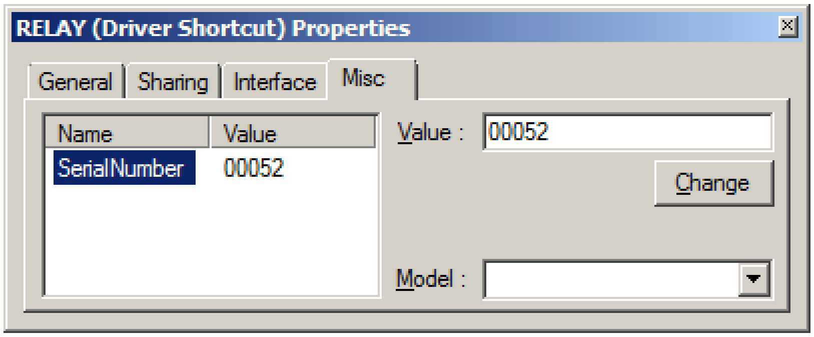
▼ Test your new driver

41. Open the **Properties Editor** for the new driver, currently named **RELAY**.

42. Navigate to the **Misc** tab.

43. Click on the **SerialNumber** parameter and enter the serial number located on the back on your GTDEMO board in the **Value** field.

44. Click **Change**. The serial number value will be added next to the **SerialNumber** parameter.



Name	Value
SerialNumber	00052

Value : 00052

45. In the Workspace Navigator, renamed the default Untitled Task to **Get/Set Relay State**. Rename the Untitled Test to **Relay K1 Open**.

46. Open the **Properties Editor** for the test and change the **Test Type** to **Precise**, and the **Value** to **0**. As stated in the procedure comments, the **GetState** procedure will assign a 0 when the relay is open and a 1 when the relay is closed. Since we are opening the relay, we are expecting a 0.

47. In the code editor for the **Relay K1 Open** test, add the following:

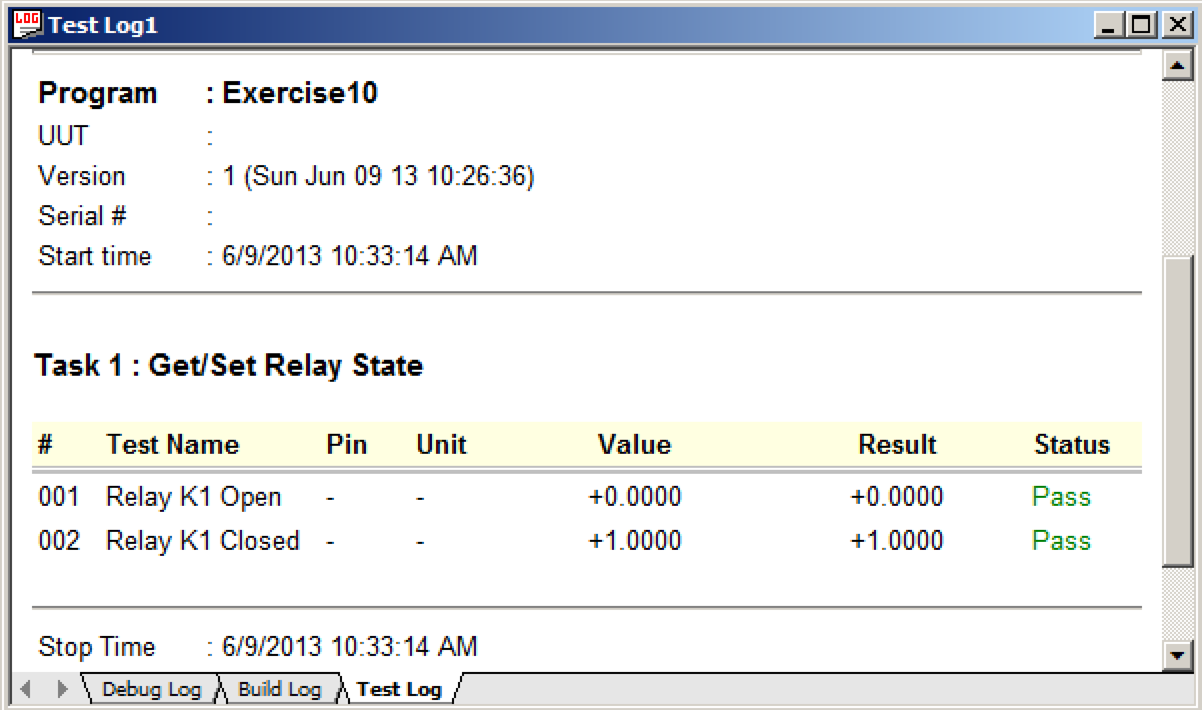
```
RELAY Switch(1, 0)
RELAY GetState(1, TestResult)
```

48. Add another test after **Relay K1 Open** and name it **Relay K1 Close**.

49. Open the **Properties Editor** for the test and change the **Test Type** to **Precise**, and the **Value** to **1.**

50. In the code editor for the **Relay K1 Open** test, add the following:

```
RELAY Switch(1, 1)
RELAY GetState(1, TestResult)
```



Program : Exercise10
UUT :
Version : 1 (Sun Jun 09 13 10:26:36)
Serial # :
Start time : 6/9/2013 10:33:14 AM

Task 1 : Get/Set Relay State

#	Test Name	Pin	Unit	Value	Result	Status
001	Relay K1 Open	-	-	+0.0000	+0.0000	Pass
002	Relay K1 Closed	-	-	+1.0000	+1.0000	Pass

Stop Time : 6/9/2013 10:33:14 AM

▼ Optional extra exercise

Switch is a bit of vague command word. Open and Close would make more sense in the context of the instrument and since parameters for commands can be hard-coded, replace the Switch command with an Open and Close command that take a single parameter each.

Additionally, we have been using basic data types to denote the target relay. To increase readability, this parameter can be replaced with an enumeration. Create an enum data type in the Types submodule and change the nRelay parameter in Switch( ) and GetState( ) to use your newly create enum.

Summary

This exercise demonstrated not only the process of importing in an external DLL library, but also some of the techniques and features of ATEasy that can be used to simplify top-level code generation. As a review, these features were automatic initialization, automatic error checking, and module-level handling of the device handle and error status. If you want to further simplify the driver, you can replace the lRelay and lRelayState parameters with enumerations, so the user can enter user-readable values instead of ambiguous numbers.

Exercise 11 – Form Creation and Use

Objectives

Create a new form.

Add controls to the event.

Write events to show the ideal waveforms that are being programmed into the DAC's burst memory.

Optionally, write events to automatically digitize the DAC-generated waveforms.

In this exercise, we will first create a new form. We will add several controls such as AButton and AChart to the form. Optionally, we will program a timer event to automatically digitize and update the charts. This project builds on the procedures that were created in Exercise 7.

▼ Open the Previously Created Example 7

1. From the file menu, select **File | Startup** to open the Startup dialog. Click on the **Recent** tab to view recently opened projects. If **Example7** was opened on this computer, it will be listed in the Recent Workspaces list. **Double-click Example7.wsp**.
2. Add an instance of the **GTDEMO** driver to the project.
3. Configure the Interface properties of the **GTDEMO** board to use the appropriate interface: **USBTMC** or **Geotest GT98901 Educational Demo Board** or **USB Test and Measurement**.

▼ Create a New Form

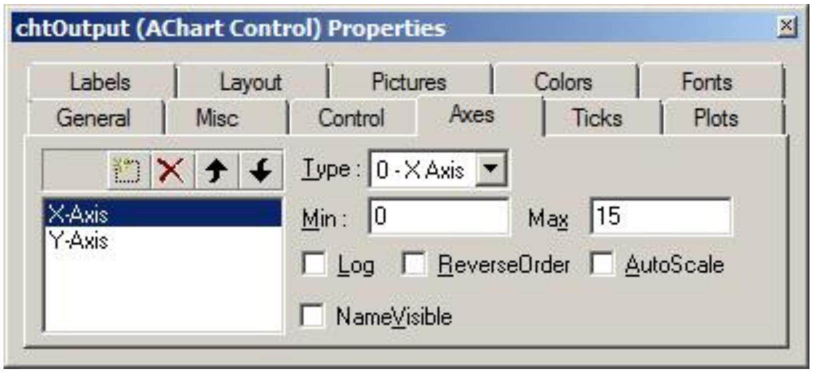
4. **Right-click** the Forms sub-module of the program module in the Workspace window and select **Insert Form Below**. A new form class name Form1 will be created.
5. Using the Properties Editor, rename the new form class to **DigitizerExample**.
6. Change the **ClientWidth** to **500**, and change the **ClientHeight** to **500**.
7. Change the **Caption** to **My Digitizer Example**.

When you have the form editor in focus, a controls toolbar should appear in the ATEasy environment to allow you to add common controls such as buttons, labels and textboxes. If this form does not automatically appear, it may be because it is currently hidden. Open it by select **Tools | Customize** from the menu bar. In the customize dialog, **click** on the **Toolbars tab** and make sure that **Controls** is enabled.

▼ Add Controls to the Form

8. **Select** the **AChart** control from the controls toolbar. After a control is selected, it can be added to a form by clicking and dragging on the form. **Add an AChart control** to the form by clicking and dragging on the form.
9. Once the control is placed, the pointer will return to selection mode . In selection mode, you can move, resize and select controls. Use this tool now to **move the new chart** to **the top-left corner** of the form.
10. This newly created chart will be used to display the waveform that is being loaded to the arbitrary waveform generator. Open the properties editor of the newly created chart control, currently named **cht1**.
11. In the **General** tab of the **cht1** control, change the **Name** to **chtOutput.**
12. In the **Control** tab of the **chtOutput** control, change the **Caption** to **AWG Output**.

13. In the **Axes** tab of **chtOutput**, select the **X-Axis** and change the range from **0 to 15**.



chtOutput (AChart Control) Properties

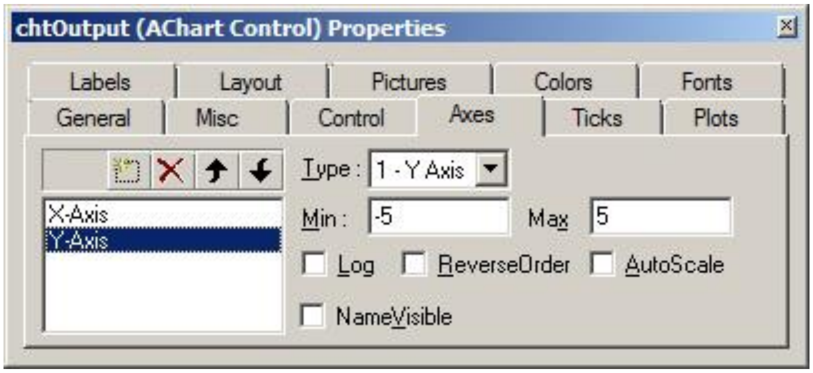
Type : 0 - X Axis

Min : 0 Max 15

□ Log □ ReverseOrder □ AutoScale

□ NameVisible

14. Then select the Y-Axis and change the range to include **-5 to 5**.



chtOutput (AChart Control) Properties

Type : 1 - Y Axis

Min : -5 Max 5

□ Log □ ReverseOrder □ AutoScale

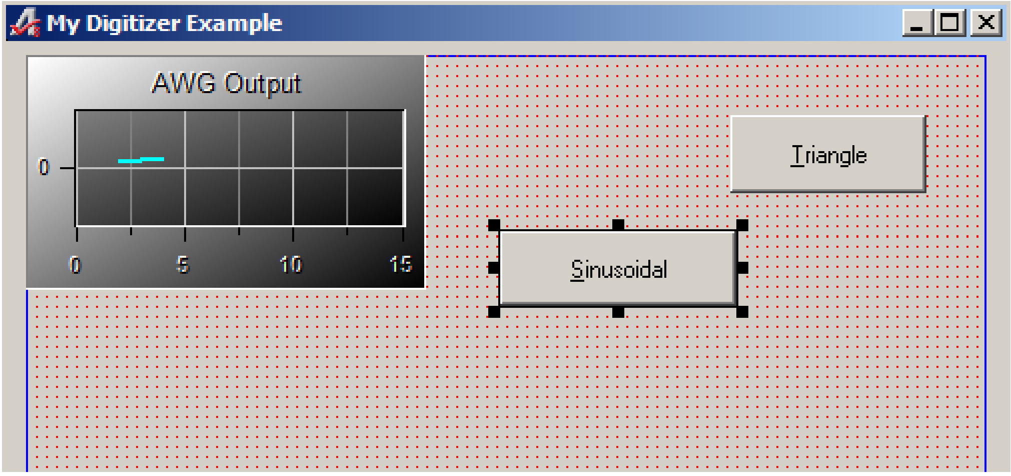
□ NameVisible

15. **Select** the **AButton** control from the controls toolbar and add it to the form to the right of **chtOutput**. The new button will be named **btn1** by default.

16. Change the name of btn1 to **btnTriangle**.

17. In the control tab of btnTriangle's properties, change the **Caption** to **&Triangle**. Adding an ampersand before the T in Triangle will make T an access key. This means that when you press Alt+T, the triangle button's **OnClick()** event will be executed.

18. Add another button below **btnTriangle**. Name this button **btnSine** and make the caption **&Sinusoidal**.



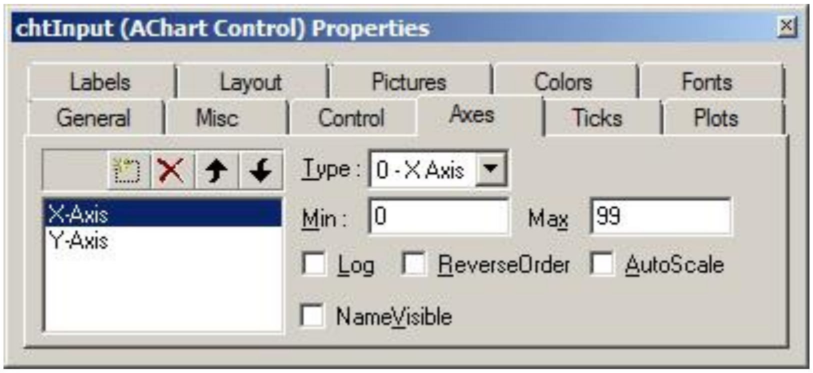
The chart and two buttons that we just finished creating will be used to select the output of the AWG and show the form user a visual representation of the currently selected output waveform. These controls should reside in the top-half of the form. The next controls to be created will be used for acquiring a waveform from the analog input ports.

19. **Select** the **AChart** control from the controls toolbar. **Add** a new **AChart control** in the bottom-left corner of the form.

20. This chart will be used to display the waveform that is being sampled by the analog input port. Change the **Name** to **chtInput**.

21. In the **Control** tab of the **chtInput** control, change the **Caption** to **ADC Input**.

22. In the **Axes** tab of **chtInput**, select the **X-Axis** and change the range to cover **0 to 99**.



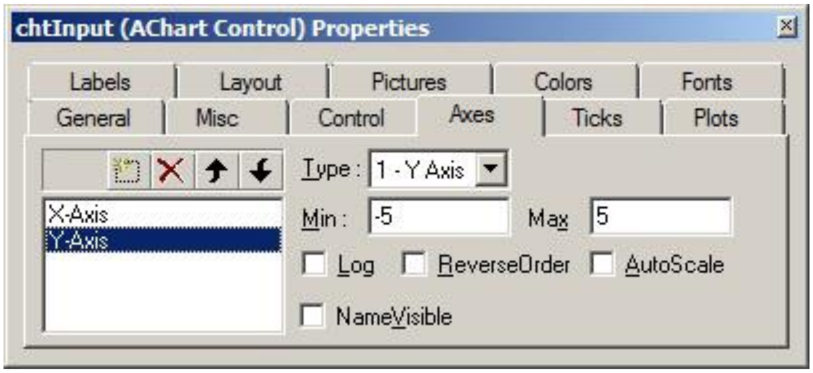
Type : 0 - X Axis

Min : 0 Max 99

□ Log □ ReverseOrder □ AutoScale

□ NameVisible

23. Select the Y-Axis and change the range to include **-5 to 5**.



chtInput (AChart Control) Properties

Labels | Layout | Pictures | Colors | Fonts

General | Misc | Control | Axes | Ticks | Plots

X-Axis

Y-Axis

Type : 1 - Y Axis

Min : -5 Max 5

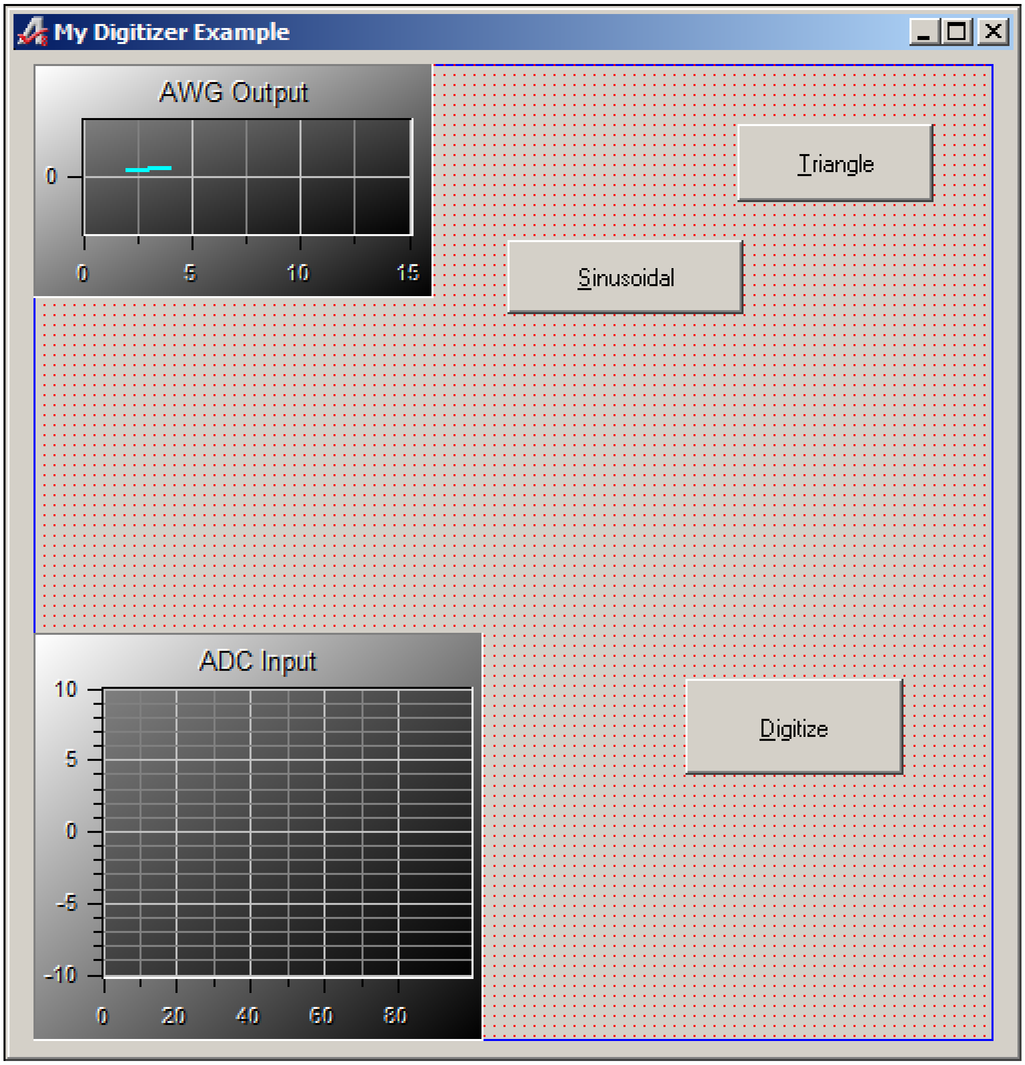
□ Log □ ReverseOrder □ AutoScale

□ NameVisible

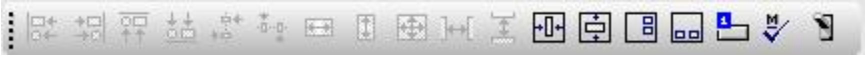
24. **Select** the **AButton** control from the controls toolbar and add a new button to the right of chtInput.

25. Change the name of the new button to **btnDigitize**.

26. Change the caption of btnDigitize to **&Digitize**.



So far, the only tools used for the positioning and resizing of tools has been the selection tool. In addition to the ability to resize and move controls. There is a toolbar that appears when we are working with the form called the Form Design toolbar.

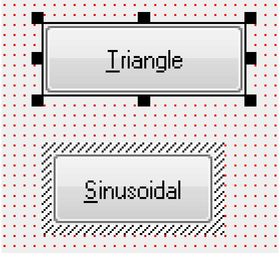


The tools that are supplied with the Form Design toolbar also us to align, resize, and otherwise improve the user experience of your form. In this section, we will explore the use of some of these tools.

▼ Reposition the controls

27. We will start by resizing and aligning the **Triangle** and **Sinusoidal** buttons to the top-right corner of the form. First, use the **Selection Tool** to make the Triangle button an appropriate size.

28. **Click** on the **Sinusoidal** to select it. Then **Ctrl+Click** on the **Triangle** button. This will make Triangle the selected button and Sinusoidal will be anchored to it.



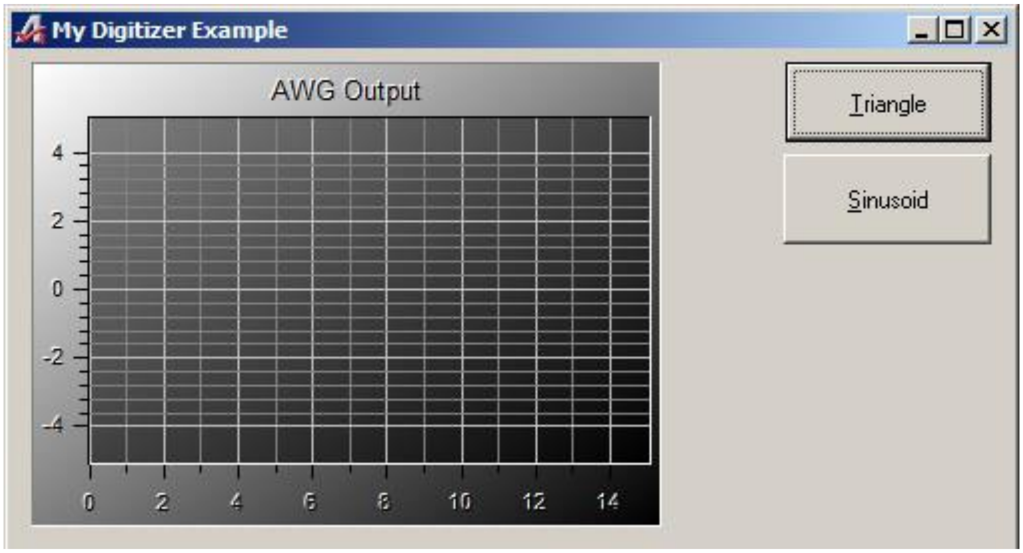
29. With Sinusoidal anchored to Triangle, use the **Same Height and Width** tool from the Form Design toolbar. This will make the anchor button the same size as the triangle button.

30. With Sinusoidal still anchored to Triangle, use the **Arrange Top Right** tool from the From Design toolbar. This will place the two buttons in the top-right corner of the form.

31. Use the selection tool to **stretch the width** of the **AWG Output chart** so that it takes up the remaining room of the form, which should be approximately 350 pixels.

32. **Stretch the height** of the **AWG Output chart** so that it takes up less than half of the height of the form, which should be approximately 200 pixels.

At this point, the top-half of your form should look similar to this:

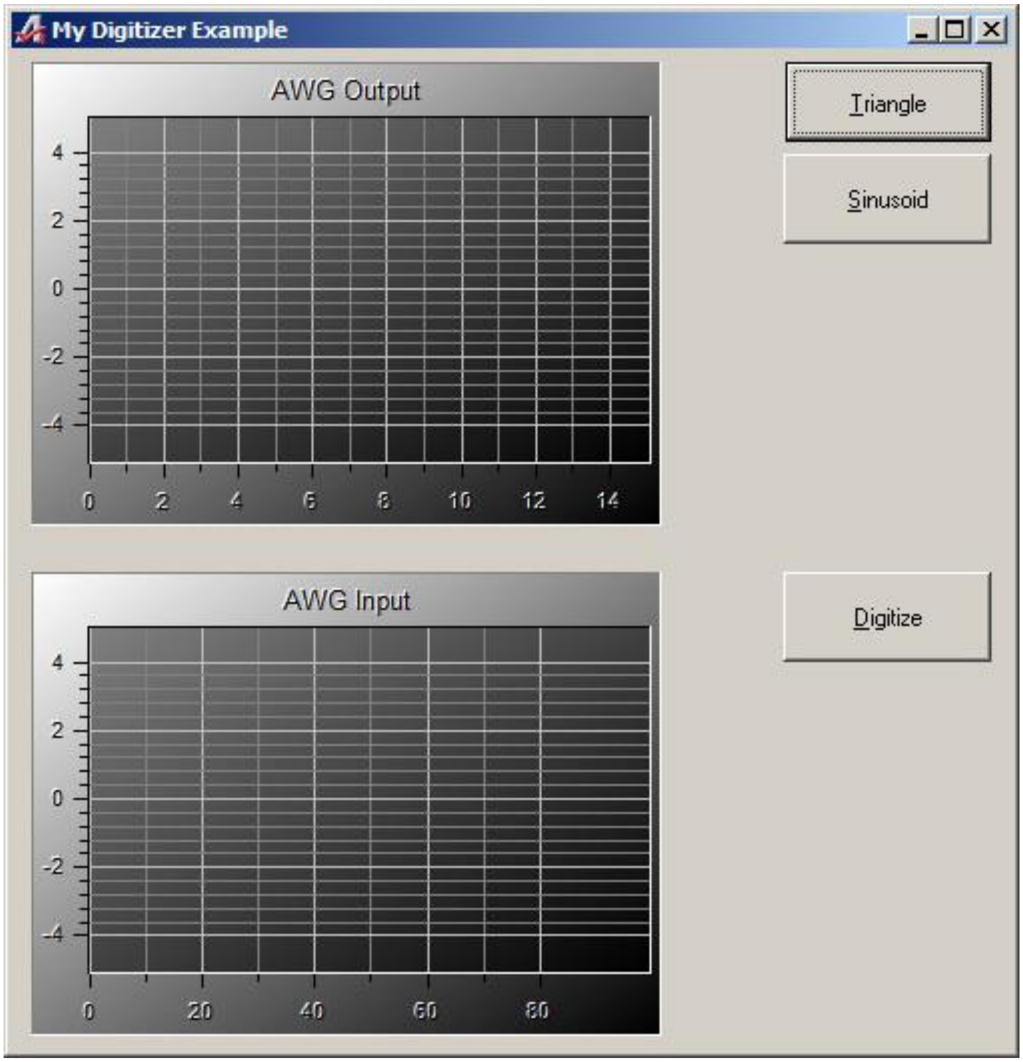


33. **Click** on the **Digitize** button and then **Ctrl+Click** the Sinusoidal button. This will anchor the Digitize button to the Sinusoidal button. First, **click** the **Same Height and Width** option from Form Design toolbar.

34. Then, with **Digitize** still anchored to **Sinusoidal**, select the **Align Right** tool from the Form Design to align the Digitize button to the rest of the buttons.

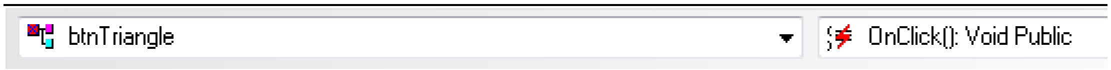
35. **Click** on the **Input** chart and then **Ctrl+Click** the Output chart. Then **click** the **Same Height and Width** tool from the Form Design toolbar.

36. This concludes our user experience improvement. Use **Test Form!** from the Form Design toolbar to view your form in another window.



▼ Write Control Events

37. Double-Click the **Triangle** button. This will open the control's **OnClick()** event in the code editor below the Form layout view. When you do this, the code editor's drop-down menu should appear as follows:



38. Add the following code to the **OnClick()** event:

```
! Create the triangle waveform array
redim afSamples[16]
CreateTriangleWave(5, afSamples, 16)
! Update the output chart to show the waveform
chtOutput.SetData(,afSamples,,,,True)
! Write the waveform to the GTDEMO board, AOut1
GTDEMO Analog Output Set ClockDivider(10)
GTDEMO Analog Output Write Array(aAnalogOutputChannel1,
   afSamples, 16)
```

39. Repeat steps 35-36 to add the following code to the btnSine.OnClick() event:

```
   ! Create the sinusoidal waveform array
redim afSamples[16]
CreateSineWave(5, afSamples, 16)
! Update the output chart to show the waveform
chtOutput.SetData(,afSamples,,,,True)
! Write the waveform to the GTDEMO board, AOut1
GTDEMO Analog Output Set ClockDivider(10)
GTDEMO Analog Output Write Array(aAnalogOutputChannel1,
   afSamples, 16)
```

40. Add the following code to the btnDigitize.OnClick() event:

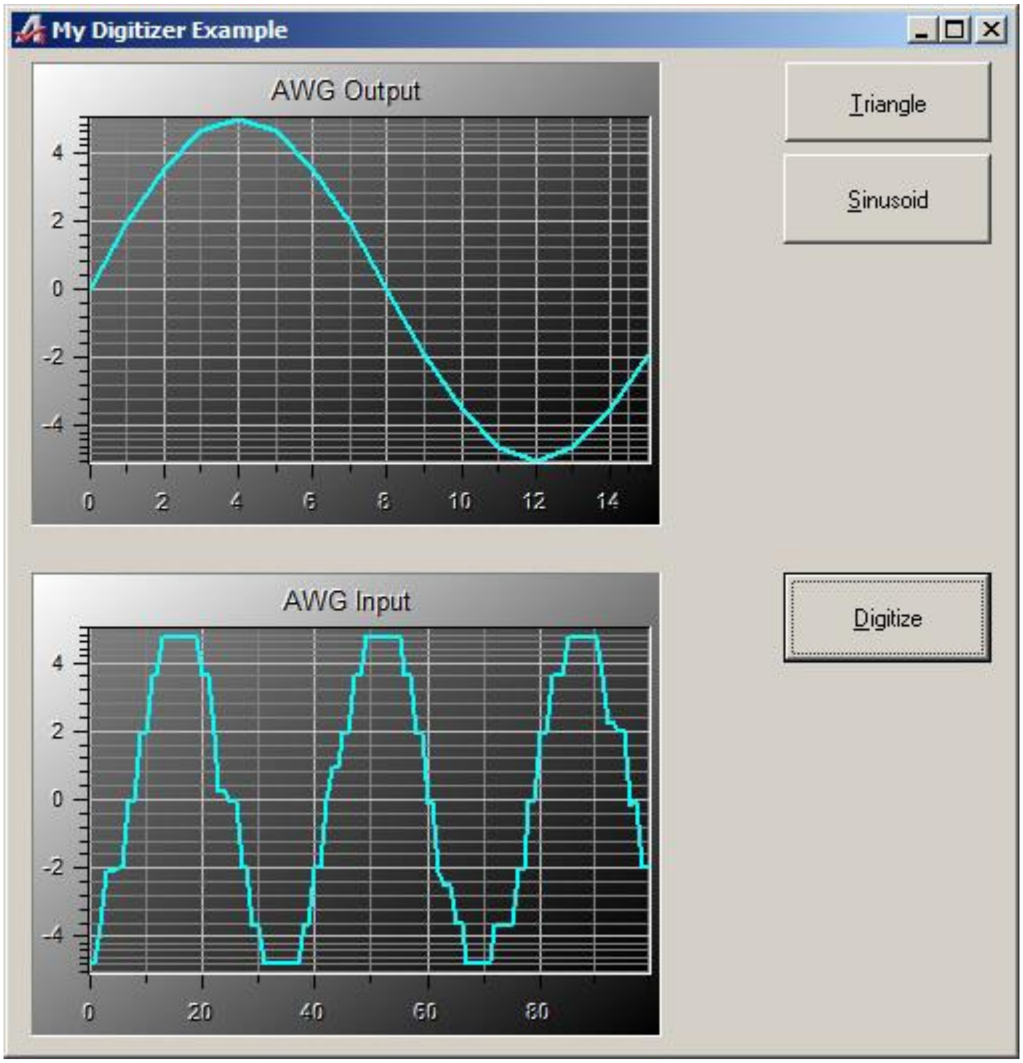
```
! Resize the array variable to hold the samples
redim afSamples[100]
! Digitize the signal on AIn1
GTDEMO Analog Input Read Array(aAnalogInputChannel1, afSamples, 100)
! Update the Input chart
chtInput.SetData(,afSamples,,,,True)
```

At this point, you can a complete working form that allows the user to select a waveform to output and also to digitize a waveform. Before we load the form in our program module, let's test it.

▼ Test Your Form Using FormIt!

41. Before running this test, make sure AOut1 is wired to AIn1.

42. Select your form by clicking on it. Then use the menu bar to select **Debug | FormIt!** A form will launch. Try clicking the **Triangle, Sinusoidal, and Digitize** buttons and observing the results. You should see results similar to the following, where the signal generated above can be read in using the **Digitize** button below:



▼ Test your form by calling it from within your Tests sub-module

43. In the Workspace Navigator, expand your program module's Tests sub-module. **Right-click** the Procedures Tests entry, and select **Insert Task After**.

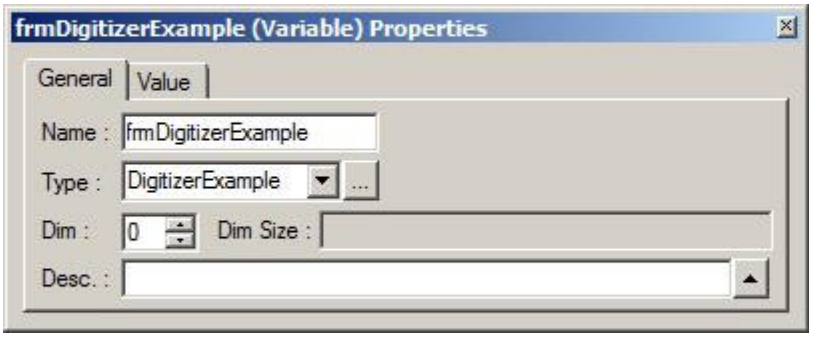
44. **Rename** the new Untitled Task to **Form Tests**.

45. **Rename** the new Untitled Test to **Load DigitizerExample**.

46. **Insert a new variable** into the Program module. This new variable will be an instance of the form class called **DigitizerExample** that we just finished creating.

47. Use the properties editor to change the **Name** to **frmDigitizerExample**

48. Change the **Type** to **DigitizerExample**.


frmDigitizerExample (Variable) Properties

Name : frmDigitizerExample

Type : DigitizerExample

Dim : 0

Dim Size :

Desc. :

49. Add the following code to your **Load DigitizerExample** test in the Tests sub-module:

```
Load frmDigitizerExample, True
```

Now, when the test is run, it will load your DigitizerExample form modally. Since the form is modal, your test application will not continue until the form is closed.

▼ Optional extra exercise

The form that we have created should be functional at this stage, but it could be improved. You can add controls and events to give the form user the option to automatically acquire waveforms rather than having to click the Digitize button. You could also look at the first twenty elements of the digitize data to trigger on the same voltage. This would keep your waveform fixed and prevent it from jumping around.

Summary

This exercise improved upon our procedure exercise by adding and introduced the process of designing a form, writing control event code, and testing your completed form.

Appendix A – Function Reference

Introduction

The functions reference chapter organizes the list of GTDEMO driver functions in an alphabetical order. Each function description contains the function name; purpose, syntax, parameters description and type followed by Comments, an Example (written in C), and a See Also sections.

All function and parameter syntax follow the same rules:

Strings are ASCIIZ (null or zero character terminated).

The first parameter of most functions is *dwHandle* (32-bit integer). This parameter is required for operating the board and is returned by the **GtDemoInitialize** function. The *dwHandle* is used to identify the board when calling a function for programming and controlling the operation of that board.

All functions return a status with the last parameter named *pnStatus*. The *pnStatus* is zero if the function was successful, or non-zero on error. The description of the error is available using the **GtDemoGetErrorString** function or by using a predefined constant, defined in the driver interface files: GTDEMO.H

Parameter name are prefixed as follows:

Prefix	Type	Example
a	Array - prefix this before the simple type.	*anArray* (Array of Short)
b	BOOL – Boolean, 0 for FALSE; <>0 for TRUE	*bUpdate*
d	DOUBLE - 8 bytes floating point	*dReading*
dw	DWORD - double word (unsigned 32-bit)	*dwTimeout*
hwnd	Window handle (32-bit integer).	*hwndPanel*
l	LONG - (signed 32-bit)	*lBits*
n	SHORT - (signed 16-bit)	*nMode*
p	Pointer - Usually used to return a value. Prefix this before the simple type.	*pnStatus*
sz	Null - (zero value character) terminated string	*szMsg*
uc	BYTE - (8 bits) unsigned.	*ucValue*
w	WORD - Unsigned short (unsigned 16-bit)	*wParam*

Table 0-1: Parameter Name Prefixes

GTDEMO Functions

The following list is a summary of functions available for the GT98901:

Driver Functions	Description
GtDemoAnalogInputReadArray	Reads measurements from the analog input buffer.
GtDemoAnalogInputReadSingle	Triggers a measurement from the specified analog input channel.
GtDemoAnalogOutputAbort	Disables output of the analog burst signal.
GtDemoAnalogOutputGetClockDivider	Returns the analog ouput clock's divider.
GtDemoAnalogOutputGetRunning	Determines whether the analog output is currently running.
GtDemoAnalogOutputReadArray	Reads the array of voltages currently programmed to the analog output burst channel.
GtDemoAnalogOutputSetClockDivider	Sets the analog output clock's divider.
GtDemoAnalogOutputTrigger	Begin generating analog burst output.
GtDemoAnalogOutputWriteArray	Programs an array of voltages to the specified analog output burst channel.
GtDemoAnalogOutputWriteSingle	Sets a specified static voltage for an analog output channel.
GtDemoBuzzerGetState	Returns the state of the buzzer.
GtDemoBuzzerSetState	Sets the state of the buzzer.
GtDemoClose	Closes communications with the demo board.
GtDemoDigitalGetDirection	Returns the direction of the digital IO channels.
GtDemoDigitalPWMGetDivider	Returns the divider for the 120MHz pulse width modulator's clock.
GtDemoDigitalPWMGetDuty	Returns the duty cycle percentage of the pulse-width modulation.
GtDemoDigitalPWMGetPeriod	Returns the period for the pulse-width modulation.
GtDemoDigitalPWMGetState	Return the state of the pulse-width modulation.
GtDemoDigitalPWMSetDivider	Sets the divider for the 120MHz pulse width modulator's clock.
GtDemoDigitalPWMSetDuty	Sets the duty cycle percentage of the pulse-width modulation.
GtDemoDigitalPWMSetPeriod	Sets the period for the pulse-width modulation.
GtDemoDigitalPWMSetState	Sets the state of the pulse-width modulation.
GtDemoDigitalReadData	Reads the values of the digital IO's input channels.
GtDemoDigitalSetDirection	Sets the direction of the digital I/O channels.
GtDemoDigitalWriteData	Sets the value of the digital I/O's output channels.
GtDemoDipSwitchGetState	Returns the state of the specified dip switch.
GtDemoDisplayClear	Clears the LCD display.
GtDemoDisplayGetLineNumber	Returns the current line number of the LCD display.
GtDemoDisplaySetLineNumber	Sets the current line number of the LCD display.
GtDemoDisplaySetText	Sets the new text to the LCD's current line.

Driver Functions	Description
GtDemoGetErrorString	Returns the error string associated with the specified error number.
GtDemoInitialize	Initializes the driver for the demo board.
GtDemoLedSetState	Sets the state of the specified LED.
GtDemoReferenceGetDivider	Returns the divider of the reference clock.
GtDemoReferenceSetDivider	Sets the divider of the reference clock.
GtDemoRelayGetState	Returns the state of the specified SPDT relay.
GtDemoRelaySwitch	Sets the state of the specified SPDT relay.
GtDemoReset	Resets the board to its default settings.
GtDemoPanel	Opens a virtual panel used to interactively control the demo board.
GtDemoPushButtonGetState	Returns the push button flag.
GtDemoSystemClearEventStatus	Clear the event status register.
GtDemoSystemGetEventStatusEnable	Returns the Event Status Enable register.
GtDemoSystemGetEventStatusRegister	Returns the Event Status register.
GtDemoSystemGetIdentification	Returns the board’s description and firmware information.
GtDemoSystemGetOperationCompleteBit	Returns the Operation Complete bit.
GtDemoSystemGetServiceRequestEnable	Returns the Service Request Enable register.
GtDemoSystemSelfTest	Runs the demo board self-test and returns the board status.
GtDemoSystemSetEventStatusEnable	Sets the Event Status Enable register.
GtDemoSystemSetOperationCompleteBit	Sets the Operation Complete bit.
GtDemoSystemSetServiceRequestEnable	Sets the Service Request Enable register.

Function to ATEasy Command Reference

The following list shows the relation between GtDemo functions and the associated GtDemo.drv ATEasy driver commands:

GtDemo Functions	Associated ATEasy driver command
GtDemoAnalogInputReadArray	GTDEMO Analog Input Read Array
GtDemoAnalogInputReadSingle	GTDEMO Analog Input Read Single
GtDemoAnalogOutputAbort	GTDEMO Analog Output Abort
GtDemoAnalogOutputGetClockDivider	GTDEMO Analog Output Get ClockDivider
GtDemoAnalogOutputGetRunning	GTDEMO Analog Output Get Running
GtDemoAnalogOutputReadArray	GTDEMO Analog Output ReadArray
GtDemoAnalogOutputSetClockDivider	GTDEMO Analog Output Set ClockDivider
GtDemoAnalogOutputTrigger	GTDEMO Analog Output Trigger
GtDemoAnalogOutputWriteArray	GTDEMO Analog Output Write Array
GtDemoAnalogOutputWriteSingle	GTDEMO Analog Output Write Single
GtDemoBuzzerGetState	GTDEMO Buzzer Get State
GtDemoBuzzerSetState	GTDEMO Buzzer Set Stated
GtDemoDigitalGetDirection	GTDEMO Digital Get Direction
GtDemoDigitalPWMGetDivider	GTDEMO Digital PWM Get Divider
GtDemoDigitalPWMGetDuty	GTDEMO Digital PWM Get Duty
GtDemoDigitalPWMGetPeriod	GTDEMO Digital PWM Get Period
GtDemoDigitalPWMGetState	GTDEMO Digital PWM Get State
GtDemoDigitalPWMSetDivider	GTDEMO Digital PWM Set Divider
GtDemoDigitalPWMSetDuty	GTDEMO Digital PWM Set Duty
GtDemoDigitalPWMSetPeriod	GTDEMO Digital PWM Set Period
GtDemoDigitalPWMSetState	GTDEMO Digital PWM Set State
GtDemoDigitalReadData	GTDEMO Digital Read Data
GtDemoDigitalSetDirection	GTDEMO Digital Set Direction
GtDemoDigitalWriteData	GTDEMO Digital Write Data
GtDemoDipSwitchGetState	GTDEMO DipSwitch Get State
GtDemoDisplayClear	GTDEMO Display Clear
GtDemoDisplayGetLineNumber	GTDEMO Display Get LineNumber
GtDemoDisplaySetLineNumber	GTDEMO Display Set LineNumber
GtDemoDisplaySetText	GTDEMO Display Set Text
GtDemoGetErrorString	GTDEMO CheckError
GtDemoInitialize	GTDEMO Initialize

GtDemo Functions	Associated ATEasy driver command
GtDemoLedSetState	GTDEMO Led Set State Off GTDEMO Led Set State On
GtDemoReferenceGetDivider	GTDEMO Reference Get Divider
GtDemoReferenceSetDivider	GTDEMO Reference Set Divider
GtDemoRelayGetState	GTDEMO Relay Get State
GtDemoRelaySwitch	GTDEMO Relay Switch Open GTDEMO Relay Switch Close
GtDemoReset	GTDEMO System Reset
GtDemoPanel	GTDEMO Panel
GtDemoPushButtonGetState	GTDEMO PushButton Get State
GtDemoSystemClearEventStatus	GTDEMO System Clear EventStatus
GtDemoSystemGetEventStatusEnable	GTDEMO System Get EventStatusEnable
GtDemoSystemGetEventStatusRegister	GTDEMO System Get EventStatusRegister
GtDemoSystemGetIdentification	GTDEMO System Get Identification
GtDemoSystemGetOperationCompleteBit	GTDEMO System Get OperationCompleteBit
GtDemoSystemGetServiceRequestEnable	GTDEMO System Get ServiceRequestEnable
GtDemoSystemSelfTest	GTDEMO System SelfTest
GtDemoSystemSetEventStatusEnable	GTDEMO System Set EventStatusEnable
GtDemoSystemSetOperationCompleteBit	GTDEMO System Set OperationCompleteBit
GtDemoSystemSetServiceRequestEnable	GTDEMO System Set ServiceRequestEnable

GtDemoAnalogInputReadArray

Purpose

Begins sampling and storing measurements to an array from the specified analog input channel.

Syntax

GtDemoAnalogInputReadArray (*dwHandle, enAnalogInputChannel, pafMeasurements, dwDataElements, pnStatus*)

Parameters

Name	**Type**	**Comments**
dwHandle	DWORD	Handle to a demo board.
enAnalogInputChannel	LONG	Specifies the analog input channel to measure: 1 = GTDEMO_AI_CHANNEL1 2 = GTDEMO_AI_CHANNEL2 3 = GTDEMO_AI_CHANNEL3 4 = GTDEMO_AI_CHANNEL4
pafMeasurements	PFLOAT	Returned array of voltages measurements.
dwDataElements	DWORD	Number of elements to sample, max 128
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Comments

This function starts the specified analog input channel's digitizing feature. It sets the number of samples based on the *dwDataElements* parameter and triggers measurement. When the measurement is completed, this function will return the filled array.

Example

The following example makes 64 measurements on analog input channel 2 and stores the results to an array:

```
DWORD dwHandle;
SHORT nStatus;
FLOAT *afMeasurements;

GtDemoAnalogInputReadArray (dwHandle, GTDEMO_AI_CHANNEL2, &afMeasurements,
64, &nStatus);
```

See Also

GtDemoGetErrorString

GtDemoAnalogInputReadSingle

Purpose

Triggers a measurement from the specified analog input channel.

Syntax

GtDemoAnalogInputReadSingle (*dwHandle*, *enAnalogInputChannel*, *pdMeasurement*, *pnStatus*)

Parameters

Name	**Type**	**Comments**
dwHandle	DWORD	Handle to a demo board.
enAnalogInputChannel	LONG	Specifies the analog input channel to measure: 1 = GTDEMO_AI_CHANNEL1 2 = GTDEMO_AI_CHANNEL2 3 = GTDEMO_AI_CHANNEL3 4 = GTDEMO_AI_CHANNEL4
pdMeasurement	PDOUBLE	Returns the voltage measurement
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Example

The following example reads a measurement from channel 2:

```
SHORT nStatus;
DWORD dwHandle;
DOUBLE dMeasurement;

GtDemoAnalogInputReadSingle (dwHandle, GTDEMO_AI_CHANNEL2, &dMeasurement,
&nStatus);
```

See Also

GtDemoGetErrorString

GtDemoAnalogOutputAbort

Purpose

Disables output of the analog burst signal.

Syntax

GtDemoAnalogOutputAbort (*dwHandle*, *pnStatus*)

Parameters

Name	Type	Comments
dwHandle	DWORD	Handle to a demo board.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Comments

The Arm.

Example

The following example triggers and then aborts the GT98901.

```
DWORD dwHandle;
SHORT nStatus;

GtDemoAnalogOutputWriteArray(dwHandle, ..., &nStatus);
GtDemoAnalogOutputAbort (dwHandle, &nStatus);
```

See Also

GtDemoAnalogOutputWriteArray, GtDemoGetErrorString

GtDemoAnalogOutputGetClockDivider

Purpose

Returns the analog output clock’s divider.

Syntax

GtDemoAnalogOutputGetClockDivider(*dwHandle*, *plDivider*, *pnStatus*)

Parameters

Name	**Type**	**Comments**
dwHandle	DWORD	Handle to a demo board.
plDivider	PLONG	The returned analog output clock’s divider.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Example

The following example returns the analog output clock’s divider:

```
SHORT nStatus;
DWORD dwHandle;
LONG lDivider;

GtDemoAnalogOutputGetClockDivider (dwHandle, &lDivider, &nStatus);
```

See Also

GtDemoAnalogOutputSetClockDivider, **GtDemoGetErrorString**

GtDemoAnalogOutputGetRunning

Purpose

Determines whether the analog output is currently running.

Syntax

GtDemoAnalogOutputGetRunning (*dwHandle, pbRunning, pnStatus*)

Parameters

Name	Type	Comments
dwHandle	DWORD	Handle to a demo board.
pbRunning	PBOOL	TRUE if the analog output channels are currently generating a waveform, FALSE if they are not.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Example

The following example loops until the analog output stops running:

```
SHORT nStatus;
DWORD dwHandle;
BOOL bRunning;

do
{
      GtDemoAnalogOutputGetRunning (dwHandle, &bRunning, &nStatus);
      Sleep(10);
} while (bRunning);
```

See Also

GtDemoGetErrorString

GtDemoAnalogOutputReadArray

Purpose

Reads the array of voltages currently programmed to the analog output burst channel.

Syntax

GtDemoAnalogOutputReadArray (*dwHandle, enAnalogOutputChannel, pafData, dwDataElements, pnStatus*)

Parameters

Name	**Type**	**Comments**
dwHandle	DWORD	Handle to a demo board.
enAnalogOutputChannel	LONG	Specified analog output channel number: • 1 = GTDEMO_AO_CHANNEL1 • 2 = GTDEMO_AO_CHANNEL2
pafData	PFLOAT	Returns the currently programmed array of voltages.
dwDataElements	DWORD	Number of elements in the array.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Comments

Example

The following example reads the values that are currently programmed into the specified analog output channel:

```
DWORD dwHandle;
SHORT nStatus;
FLOAT afData[64];

GtDemoAnalogOutputReadArray (dwHandle, GTDEMO_AO_CHANNEL1, afData, 64,
&nStatus);
```

See Also

GtDemoAnalogOutputWriteArray, GtDemoGetErrorString

GtDemoAnalogOutputSetClockDivider

Purpose

Sets the analog output clock's divider.

Syntax

GtDemoAnalogOutputSetClockDivider (*dwHandle, lDivider, pnStatus*)

Parameters

Name	**Type**	**Comments**
dwHandle	SHORT	Handle to a demo board.
lDivider	LONG	Specifies the sample clock divider, 1 to 65535.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Comments

The DAC clock is set to 234 kHz divided by N where N is the programmed clock divider.

Example

The following example sets the analog output clock's divider to 10, effectively making the DACs clock equal to 23.4 kHz:

```
SHORT nStatus;
DWORD dwHandle;

GtDemoAnalogOutputSetClockDivider (dwHandle, 10, &nStatus);
```

See Also

GtDemoAnalogOutputGetClockDivider, **GtDemoGetErrorString**

GtDemoAnalogOutputTrigger

Purpose

Begin generating analog burst output.

Syntax

GtDemoAnalogOutputTrigger (*dwHandle, pnStatus*)

Parameters

Name	Type	Comments
dwHandle	DWORD	Handle to a demo board.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Example

The following example triggers and then aborts the GT98901.

```
DWORD dwHandle;
SHORT nStatus;

GtDemoAnalogOutputTrigger(dwHandle, &nStatus);
GtDemoAnalogOutputAbort (dwHandle, &nStatus);
```

See Also

GtDemoAnalogOutputWriteArray, **GtDemoAnalogOutputSetNumberOfScans**, **GtDemoGetErrorString**

GtDemoAnalogOutputWriteArray

Purpose

Write the specified array to the analog output burst channel.

Syntax

GtDemoAnalogOutputWriteArray (*dwHandle*, *enAnalogOutputChannel*, *afData*, *dwDataElements*, *pnStatus*)

Parameters

Name	**Type**	**Comments**
dwHandle	DWORD	Handle to a demo board.
enAnalogOutputChannel	LONG	Specified analog output channel number: • 1 = GTDEMO_AO_CHANNEL1 • 2 = GTDEMO_AO_CHANNEL2
afData	PFLOAT	Array of voltages to be programmed, each element from -10.0 to 10.0 volts.
dwDataElements	DWORD	Number of elements.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Comments

This function programs the specified number of elements to the specified analog output channel from the provided array.

Example

The following writes burst data to analog output channel 1:

```
DWORD dwHandle;
FLOAT afData[32];
SHORT nStatus;

GtDemoAnalogOutputWriteArray (dwHandle, GTDEMO_AO_CHANNEL1, afData, 32,
&nStatus);
```

See Also

GtDemoAnalogOutputTrigger, **GtDemoGetErrorString**

GtDemoAnalogOutputWriteSingle

Purpose

Sets a specified static voltage for an analog output channel.

Syntax

GtDemoAnalogOutputWriteSingle (*dwHandle*, *enAnalogOutputChannel*, *fVoltage*, *pnStatus*)

Parameters

Name	**Type**	**Comments**
dwHandle	SHORT	Handle to a demo board.
enAnalogOutputChannel	LONG	Specified analog output channel number: • 1 = GTDEMO_AO_CHANNEL1 • 2 = GTDEMO_AO_CHANNEL2
fVoltage	FLOAT	Specifies the voltage, -10.0 to 10.0 volts
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Comments

The Filter.

Example

The following example sets the voltage of AOut1 to 3.3 volts:

```
SHORT nStatus;
DWORD dwHandle;

GtDemoAnalogOutputWriteSingle (dwHandle, GTDEMO_AO_CHANNEL1, 3.3, &nStatus);
```

See Also

GtDemoGetErrorString

GtDemoBuzzerGetState

Purpose

Returns the state of the buzzer.

Syntax

GtDemoBuzzerGetState (*dwHandle*, *penBuzzerState*, *pnStatus*)

Parameters

Name	Type	Comments
dwHandle	DWORD	Handle to a demo board.
penBuzzerState	PLONG	Returned state of the buzzer number: 0 = GTDEMO_BUZZER_OFF 1 = GTDEMO_BUZZER_ON
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Example

The following example returns the state of the buzzer:

```
SHORT nStatus;
LONG lBuzzerState;
DWORD dwHandle;

GtDemoBuzzerGetState (dwHandle, &lBuzzerState, &nStatus);
```

See Also

GtDemoBuzzerSetState, **GtDemoGetErrorString**

GtDemoBuzzerSetState

Purpose

Sets the state of the buzzer.

Syntax

GtDemoBuzzerSetState (*dwHandle*, *enBuzzerState*, *pnStatus*)

Parameters

Name	**Type**	**Comments**
dwHandle	DWORD	Handle to a demo board.
enBuzzerState	LONG	State of the buzzer: 0 = GTDEMO_BUZZER_OFF 1 = GTDEMO_BUZZER_ON
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Example

The following example turns the buzzer off:

```
SHORT nStatus;
DWORD dwHandle;

GtDemoBuzzerSetState (dwHandle, GTDEMO_BUZZER_OFF, &nStatus);
```

See Also

GtDemoBuzzerGetState, **GtDemoGetErrorString**

GtDemoClose

Purpose

Closes communications with the demo board.

Syntax

GtDemoClose (*dwHandle*, *pnStatus*)

Parameters

Name	Type	Comments
dwHandle	DWORD	Handle to a demo board.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Example

The following example initializes and then closes communication with the demo board:

```
SHORT nStatus;
DWORD dwHandle;

GtDemoInitialize(“00030”, &dwHandle, &nStatus);
GtDemoClose(dwHandle, &nStatus);
```

See Also

GtDemoInitialize, **GtDemoGetErrorString**

GtDemoDigitalGetDirection

Purpose

Returns the direction of the digital IO channels.

Syntax

GtDemoDigitalGetDirection (*dwHandle*, *pwDirection*, *pnStatus*)

Parameters

Name	**Type**	**Comments**
dwHandle	DWORD	Handle to a demo board.
pwDirection	PWORD	The direction of the DIO channels. 1 is Input, 0 is Output. The 8 digital I/O channels are represented by 8 bits. Each bit represents a channel starting with channel 0, e.g. channel 0 is represented by bit 0.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

.

Example

The following example returns the direction of the digital channels:

```
SHORT nStatus;
DWORD dwHandle;
WORD wDirection;

GtDemoDigitalGetDirection (dwHandle, &wDirection, &nStatus);
```

See Also

GtDemoDigitalSetDirection, **GtDemoGetErrorString**

GtDemoDigitalPWMGetDivider

Purpose

Returns the divider for the 120MHz pulse width modulator's clock.

Syntax

GtDemoDigitalPWMGetDivider (*dwHandle*, *pnDivider*, *pnStatus*)

Parameters

Name	Type	Comments
dwHandle	DWORD	Handle to a demo board.
pnDivider	PSHORT	Returns the PWM clock divider.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Example

The following example returns the PWM's divider:

```
SHORT nStatus, nDivider;
DWORD dwHandle;

GtDemoDigitalPWMGetDivider (dwHandle, &nDivider, &nStatus)
```

See Also

GtDemoDigitalPWMSetDivider, **GtDemoGetErrorString**

GtDemoDigitalPWMGetDuty

Purpose

Returns the duty cycle percentage of the pulse-width modulation.

Syntax

GtDemoDigitalPWMGetDuty (*dwHandle*, *pnDuty*, *pnStatus*)

Parameters

Name	Type	Comments
dwHandle	DWORD	Handle to a demo board.
pnDuty	PSHORT	Returns the duty cycle percent.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Example

The following example returns the current duty cycle of the PWM:

```
SHORT nDuty, nStatus;
DWORD dwHandle;

GtDemoDigitalPWMGetDuty (dwHandle, &nDuty, &nStatus);
```

See Also

GtDemoDigitalPWMSetDuty, **GtDemoGetErrorString**

GtDemoDigitalPWMGetPeriod

Purpose

Returns the period for the pulse-width modulation.

Syntax

GtDemoDigitalPWMGetPeriod (*dwHandle*, *pwPeriod*, *pnStatus*)

Parameters

Name	Type	Comments
dwHandle	DWORD	Handle to a demo board.
pwPeriod	PWORD	Returns the period.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Example

The following example returns the period of the PWM:

```
DWORD dwHandle;
SHORT nActiveInputChannel;
WORD wPeriod;

GtDemoDigitalPWMGetPeriod (dwHandle, &wPeriod, &nStatus);
```

See Also

GtDemoDigitalPWMSetPeriod, **GtDemoGetErrorString**

GtDemoDigitalPWMGetState

Purpose

Return the state of the pulse-width modulation.

Syntax

GtDemoDigitalPWMGetState (*dwHandle, penPWMState, pnStatus*)

Parameters

Name	Type	Comments
dwHandle	DWORD	Handle to a demo board.
penPWMState	PLONG	Returns the PWM state: • 0 = GTDEMO_PWM_OFF • 1 = GTDEMO_PWM_ON
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Comments

When the state is set to GTDEMO_PWM_ON, the pulse width modulator will output on D6. When it is set to GTDEMO_PWM_OFF, D6 will operate as a digital port.

Example

The following returns the state of the pulse width modulator:

```
DWORD dwHandle;
LONG lState;
SHORT nStatus;

GtDemoDigitalPWMGetState (dwHandle, &lState, &nStatus);
```

See Also

GtDemoDigitalPWMSetState, **GtDemoGetErrorString**

GtDemoDigitalPWMSetDivider

Purpose

Sets the divider for the 120MHz pulse width modulator's clock.

Syntax

GtDemoDigitalPWMSetDivider (*dwHandle, nDivider, pnStatus*)

Parameters

Name	Type	Comments
dwHandle	DWORD	Handle to a demo board.
nDivider	SHORT	Specifies the divider for 120MHz clock is 2^nDivider: 0 to 6.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Example

The following example returns sets the PWM divider so that the clock is 30 MHz:

```
SHORT nStatus;
DWORD dwHandle;

GtDemoDigitalPWMSetDivider (dwHandle, 2, &nStatus);
```

See Also

GtDemoDigitalPWMGetDivider, **GtDemoGetErrorString**

GtDemoDigitalPWMSetDuty

Purpose

Sets the duty cycle percentage of the pulse-width modulation.

Syntax

GtDemoDigitalPWMSetDuty (*dwHandle*, *nDuty*, *pnStatus*)

Parameters

Name	**Type**	**Comments**
dwHandle	DWORD	Handle to a demo board.
nDuty	SHORT	Sets the duty cycle percent: 0 to 100.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Comments

The duty cycle percentage determines the percent of the pulsed signal that is active. A value of 10 means that the signal will be active for 10% of the signal and non-active for the remaining 90%

Example

The following example sets the duty cycle of the PWM to 50%:

```
DWORD dwHandle;
SHORT nStatus;

GtDemoDigitalPWMSetDuty (dwHandle, 50, &nStatus);
```

See Also

GtDemoDigitalPWMGetDuty, **GtDemoGetErrorString**

GtDemoDigitalPWMSetPeriod

Purpose

Sets the period for the pulse-width modulation.

Syntax

GtDemoDigitalPWMSetPeriod (*dwHandle*, *wPeriod*, *pnStatus*)

Parameters

Name	**Type**	**Comments**
dwHandle	DWORD	Handle to a demo board.
wPeriod	WORD	Specified the period: 1 to 65535.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Example

The following example sets the period of the PWM to 25 steps:

```
DWORD dwHandle;
SHORT nStatus;

GtDemoDigitalPWMSetPeriod (dwHandle, 25, &nStatus);
```

See Also

GtDemoDigitalPWMGetPeriod, **GtDemoGetErrorString**

GtDemoDigitalPWMSetState

Purpose

Sets the state of the pulse-width modulation.

Syntax

GtDemoDigitalPWMSetState (*dwHandle*, *enPWMState*, *pnStatus*)

Parameters

Name	Type	Comments
dwHandle	DWORD	Handle to a demo board.
enPWMState	LONG	Sets the PWM state: • 0 = GTDEMO_PWM_OFF • 1 = GTDEMO_PWM_ON
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Comments

When the state is set to GTDEMO_PWM_ON, the pulse width modulator will output on D6. When it is set to GTDEMO_PWM_OFF, D6 will operate as a digital port.

Example

The following example turns on pulse width modulation:

```
DWORD dwHandle;
SHORT nStatus;

GtDemoDigitalPWMSetState (dwHandle, GTDEMO_PWM_ON, &nStatus);
```

See Also

GtDemoDigitalPWMGetState, **GtDemoGetErrorString**

GtDemoDigitalReadData

Purpose

Reads the values of the digital IO's channels.

Syntax

GtDemoDigitalReadData (*dwHandle, pwValues, pnStatus*)

Parameters

Name	Type	Comments
dwHandle	DWORD	Handle to a demo board.
pwValues	PWORD	The 8 digital I/O channels are represented by 8 bits. Each bit represents a channel starting with channel 0, e.g. channel 0 is represented by bit 0.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Comments

This function reads data back from all of the digital lines. The digital lines that are currently set to input will be read and resolve to either high or low. The digital lines that are currently set to output will report back their current programmed value.

Example

The following example returns the data information of the digital ports:

```
DWORD dwHandle;
WORD wValues;
SHORT nStatus;

GtDemoDigitalReadData (dwHandle, &wValues, &nStatus);
```

See Also

GtDemoDigitalWriteData, **GtDemoGetErrorString**

GtDemoDigitalSetDirection

Purpose

Sets the direction of the digital I/O channels.

Syntax

GtDemoDigitalSetDirection (*dwHandle*, *wDirection* , *pnStatus*)

Parameters

Name	**Type**	**Comments**
dwHandle	DWORD	Handle to a demo board.
wDirection	WORD	The direction of the DIO channels. 1 is Input, 0 is Output. The 8 digital I/O channels are represented by 8 bits. Each bit represents a channel starting with channel 0, e.g. channel 0 is represented by bit 0.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Example

The following example sets the direction of the digital channels so that D0-D3 are input and D4-7 are output:

```
SHORT nStatus;
DWORD dwHandle;
WORD wDirection;

GtDemoDigitalSetDirection (dwHandle, 0x0F, &nStatus)
```

See Also

GtDemoDigitalGetDirection, **GtDemoGetErrorString**

GtDemoDigitalWriteData

Purpose

Sets the value of the digital I/O's output channels.

Syntax

GtDemoDigitalWriteData (*dwHandle, wValues, pnStatus*)

Parameters

Name	Type	Comments
dwHandle	DWORD	Handle to a demo board.
wValues	WORD	The 8 digital I/O channels are represented by 8 bits. Each bit represents a channel starting with channel 0, e.g. channel 0 is represented by bit 0..
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Example

The following example writes an alternating pattern to the digital channels, the even channels are high and the odd channels are low:

```
SHORT nStatus;
DWORD dwHandle;

GtDemoDigitalSetDirection (dwHandle, 0x0, &nStatus);
GtDemoDigitalWriteData (dwHandle, 0xAA, &nStatus);
```

See Also

GtDemoDigitalReadData, **GtDemoGetErrorString**

GtDemoDipSwitchGetState

Purpose

Returns the state of the specified dip switch.

Syntax

GtDemoDipSwitchGetState (*dwHandle,enDipSwitchChannel, pbEnabled, pnStatus*)

Parameters

Name	**Type**	**Comments**
dwHandle	DWORD	Handle to a demo board.
enDipSwitchChannel	LONG	Specifies the dip switch channel: 0 = GTDEMO_DIP_CHANNEL1 1 = GTDEMO_DIP_CHANNEL2 2 = GTDEMO_DIP_CHANNEL3
pbEnabled	PBOOL	Specifies the dip switch status: TRUE is on, FALSE is off.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Example

The following example gets the state of the second dip switch:

```
SHORT nStatus;

DWORD dwHandle;

BOOL bDipSwitchOn;


GtDemoDipSwitchGetState (dwHandle, GTDEMO_DIP_CHANNEL2, &bDipSwitchOn,
&nStatus);
```

See Also

GtDemoGetErrorString

GtDemoDisplayClear

Purpose

Clears the LCD display.

Syntax

GtDemoDisplayClear (*dwHandle*, *pnStatus*)

Parameters

Name	Type	Comments
dwHandle	DWORD	Handle to a demo board.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Comments

This function blanks the screen, removing any user set text and/or the splash screen.

Example

The following example clears text from the LCD display:

```
SHORT nStatus;
DWORD dwHandle;

GtDemoDisplayClear (dwHandle, &nStatus)
```

See Also

GtDemoGetErrorString

GtDemoDisplayGetLineNumber

Purpose

Returns the current line number of the LCD display.

Syntax

GtDemoDisplayGetLineNumber (*dwHandle*, *plLineNumber* , *pnStatus*)

Parameters

Name	**Type**	**Comments**
dwHandle	DWORD	Handle to a demo board.
plLineNumber	PLONG	Returns the current LCD display’s line number.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Comments

The user can only write to one LCD line at a time. This function allows you to determine to which line a call to GtDemoDisplaySetText will write.

Example

The following example returns the current display line:

```
SHORT nStatus;

LONG lLineNumber;

DWORD dwHandle;


GtDemoDisplayGetLineNumber (dwHandle, &lLineNumber, &nStatus);
```

See Also

GtDemoDisplaySetLineNumber, **GtDemoGetErrorString**

GtDemoDisplaySetLineNumber

Purpose

Sets the current line number of the LCD display.

Syntax

GtDemoDisplaySetLineNumber (*dwHandle*, *lLineNumber*, *pnStatus*)

Parameters

Name	Type	Comments
dwHandle	DWORD	Handle to a demo board.
lLineNumber	LONG	Specifies the LCD display line number: 1 to 8.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Comments

The user can only write to one line of the LCD display at a time.

Example

The following example writes the word 'Hello' to the LCD display on line number 4:

```
SHORT nStatus;
DWORD dwHandle;

GtDemoDisplaySetLineNumber (dwHandle, 4, &nStatus);
GtDemoDisplaySetText (dwHandle, "Hello", &nStatus);
```

See Also

GtDemoGetLineNumber, **GtDemoDisplaySetText**, **GtDemoGetErrorString**

GtDemoDisplaySetText

Purpose

Sets the new text to the LCD's current line.

Syntax

GtDemoDisplaySetText (*dwHandle*, *sLineText*, *pnStatus*)

Parameters

Name	Type	Comments
dwHandle	DWORD	Handle to a demo board.
sLineText	PSTR	Specifies the line text, up to 21 characters.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Comments

The user can only write to one line of the LCD display at a time. Use GtDemoDisplaySetLineNumber to set which line this function will write.

Example

The following example writes the word ‘Hello’ to the LCD display on line number 4:

```
SHORT nStatus;
DWORD dwHandle;

GtDemoDisplaySetLineNumber (dwHandle, 4, &nStatus);
GtDemoDisplaySetText (dwHandle, “Hello”, &nStatus);
```

See Also

GtDemoDisplaySetLineNumber, **GtDemoGetErrorString**

GtDemoGetErrorString

Purpose

Returns the error string associated with the specified error number.

Syntax

GtDemoGetErrorString (*nError*, *psErrorMsg*)

Parameters

Name	**Type**	**Comments**
nErrorCode	SHORT	Error number.
psErrorMsg	PSTR	Buffer to the returned error string.
nErrorMaxLen	SHORT	The size of the error string buffer.

Comments

The function returns the last error code and error string that was generated. If an error is generated by the GTDEMO driver, it will be reported through this function.

The following table displays the possible error values that can be reported by the

Resource Errors

0	No error has occurred
-100	Bad command. The SCPI command entered does not exist
-108	Too many parameters were passed
-109	Too few parameters were passed
-200	Execution problem
-203	Bad or unentered password, cannot execute command
-220	Problem with a parameter, maybe entered in a previous command
-222	Parameter was out of range
-225	The device has insufficient memory to perform the requested operation
-240	Communication with hardware (EEPROM, DAC, etc) failed
-300	Unable to start communication with the specified demo board
-350	Error Queue Overflow

Example

The following example initializes the board. If the initialization failed, the following error string is printed:

```
CHAR    sz[256];
SHORT   nStatus;
GtDemoDisplaySetText (dwHandle, “Hello”, &nStatus);
if (nStatus<0)
{
    GtDemoGetErrorString(nStatus, sz, 256);
    printf(sz); // prints the error string returns
}
```

GtDemoInitialize

Purpose

Initializes the driver for the demo board.

Syntax

GtDemoInitialize (*sSerialNumber*)

Parameters

Name	Type	Comments
sSerialNumber	PSTR	GT98901 board's serial number

Comments

When called, will update the interface options so that commands can be sent. This function does not need to be called if the user sets up the interface in the ATEasy Driver Shortcut properties. Calling this function will not change or update any of the devices on the GT98901. It simply establishes communication.

Example

The following example initializes a demo board with the serial number 00033:

```
SHORT nStatus;
DWORD dwHandle;

GtDemoInitialize(“00033”, &dwHandle, &nStatus);
```

See Also

GtDemoGetErrorString, GtDemoReset

GtDemoLedSetState

Purpose

Sets the state of the specified LED.

Syntax

GtDemoLedSetState (*dwHandle*, *enLed*, *enLedState*, *pnStatus*)

Parameters

Name	**Type**	**Comments**
dwHandle	DWORD	Handle to a demo board.
enLed	LONG	Specifies the target LED: 1 = GTDEMO_LED_ORANGE 2 = GTDEMO_LED_GREEN
enLedState	LONG	Specifies the LED state: • 0 = GTDEMO_LED_OFF • 1 = GTDEMO_LED_ON
pnStatus	PSHORT	Returned status: 0 on success, 1 on failure.

Example

The following example turns the leds on and then off:

```
SHORT nStatus;
DWORD dwHandle;

GtDemoLedSetState(dwHandle, GTDEMO_LED_ORANGE, GTDEMO_LED_ON, &nStatus);
GtDemoLedSetState(dwHandle, GTDEMO_LED_GREEN, GTDEMO_LED_ON, &nStatus);

GtDemoLedSetState(dwHandle, GTDEMO_LED_ORANGE, GTDEMO_LED_OFF, &nStatus);
GtDemoLedSetState(dwHandle, GTDEMO_LED_GREEN, GTDEMO_LED_OFF, &nStatus);
```

See Also

GtDemoGetErrorString

GtDemoReferenceGetDivider

Purpose

Returns the divider of the reference clock.

Syntax

GtDemoReferenceGetDivider (*dwHandle*, *pnDivider*, *pnStatus*)

Parameters

Name	Type	Comments
dwHandle	DWORD	Handle to a demo board.
pnDivider	PSHORT	Returns the current reference clock divider.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Example

The following example returns the current divider applied to the reference clock:

```
SHORT nStatus, nDivider;
DWORD dwHandle;

GtDemoReferenceGetDivider (dwHandle, &nDivider, &nStatus)
```

See Also

GtDemoReferenceSetDivider, **GtDemoGetErrorString**

GtDemoReferenceSetDivider

Purpose

Sets the divider of the reference clock.

Syntax

GtDemoReferenceSetDivider (*dwHandle*, *nDivider*, *pnStatus*)

Parameters

Name	Type	Comments
dwHandle	DWORD	Handle to a demo board.
nDivider	SHORT	Specifies the reference clock divider: 0 to 15.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Comments

The base reference clock operates at 120 MHz. This function can be to used to set the divider N, where the reference clock output will be 120 MHz / 2^N.

Example

The following example sets the reference clock to operate at 30 MHz:

```
SHORT nStatus;
DWORD dwHandle;

GtDemoReferenceSetDivider (dwHandle, 2, &nStatus);
```

See Also

GtDemoReferenceGetDivider, **GtDemoGetErrorString**

GtDemoRelayGetState

Purpose

Returns the state of the specified SPDT relay.

Syntax

GtDemoRelayGetState (*dwHandle*, *enRelay*, *penRelayState*, *pnStatus*)

Parameters

Name	**Type**	**Comments**
dwHandle	DWORD	Handle to a demo board.
enRelay	LONG	Specified relay: • 1 = GTDEMO_RELAY_K1 • 2 = GTDEMO_RELAY_K2
penRelayState	PLONG	Returned state of the specified relay: • 0 = GTDEMO_RELAY_OPEN • 1 = GTDEMO_RELAY_CLOSE
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Example

The following example returns the state of the K1 relay:

```
SHORT nStatus;
DWORD dwHandle;
LONG lRelayState;

GtDemoRelayGetState (dwHandle, GTDEMO_RELAY_K1, &lRelayState, &nStatus);
```

See Also

GtDemoRelaySwitch, **GtDemoGetErrorString**

GtDemoRelaySwitch

Purpose

Sets the state of the specified SPDT relay.

Syntax

GtDemoRelaySwitch (*dwHandle, enRelay, enRelayState, pnStatus*)

Parameters

Name	Type	Comments
dwHandle	DWORD	Handle to a demo board.
enRelay	LONG	Specified relay: • 1 = GTDEMO_RELAY_K1 • 2 = GTDEMO_RELAY_K2
enRelayState	LONG	Sets the state of the specified relay: • 0 = GTDEMO_RELAY_OPEN • 1 = GTDEMO_RELAY_CLOSE
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Example

The following example sets both relays to connect their CO to their NO ports:

```
SHORT nStatus;
DWORD dwHandle;

GtDemoRelaySwitch (dwHandle, GTDEMO_RELAY_K1, GTDEMO_RELAY_OPEN, &nStatus);
GtDemoRelaySwitch (dwHandle, GTDEMO_RELAY_K2, GTDEMO_RELAY_OPEN, &nStatus);
```

See Also

GtDemoRelayGetState, **GtDemoGetErrorString**

GtDemoReset

Purpose

Resets the board to its default settings.

Syntax

GtDemoReset (*dwHandle*, *pnStatus*)

Parameters

Name	**Type**	**Comments**
dwHandle	DWORD	Handle to a demo board.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Comments

Default settings are as follow:

The LCD screen will display the company logo and model number.

The SPDT relays will be set to their normally closed position.

The LEDs will be turned off.

The DACs will output 0 Volts.

All DIO ports will be set to Input only.

The buzzer will be turned off.

Example

The following example initializes and resets the demo board:

```
DWORD dwHandle;
SHORT nStatus;

GtDemoInitialize (“00001”, &dwHandle, &nStatus);
GtDemoReset (dwHandle, &nStatus);
```

See Also

GtDemoInitialize, **GtDemoGetErrorString**

GtDemoPanel

Purpose

Opens a virtual panel used to interactively control the demo board.

Syntax

GtDemoPanel (*pdwHandle, hwndParent, nMode, phwndPanel, pnStatus*)

Parameters

Name	Type	Comments
pdwHandle	PDWORD	Handle to a demo board.
hwndParent	HWND	Panel parent window handle. A value of 0 sets the desktop as the parent window.
nMode	SHORT	The mode in which the panel main window is created. 0 for modeless window and 1 for modal window.
phwndPanel	HWND	Returned window handle for the panel.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Comments

The function is used to create the panel window. The panel window may be open as a modal or a modeless window depending on the *nMode* parameters.

If the mode is set to modal dialog (*nMode*=1), the panel will disable the parent window (*hwndParent*) and the function will return only after the window closes. In that case, the *pdwHandle* may return the handle created by the user using the panel Initialize dialog. This handle may be used when calling other demo board functions.

If a modeless dialog was created (*nMode*=0), the function returns immediately after creating the panel window returning the handle to the panel - *phwndPanel.* It is the responsibility of calling program to dispatch windows messages to this window so that the window can respond to messages.

Example

The following example opens the panel in modal mode:

```
DWORD dwPanel, dwHandle=0;
SHORT nStatus;

GtDemoPanel(&dwHandle, 0, 1, &dwPanel, &nStatus);
```

See Also

GtDemoInitialize, **GtDemoGetErrorString**

GtDemoPushButtonGetState

Purpose

Returns the push button flag.

Syntax

GtDemoPushButtonGetState (*dwHandle, pbPushed, pnStatus*)

Parameters

Name	**Type**	**Comments**
dwHandle	DWORD	Handle to a demo board.
pbPushed	PBOOL	Returns the push button flag. See comments.
pnStatus	PSHORT	Returned status: 0 on success, negative number on failure.

Comments

Determines if the push button has been pressed. Calling this function will clears the button-pushed flagged. TRUE if pushed, FALSE if not pushed.

Example

The following example reads the push button state:

```
SHORT nStatus;
DWORD dwHandle;
BOOL bPushed;

GtDemoPushButtonGetState (dwHandle, &bPushed, &nStatus);
```

See Also

GtDemoGetErrorString

GtDemoSystemGetIdentification

Purpose

Returns the board's description and firmware information.

Syntax

GtDemoSystemGetIdentification (*dwHandle*, *psDescription*, *pnStatus*)

Parameters

Name	Type	Comments
dwHandle	DWORD	Handle to a demo board.
psDescription	PSTR	Buffer to receive the summary string.
pnStatus	LPSHORT	Returned status: 0 on success, negative number on failure.

Example

The following example returns the driver summary:

```
SHORT nStatus;
DWORD dwHandle;
CHAR szSummary[128];

GtDemoSystemGetIdentification(dwHandle, szSummary, &nStatus);
```

See Also

GtDemoGetErrorString

Appendix B – SCPI Command Reference

Introduction

The SCPI command reference chapter organizes the list of GT98901 SCPI commands in an alphabetical order. Each command description contains the function name; purpose, syntax, parameters description and type followed by Comments, an Example and a See Also sections.

GT98901 SCPI Commands

The following list is a summary of functions available for the GT98901:

Driver Functions	Description
ADC:FETC	Reads measurements from the analog input buffer.
ADC:READ	Triggers a measurement from the specified analog input channel.
ADC:SCAN	Sets or returns the number of samples for analog input measurements.
ADC:TRIG	Triggers scanning on the specified analog input channel.
DAC:ABOR	Disables output of the analog burst signal.
DAC:BUFF:SCAN	Set or query the number of scans to make after triggering the analog output channels.
DAC:BUFF:DATA	Set or query the data that will be loaded into the current analog output page.
DAC:CLOC:DIV	Set or query the analog output channel's clock divider.
DAC:RUN	Queries the DAC for its current run status.
DAC:TRIG	Begin generating waveforms on both analog output channels.
DAC:OUTP	Output a static voltage on the specified channel.
DIO:OUTP:STATE	Set or query the direction of the digital input / output channels.
DIO:OUTP	Sets the output values of the digital input / output channels.
DIO:INP	Queries the values of the digital input / output channels.
DIO:PWM	Sets or queries the state of the PWM.
DIO:PWM:DUTY	Sets or queries the duty cycle percentage of the PWM.
DIO:PWM:DIV	Sets or queries the clock divider for the 120 MHz PWM clock.
DIO:PWM:PER	Sets or queries the period of the PWM.
DISP:CLEA	Clears all the lines of the LCD display.
DISP:TEXT	Prints the specified text to the currently selected display line.
DISP:LINE	Sets or queries the currently selected display line.
REL:OUTP	Sets or queries the state of the SPDT relays.
LED	Sets or queries the state of the specified LED
BUZZ	Sets or queries the state of the on-board speaker.

Driver Functions	Description
REF:DIV	Sets or queries the divider of the 120 MHz reference clock.
PUSH:READ	Queries the state of the push button flag and then resets it.
DIP	Queries the state of the specified dip switch.
SYST:ERR	Returns the last error received and removes it from the buffer.
***RST**	Resets the board to a known state.

ADC:FETC

Command Type

Query only.

Purpose

Get the data that is waiting in the analog input buffer. Samples in response are separated by commas.

Syntax

ADC:FETC?

Example Query Response

3.141,5.926,5.358,9.793.2.384\n

ADC:READ

Command Type

Query only.

Purpose

Take a single reading from the specified analog input channel.

Syntax

ADC:READ:[AIN1/AIN2/AIN3/AIN4]?

Example Query Response

1.234\r\n

ADC:SCAN

Command Type

Set or Query.

Purpose

Set or query the number of scans to make after triggering one of the analog input channels.

Syntax

ADC:SCAN <1 to 30>

ADC:SCAN?

Example Query Response

1\r\n

ADC:TRIG

Command Type

Set only.

Purpose

Begin sampling on the specified channel.

Syntax

ADC:TRIG:[AIN1/AIN2/AIN3/AIN4]

DAC:ABOR

Command Type

Set only.

Purpose

Stop generating waveforms from the analog output channels.

Syntax

DAC:ABOR

DAC:BUFF:SCAN

Command Type

Set or Query

Purpose

Set or query the number of scans to make after triggering the analog output channels.

Syntax

DAC:BUFF:SCAN <1 to 128>

DAC:BUFF:SCAN?

Example Query Response

13\r\n

DAC:BUFF:DATA

Command Type

Set or Query.

Purpose

Set or query the data that will be loaded into the current analog output page. When setting, each sample is separated by a comma.

Syntax

DAC:BUFF:DATA sample,sample,sample

DAC:BUFF:DATA?

Example Query Response

3.141,5.926,5.358,9.793.2.384\n

DAC:BUFF:PAGE

Command Type

Set or Query.

Purpose

Set or query the current DAC page number. 0 to 7 are for AOut1 and 8 to 15 are for AOut2.

Syntax

DAC:BUFF:PAGE <0 to 15>

DAC:BUFF:PAGE?

Example Query Response

0\n

DAC:CLOC:DIV

Command Type

Set or Query.

Purpose

Set or query the analog output channel's clock divider.

Syntax

DAC:CLOC:DIV <1 to 65535>

DAC:CLOC:DIV?

Example Query Response

1024\n

DAC:RUN

Command Type

Query only.

Purpose

Checks if the DACs are currently running in dynamic mode. A response of 1 indicates the the DACs are currently sampling and 0 indicates that they are idle.

Syntax

DAC:RUN?

Example Query Response

1\n

DAC:TRIG

Command Type

Set Only.

Purpose

Begin generating waveforms on both analog output channels.

Syntax

DAC:TRIG

DAC:OUTP

Command Type

Set Only.

Purpose

Output a static voltage on the specified channel. This setting is ignored in the DACs are currently running in dynamic/waveform generation mode. The sample can be within -10 to 10.

Syntax

DAC:OUTP:[AOUT1/AOUT2] <sample>

DIO:OUTP:STATE

Command Type

Set or Query.

Purpose

Set or query the direction of the digital input/output channels. The data will be formatted as a two digit hexadecimal number. Each bit represents a digital channel's direction: the LSB represents D0 and the MSB represents D7. A 0 represents the output direction and a 1 represents input.

Syntax

DIO:OUTP:STATE <0 to 255 or #H00 to #HFF>

DIO:OUTP:STATE?

Example Query Response

127\n

DIO:OUTP

Command Type

Set Only.

Purpose

Sets the output values of the digital input/output channels. The data will be formatted as a two digit hexadecimal number. Each bit represents a digital channel's logical output: the LSB represents D0 and the MSB represents D7. A 0 represents a logical low output and a 1 represents a logical high. This command does not affect channels currently set to input.

Syntax

DIO:OUTP <0 to 255 or #H00 to #HFF>

DIO:INP

Command Type

Query Only.

Purpose

Queries the values of the digital input/output channels. The data will be formatted as a two digit hexadecimal number. Each bit represents a digital channel's logical output: the LSB represents D0 and the MSB represents D7. A 0 represents a logical low output and a 1 represents a logical high.

Syntax

DIO:INP?

Example Query Response

85\n

DIO:PWM

Command Type

Set or Query.

Purpose

Sets or queries the state of the PWM. ON indicates that the PWM is current outputting a pulse train, OFF indicates that it is not active.

Syntax

DIO:PWM [ON | OFF]

DIO:PWM?

Example Query Response

OFF\n

DIO:PWM:DUTY

Command Type

Set or Query.

Purpose

Sets or queries the duty cycle percentage of the PWM.

Syntax

DIO:PWM:DUTY <0 to 100>

DIO:PWM:DUTY?

Example Query Response

50\n

DIO:PWM:DIV

Command Type

Set or Query.

Purpose

Sets or queries the clock divider for the 120MHz PWM clock.

Syntax

DIO:PWM:DIV [1 | 2 | 4 | 8 | 16 | 32 | 64]

DIO:PWM:DIV?

Example Query Response

8\n

DIO:PWM:PER

Command Type

Set or Query.

Purpose

Sets or queries the period of the PWM.

Syntax

DIO:PWM:PER <1 to 65535>

DIO:PWM:PER?

Example Query Response

1250\n

DISP:CLEA

Command Type

Set Only.

Purpose

Clears all the lines of the LCD display.

Syntax

DIO:CLEA

DISP:TEXT

Command Type

Set Only.

Purpose

Prints the specified text to the currently selected display line.

Syntax

DISP:TEXT <0 to 21 ASCII characters>

DISP:LINE

Command Type

Set or Query.

Purpose

Sets or queries the currently selected display line. The selected display line is the line which is updated when DISP:TEXT is called.

Syntax

DIO:LINE <1 to 8>

DIO:LINE?

Example Query Response

1\n

REL:OUTP

Command Type

Set or Query.

Purpose

Sets or queries the state of the SPDT relays. SET will create a connection between common and the normally open contact. RES will reset the connection so the short is between common and normally closed.

Syntax

REL:OUTP:[K1 | K2] [SET/RES]

REL:OUTP:[K1 | K2]?

Example Query Response

SET\n

LED

Command Type

Set or Query.

Purpose

Sets or queries the state of the specified LED.

Syntax

LED:[GRN | ORN] [ON | OFF]>

LED:[GRN | ORN]?

Example Query Response

OFF\n

BUZZ

Command Type

Set or Query.

Purpose

Sets or queries the state of the on-board speaker.

Syntax

BUZZ [ON | OFF]

BUZZ?

Example Query Response

ON\n

REF:DIV

Command Type

Set or Query.

Purpose

Sets or queries the divider of the 120 MHz reference clock to 2 to the Nth power.

Syntax

REF:DIV <0 to 15>

REF:DIV?

Example Query Response

6\n

PUSH:READ

Command Type

Query only.

Purpose

Queries the state of the push button flag and then resets it.

Syntax

PUSH:READ?

Example Query Response

0\n

DIP

Command Type

Query only.

Purpose

Queries the state of the specified dip switch. Returns 0 is the switch is off, and 1 is the switch is on.

Syntax

DIP:[SWT1 | SWT2 | SWT3]?

Example Query Response

1\n

SYST:ERR

Command Type

Query only.

Purpose

Returns the last error received and removes it from the buffer. Buffer is 10 deep, also includes a short description of the error.

Syntax

SYST:ERR?

Example Query Response

ON\n

*RST

Command Type

Set only.

Purpose

Returns the demo board to its default state.

Syntax

***RST**

Index