ATEasy

Test Executive and Development Studio

Getting Started With ATEasy

GEOTEST – Marvin Test Systems, Inc.

Part No. GT90302 - Revision 8.0.146 (September 3, 2010)

Disclaimer

In no event shall Geotest or any of its representatives be liable for any consequential damages whatsoever (including, without limitation, damages for loss of business profits, business interruption, loss of business information, other loss or injury) arising out of the use of or inability to use this product, even if Geotest has been advised of the possibility for such damages. Geotest products are not intended for life critical medical use.

Support and Subscription

Unless otherwise specified, ATEasy is provided with one year support and subscription agreement. The agreement provides free support for a period of one year using our web based support portal http:\\www.geotestinc.com\magic\. In addition, any new version released during that year can be download and used with no added cost. The subscription and support agreement can be renewed on a yearly basis provided it is done before the agreement ends. See Geotest web site http:\\www.geotestinc.com\ for more information regarding the ***ATEasy*** licensing, support, subscription, upgrade, downloads, training, knowledge base and user forums.

Copyright and Version

This manual was updated to ***ATEasy*** version 8.0.

Trademarks

ATEasy, DIOEasy, WaveEasy.	Geotest – MTS, Inc.
Microsoft Developer Studio,.NET, Visual Basic, Visual C++, Excel, Word, Windows 95, 98, Me, NT, 2000, XP, VISTA, Windows 7.	Microsoft Corporation
LabView, LabWindows/CVI.	National Instruments Corporation

All other trademarks are the property of their respective companies.

TABLE OF CONTENTS

CHAPTER 1 - INTRODUCTION

ATEasy Getting Started Roadmap

This ***ATEasy*** Getting Started Guide provides all the information needed to install and use Geotest's ***ATEasy*** Automated Test Equipment (ATE) software development environment. This manual assumes you have a general knowledge of PC-based computers, Windows operating systems, and some knowledge of programming and development tools.

This manual is organized as follows:

Chapter	**Content**
1. Introduction	Introduces this Getting Started Guide.
2. Setup and Installation	Step-by-step directions on how to install and configure ***ATEasy***.
3. Overview of ***ATEasy***	An overview of ***ATEasy*** including the various module files, submodules, and the Integrated Development Environment (IDE) layout, menus, and windows.
4. Your First Project	Describes the relationship of projects and applications, the workspace, creating a project, and project settings.
5. Variables and Procedures	Explains how to declare variables, naming guidelines and properties of variables, and how to create and use procedures.
6. Drivers and Interfaces	Describes how to create and configure instrument Drivers and user interfaces.
7. Commands	Defines ***ATEasy*** commands.
8. Working with Forms	Describes using forms to control programs and instrument operation.
9. Working with External Libraries	Describes how to attach and use external libraries.
10. Where to Go From Here	Where to find more information on ***ATEasy*** topics and concepts.
Index	A guide to important topics and concepts in this manual.

Documentation Conventions

There are several naming conventions used throughout the documentation. The conventions used are:

Example	Description
Copy or **Paste**	Commands are indicated in bold type.
SHIFT+F1	Keys are often used in combinations. The example to the left instructs you to hold down the shift key while pressing the F1 key. When key combination instructions are separated by commas, such as ALT+D, A, hold the ALT key while pressing D, then press A.
cd drivers	Bolded text must be entered from the keyboard exactly as shown, especially for property, method, and event keywords.
Direction Keys	Refers to the up arrow (↑), down arrow (↓), right arrow (→), and left arrow (←) keys.
Variable	Italicized text is a placeholder for variables or other items that you must define and enter from the keyboard.
[expressionlist]	Items inside square brackets are optional.
{ **While** | **Until** }	Braces and vertical bar indicate a choice between items. You must choose one of them, unless all of them are enclosed in square brackets (optional).
GTSW1.DRV	Words in all capital letters indicate filenames.

`Examples`	Examples and source code are indicated in `Courier`, with appropriate indentation, for example
	`Procedure chk1.OnClick():Void Public`
	`{`
	`    chk1.Caption=“Hello”`
	`}`

0xhexnumber	An integer in hexadecimal notation, for example, 0x10A equals 266 in decimal.

Technical Support

General Information

Visit our web site for more information about ***ATEasy*** support options. Our Web site (http://www.geotestinc.com) contains sections describing: support options, application notes, download area for downloading upgrades, examples, instrument drivers, and how to submit support questions for ***ATEasy*** registered users.

ATEasy comes with one year subscription and support plan. The plan allows you access to our technical staff to get you started and provide basic assistance and entitled you for free upgrades during that year. The support is provided using our web-based support module name M@GIC (My Account@ Geotest Inc .COM) at http://www.geotestinc.com/magic. The web-based support allows you to log in and submit issues such as bug reports, problems, how to questions, suggestions and more. Once an issue is submitted you will be notified automatically when the issue status is changed, then you'll be able to revisit MA@GIC to reply to question or read answers posted by our professional tech support personal. Telephone support is also available on weekdays from 8:00 AM to 5:00 PM Pacific Standard Time (PST) by calling (949) 263-2222. For telephone support times and availability outside of the Americas, contact your local Geotest distributors.

After the initial year is over you can continue to receive free upgrades and technical support by renewing and purchasing additional Software Subscription and Support plan.

Software Subscription and Support

ATEasy's Subscription and Support is an all-inclusive service that provides premium support and software upgrades for ***ATEasy***. It includes:

- Unlimited technical support assistance for immediate solutions.
- Access to senior application engineers for advice and guidance in application development.
- Free updates and upgrades to ***ATEasy*** while you subscription is current.
- Free access to instrument drivers, examples, knowledge base and user forums at Geotest web site.
- Support and subscription can be renewed provided it is done before you current agreement ends.

Geotest provides both pre-sales and post-sales technical support for all products. Our Technical Support engineers are qualified to help you select hardware and software for your application, explain the specifications, and assist you in designing and building a complete test system.

To use your subscription and support plan please create a M@GIC account at Geotest web site http://www.geotestinc.com/magic. The account is also used to request software licenses and upgrades when a new version is available.

ATEasy Training

Geotest provides 3 and 5 days ATEasy training program. If you purchased a training program, contact your sales representative to schedule a training session.

Please consult our Web site or contact sales@geotestinc.com for more information and class schedules.

Contact Information

Geotest - Marvin Test Systems, Inc.

Phone: (949) 263-2222

Fax: (949) 263-1203

URL: http://www.geotestinc.com

Support: http://www.geotestinc.com/magic

Sales - mail to: sales@geotestinc.com

License - mail to: license@ateasy.net or use your M@GIC account

About 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.1ST 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:

- A 32 or 64 bit Windows system compatible with Microsoft Windows 98, Windows ME, Windows 2000, Windows XP, Windows Server 2003, or 2008, Windows VISTA, Windows 7.
- Internet Explorer 6.0 or later.
- 17 inch monitor or larger. At least 1024x768 resolution or higher is recommended for developing ***ATEasy*** applications.
- For ATEasy License Server: TCP/IP or IPX/SPX network protocols and Windows 2000 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***:

1. 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

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

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

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

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

6. 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\ATEasy`. For more information on the Installation Directories, see page 9.

7. Select the type of Setup you wish and click **Next.** For more information on the types of installation available, see page 9.

8. 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.

9. 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.

Setup Command Line

The following command line options for the setup are available:

ATEasy8 [/s [/SetupType"type"] /TargetDir"Path"]

Where:

1. Silent Installation. Optional.

   ```
   /s
   ```

2. Setup Types. Optional, may be used with /s.

   ```
   /SetupType["type"]
   ```

   The type can be one of Installation types: Typical, Full, Compact, Run-Time, License Server. The default type is "Typical".

3. Target Folder. Optional, may be used with /s.

   ```
   /TargetDir"Path"
   ```

   The default path is "c:\Program Files\ATEasy".

Silent (Automated) Setup

ATEasy can be installed in silent mode. Silent mode setup can be started by running the setup with the /s command line parameter. In silent mode no windows are displayed during the installation and the whole setup is automated based on the options selected from the setup command line. The silent mode setup is useful when incorporating the ***ATEasy*** setup to your own setup to distribute ***ATEasy*** applications and thus it allows you to automate the setup and the deployment of ***ATEasy*** applications.

Example

The following example installs ***ATEasy*** run time in silent mode (no user interface is displayed) to the default location c:\program files\ATEasy, ATEasy6.exe is the setup program downloaded from the Internet or under the CD Files folder:

```
ATEasy8 /s /SetupType”Run-Time”
```

Error Messages

In case errors occurred during the setup, ***ATEasy*** will report them in the file ATEasySetup.log in ***ATEasy*** Folder. It is recommended to review the log file to see if the installation is successful or not.

Installation Directories

ATEasy files are installed in the default folder `C:\Program Files\ATEasy`. 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 6 – 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.

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\Geotest\HW
   ```

4. At the command prompt, type the following command:

   ```
   HWSETUP -vdd install start
   ```

   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).

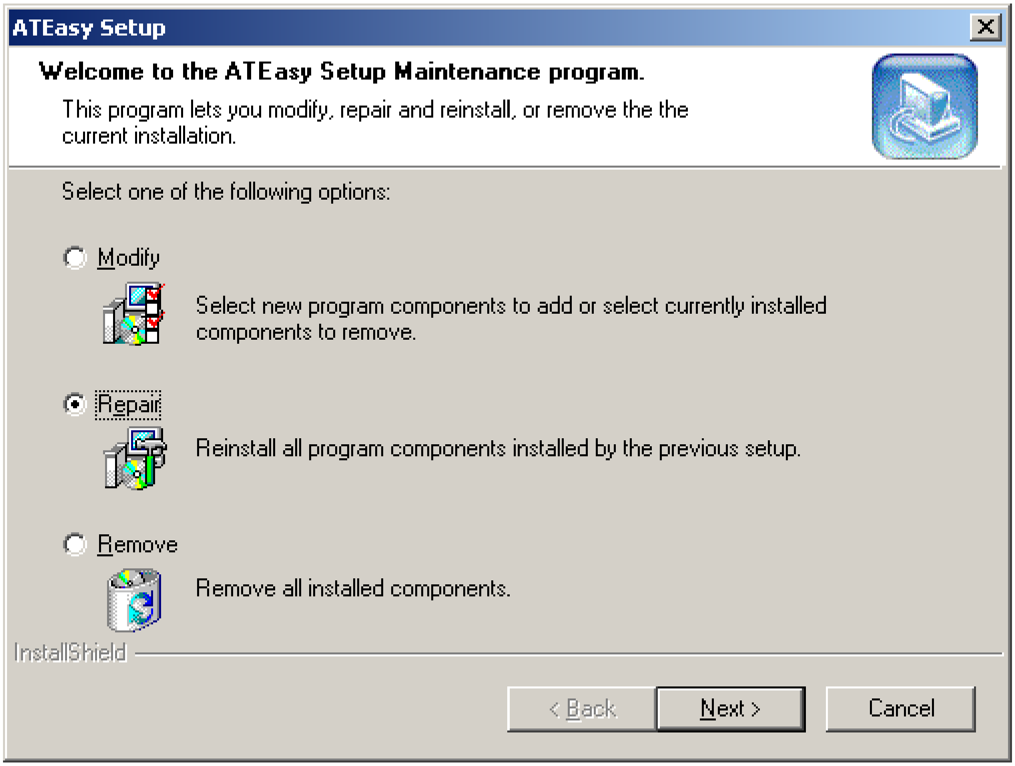
The Setup program HWSETUP.EXE, the device driver HW.SYS/HW64.SYS/HW.VXD and HWVDD.DLL files may be distributed with ***ATEasy*** applications. Additional HWSETUP.EXE command line options are available. To display these options, type **HWSETUP** without command line options.

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 Geotest. 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.geotestinc.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. See "*Technical Support*" on page 3 for more information about how to obtain support for ***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, ***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.

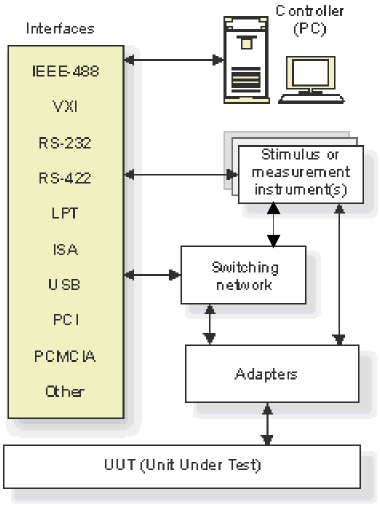
In addition, the ***ATEasy*** IDE provides a Rapid Application Development (RAD) environment. This provides a way to write, run and debug applications in very short cycles as required by instrument-based applications. The ***ATEasy*** IDE is an object-oriented environment, making the editing of common 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, USB 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.

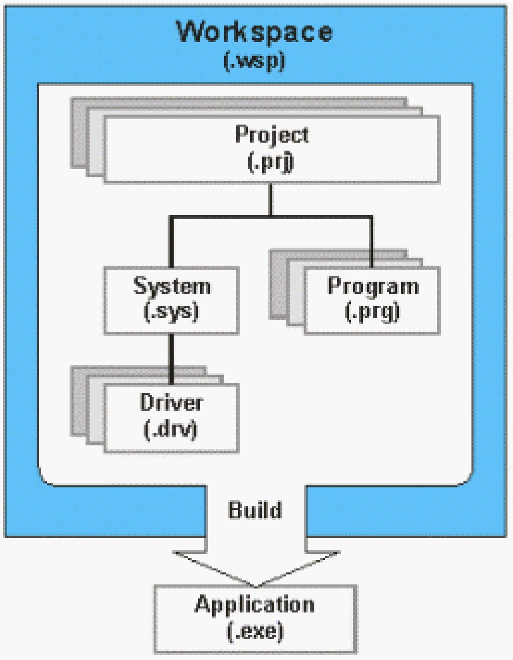
Workspace, Applications and Modules

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.

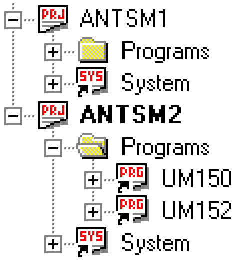
The diagram shows a Workspace, its Project file, Program, System and Driver modules.

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.

Submodule	Description
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: 1. **Dynamic Link Libraries** (DLLs), which is a file typically with .DLL file extensions containing procedures. 2. **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. 3. **.NET assemblies**, which contain classes, procedures, and other programming elements and is based on Microsoft .NET component technology. 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.

Submodule	Description
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.

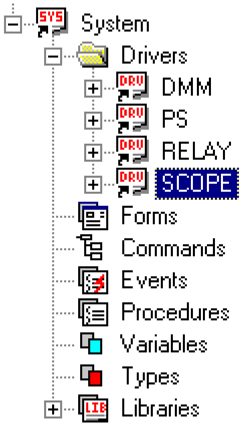
A **Test** contains programming statements (code) generating a single result (typically a measurement), or a status: **Pass, Fail, Error** or **None**. According to the type of Test, ***ATEasy*** can determine at runtime if the measurement is of acceptable value and generate a status. As an example, 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.

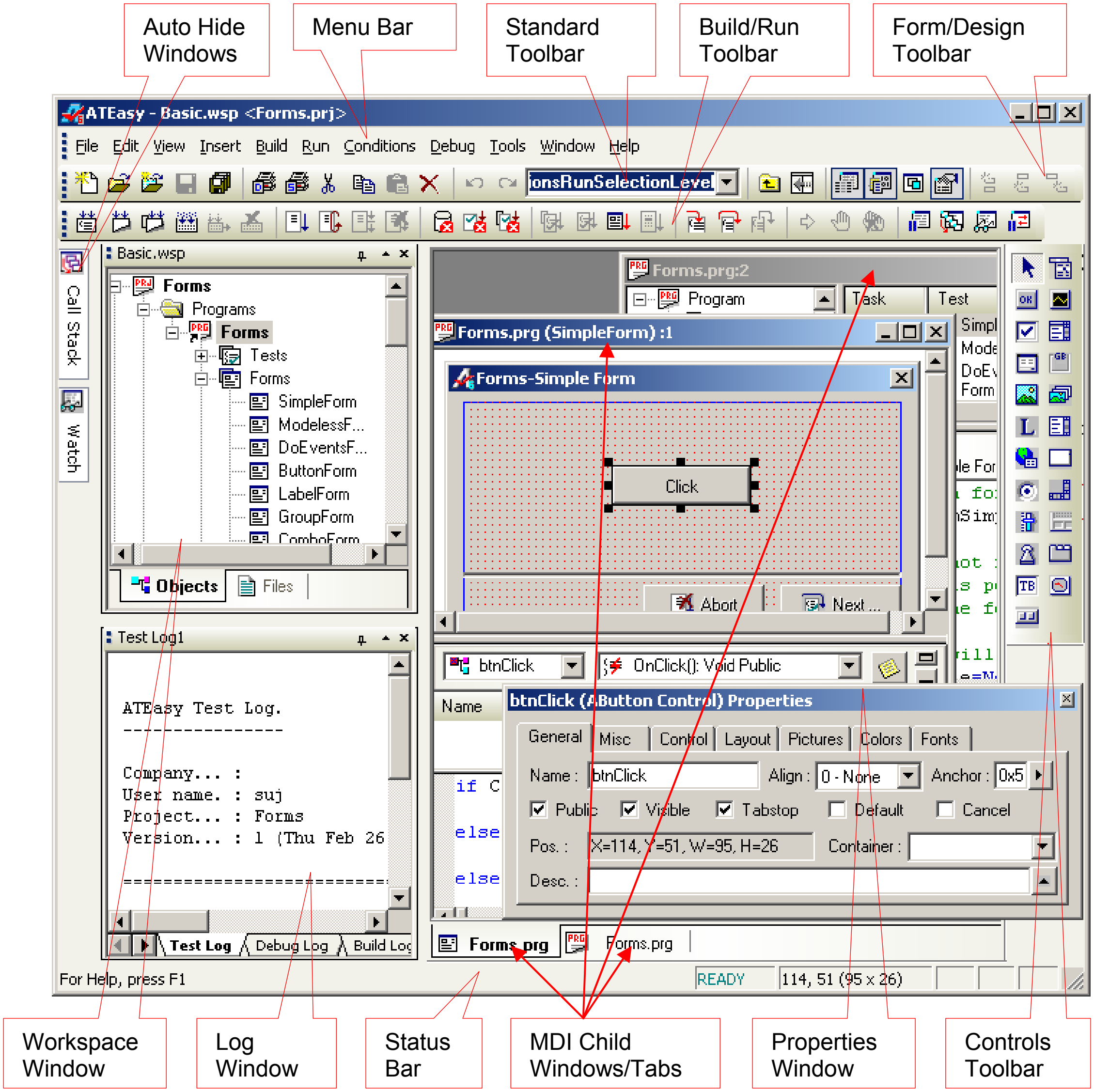
An ***ATEasy*** Driver is a reusable module. Any libraries or programming elements declared as public and used within the Driver are available to all other modules within the project by merely referencing them. The advantage is code duplication is avoided and code reuse is encouraged.

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

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

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

☑ Public ☑ Visible ☑ Tabstop □ Default □ Cancel

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

Desc. :

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.

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***. 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.

Note: A copy of the project that you will create throughout this *Getting Started Guide*, MyProject.prj, is located in the …\ATEasy\Examples folder, along with all of its associated files.

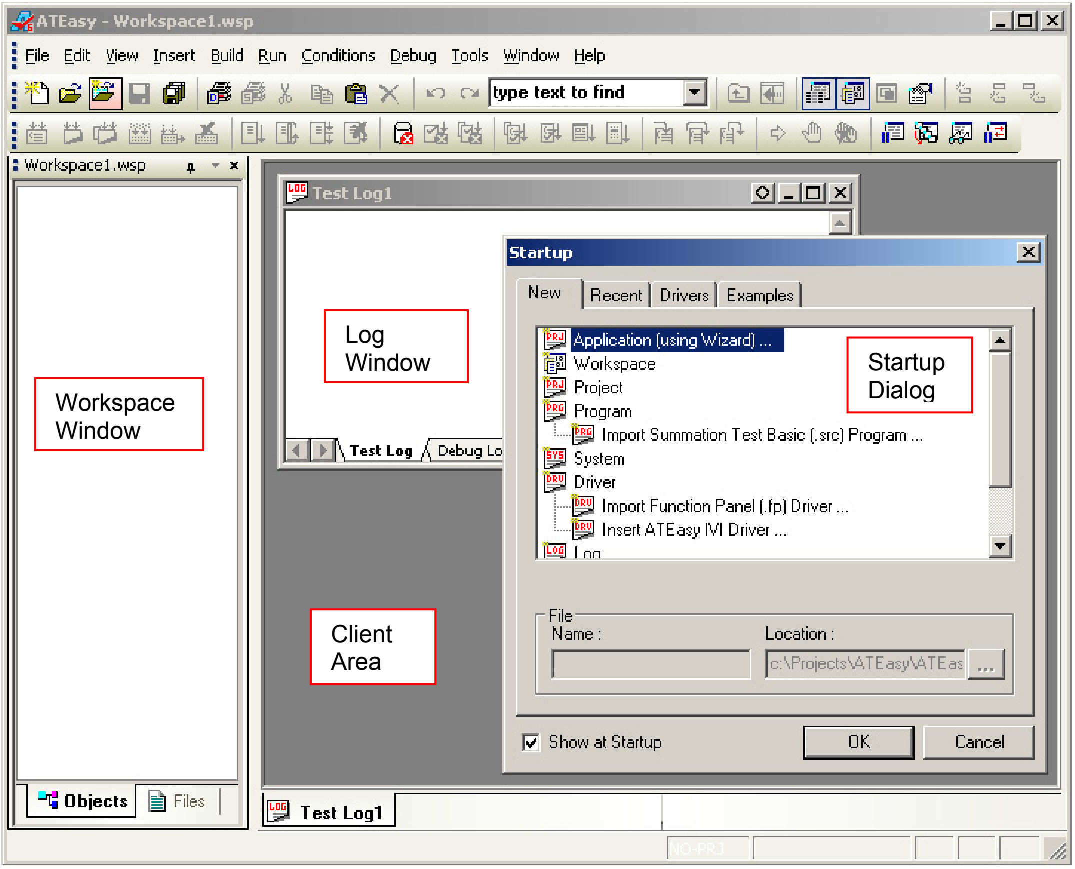
Starting *ATEasy*

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. A third driver FaultAnalysis.drv allows you to define Fault Conditions that are used to analyze your test results and thus provide a powerful troubleshooting tool.

- **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.

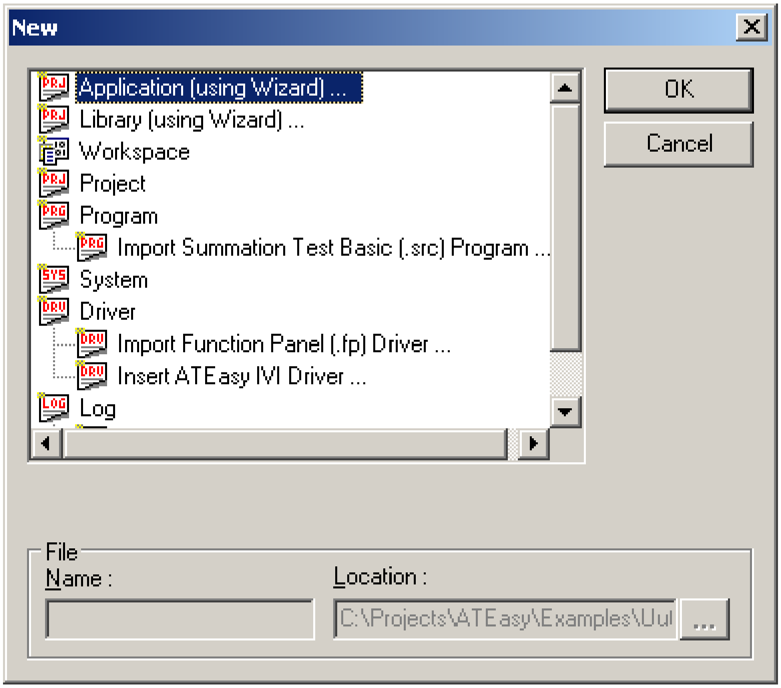
Creating an Application

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:

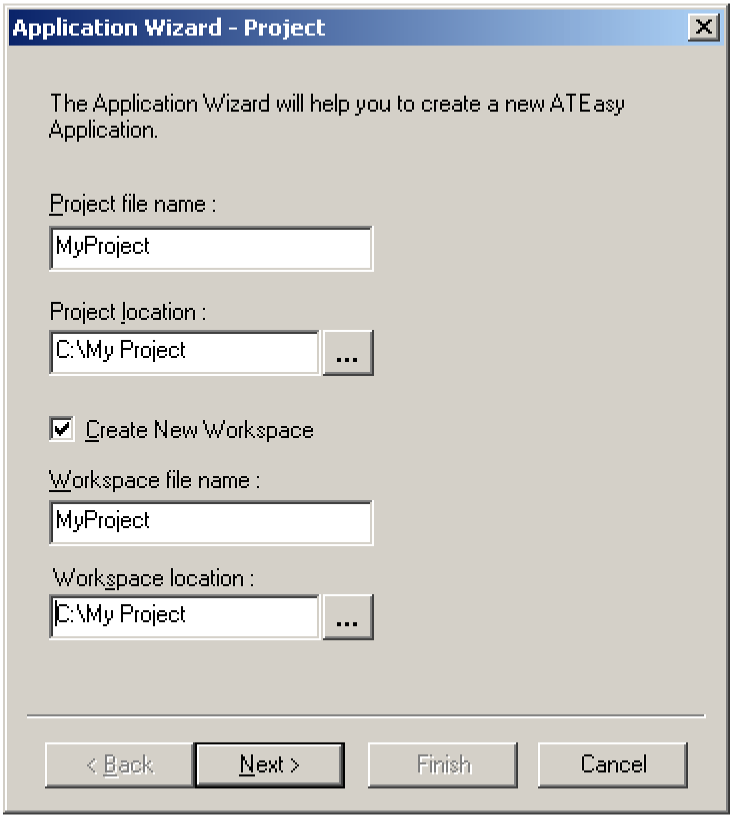

New
Application (using Wizard) ...
Library (using Wizard) ...
Workspace
Project
Program
Import Summation Test Basic (.src) Program ...
System
Driver
Import Function Panel (.fp) Driver ...
Insert ATEasy IVI Driver ...
Log
OK
Cancel
File
Name :
...

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:

1. 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.



Application Wizard - Project

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

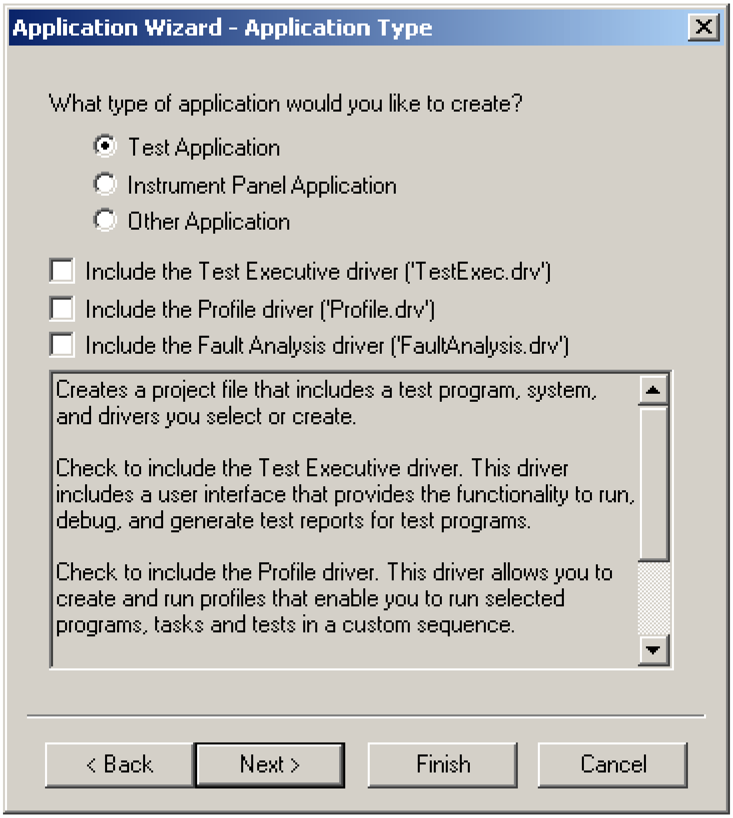
2. Make sure the checkbox is selected for **Create New Workspace and type the Workspace name and folder as shown here**.

3. 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:

1. Select **Test Application** as the project type.
2. Uncheck **Include the Test Executive** driver, the **Profile** driver **and the Fault Analysis driver**. The dialog should look as follows:



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.

3. Click **Next** to continue.

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

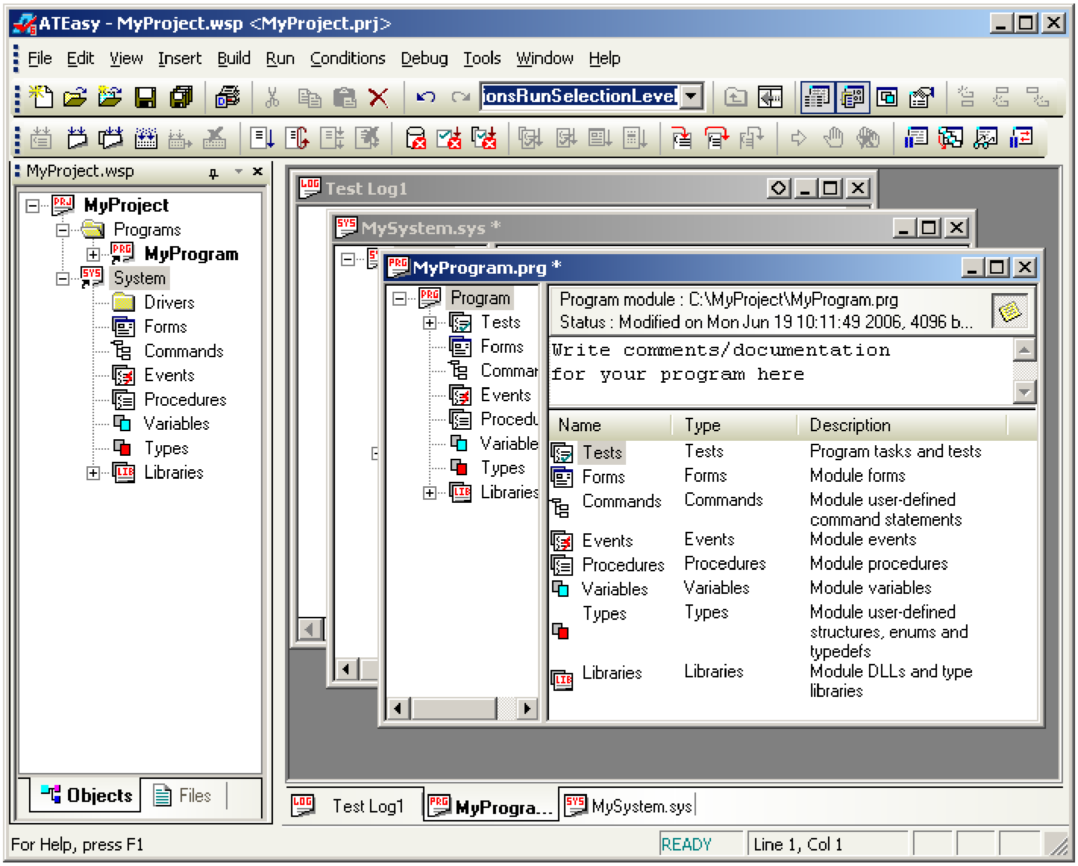
▼ To create the application:

1. 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.
2. 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.

3. 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\ATEasy\Drivers).

4. 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
```

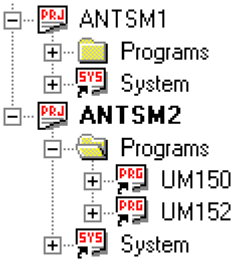
Name	Type	Description
Tests	Tests	Program tasks and tests
Forms	Forms	Module forms
Commands	Commands	Module user-defined command statements
Events	Events	Module events
Procedures	Procedures	Module procedures
Variables	Variables	Module variables
Types	Types	Module user-defined structures, enums and typedefs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

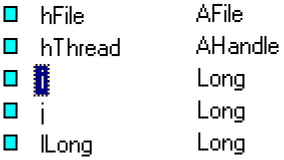
ANTSM1 and ANTM2 are projects: **ANTTM2** is the Active Project.



- **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.

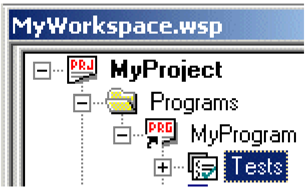


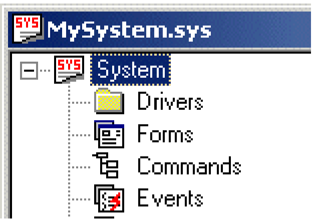
hFile	AFile
hThread	AHandle
i	Long
j	Long
lLong	Long

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.



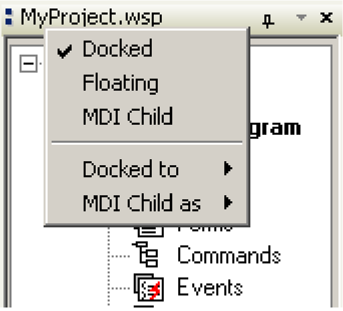


- **Dockable windows** – Dockable windows can be docked to any side of the IDE main window by dragging their title bar close to the border of the main window.

  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.

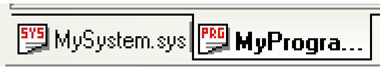
  You can see the differences between the states of these windows by right clicking on the diamond button appearing on the dockable window title bar.

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



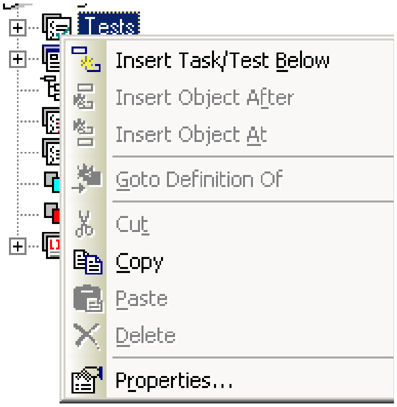
- **MDI child windows –** MDI stands for **M**ultiple **D**ocument **I**nterface. The interface displays multiple documents in the main window. Each document is in its own window, called an MDI Child window. In ***ATEasy***, all the document views as well as the built-in windows (for example, the Log 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.

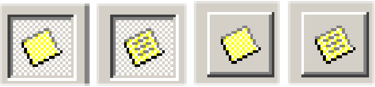


The second Tab referring to a document view displaying Tests in MyProgram.prg is the active MDI child.

- **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

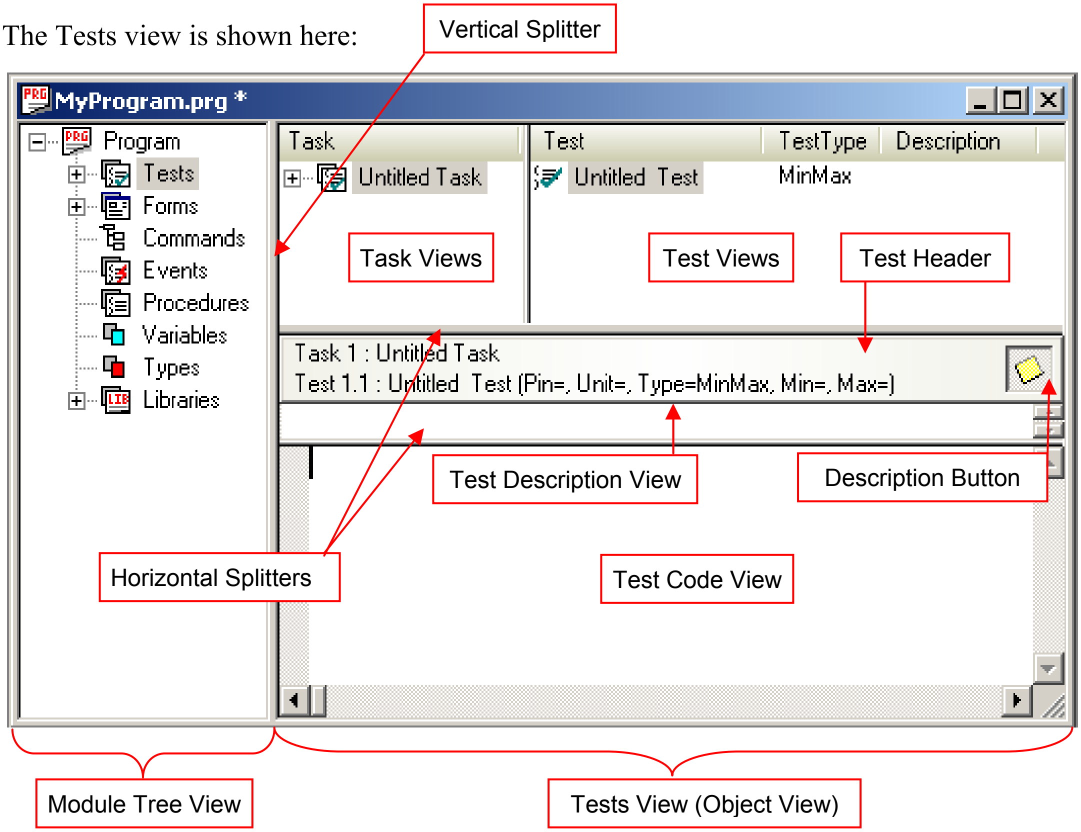
Your first program contains one task (Power Tests) and two tests (PS1 and PS2). Define the test requirements by first defining the test type and filling up the test properties. Then, write some code in the test to tell ***ATEasy*** the test result. Consequently, when you run this program you will see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

1. 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.
2. Do the same for the Untitled Test and rename it to **PS1**.

▼ To switch to Object View Only:

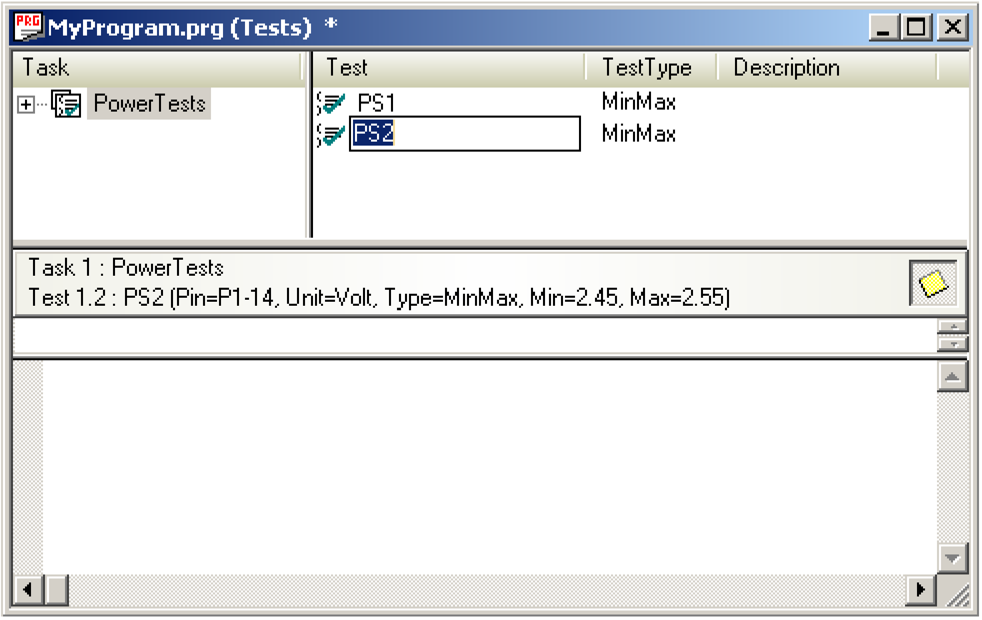
At this point, the document view displays the tree and object at the same time. Since you are planning to work only with the object view, it would be nice to have a larger work area on the screen.

1. 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:

1. 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.
2. Rename the new test from Untitled Test to **PS2**.

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



Test	TestType	Description
PS1	MinMax	
PS2	MinMax	

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.

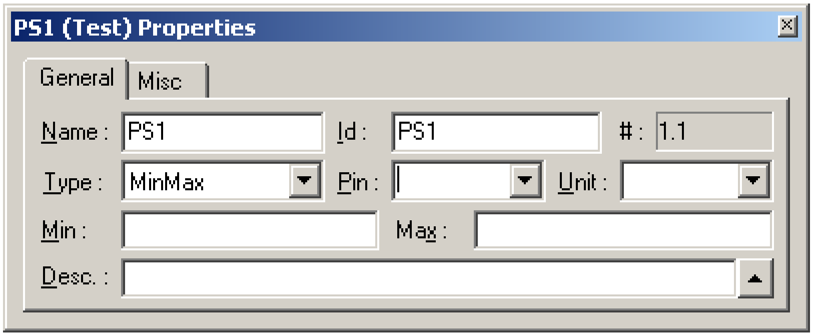
Test Properties

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:



PS1 (Test) Properties

Name : PS1
Id : PS1
: 1.1
Type : MinMax
Pin :
Unit :
Min :
Max :
Desc. :

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.
Synchronize (Misc Page)	Used for multi-threading or Multiple UUTs with parallel run mode applications. When Synchronize is checked, it allows only one thread to execute the test with the specified (optional) resource name, other threads executing tasks, tests or procedures with the same resource name will be suspended until the test is complete.
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:

1. 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**

2. 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.

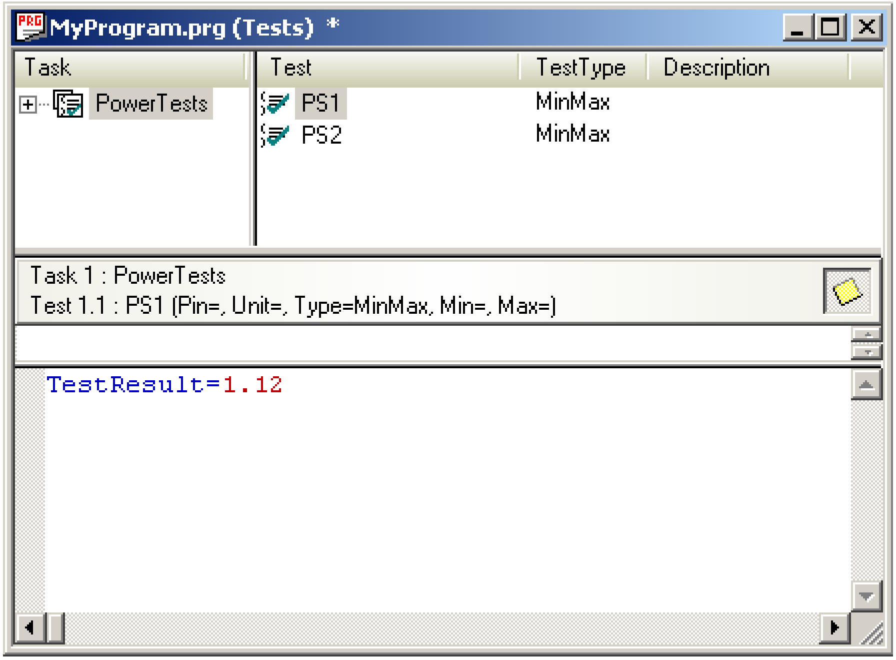
Normally, test code contains code to set up the switching and measurement instruments. A measurement will be taken and then assigned to **TestResult** as shown in the following example:

```
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:



Test	TestType	Description
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.

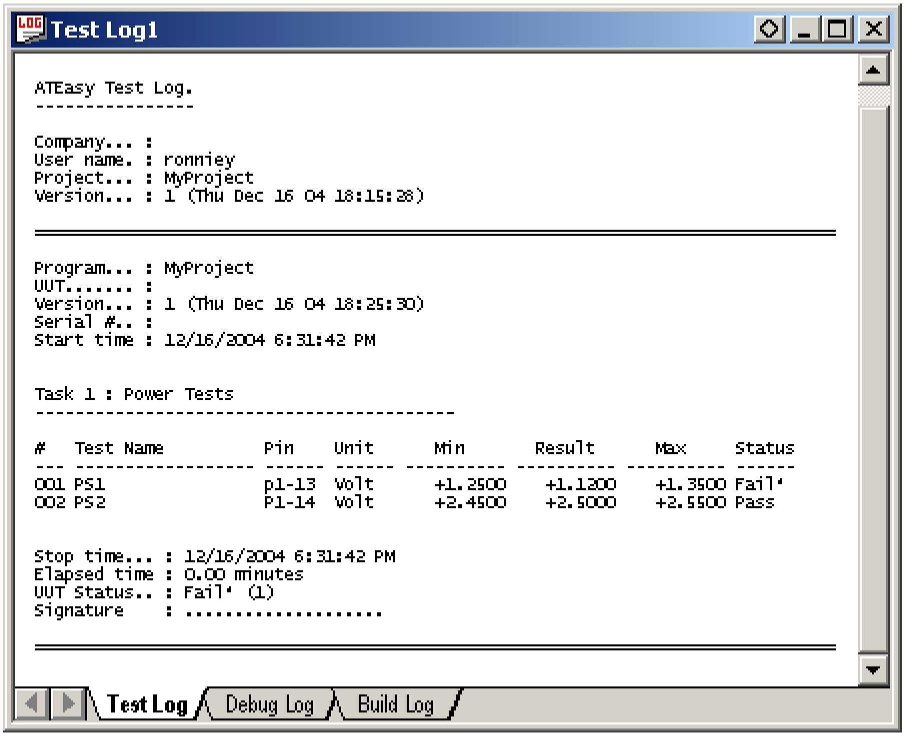
Other debugging commands are available from the **Debug** menu. These provide code level debugging. Using debug commands you can use the **Toggle Breakpoint** command to pause on a specific line of code. You can use the **Step Into**, **Step Over**, and **Step to Out** to walk through the lines of code.

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.

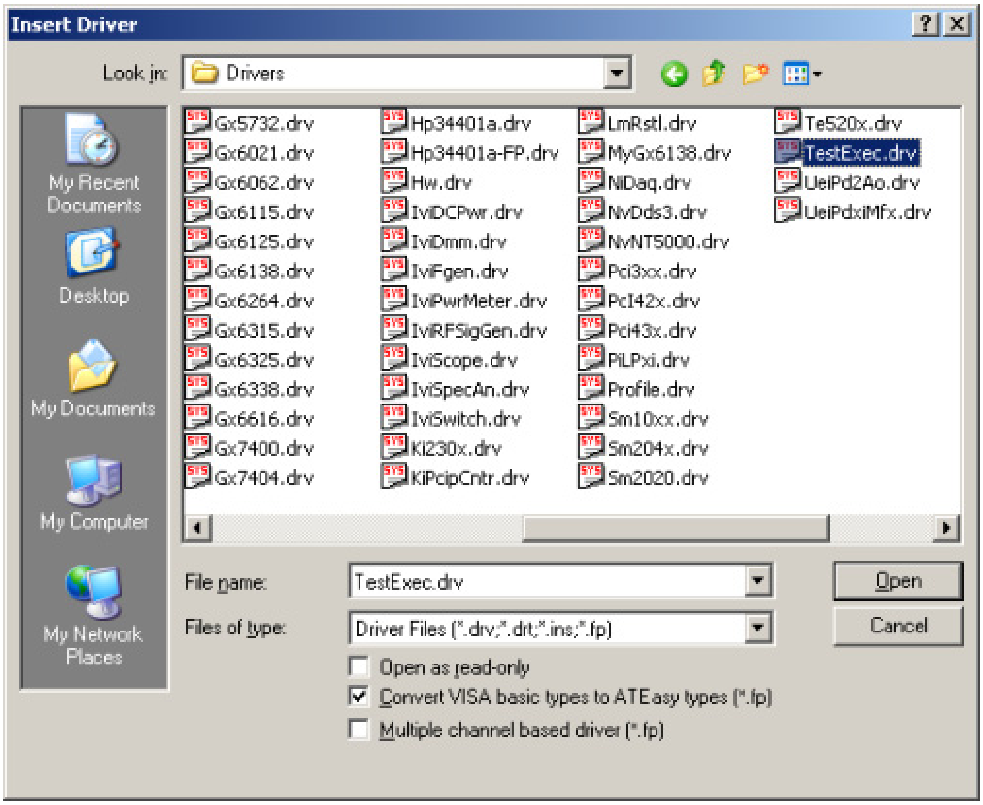
Scroll to the right to see that PS1 has a status of **Fail**, while PS2 has a status of **Pass**.

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:



Insert Driver

Look in: Drivers

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.

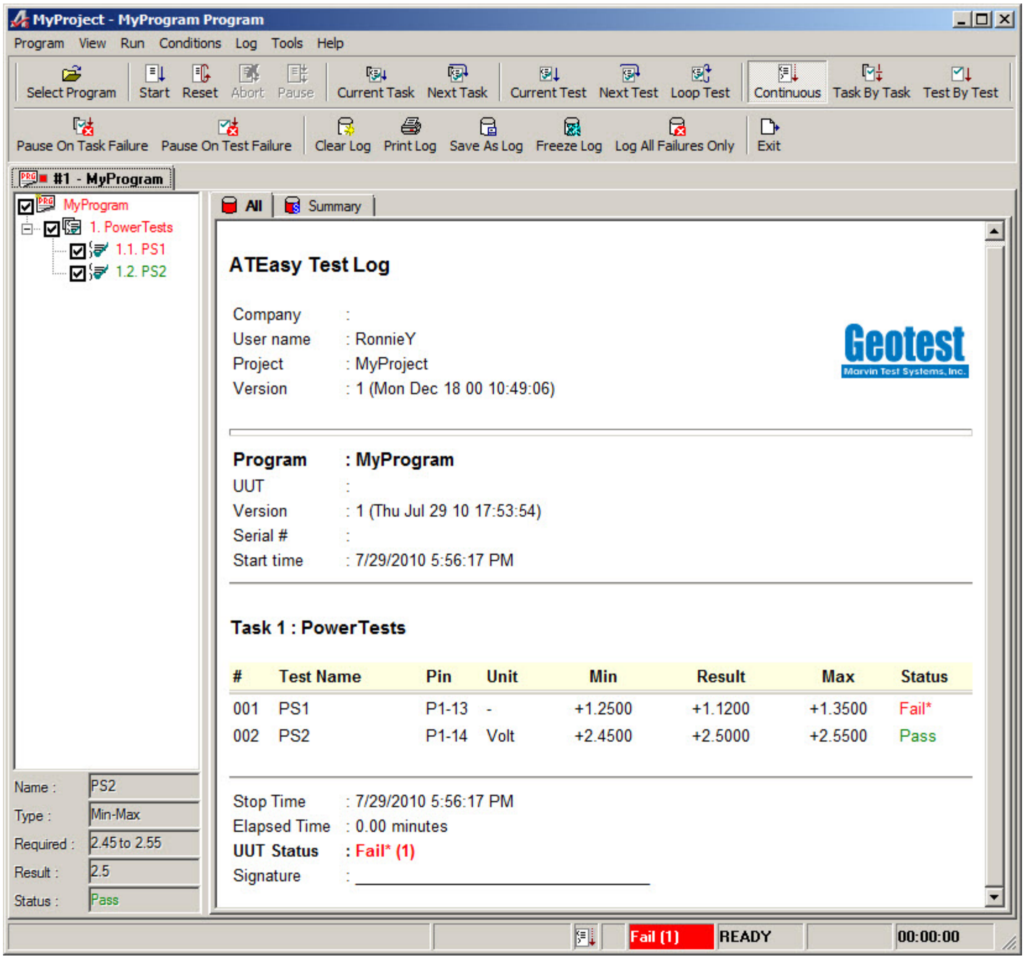
Using the Test Executive Driver

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:



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 : ______________________

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

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.

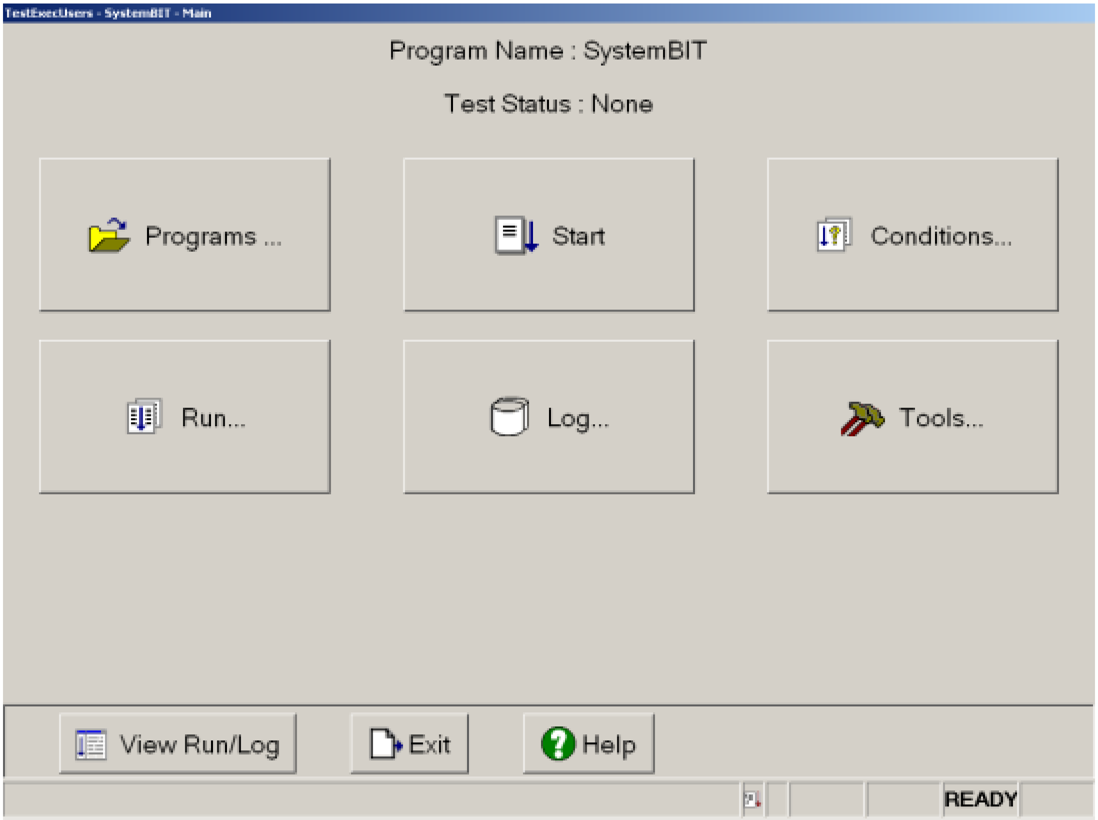
You have now created and tested a very simple application. Adding the test executive driver to the application allows you to provide a test executive to your application quickly.

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:



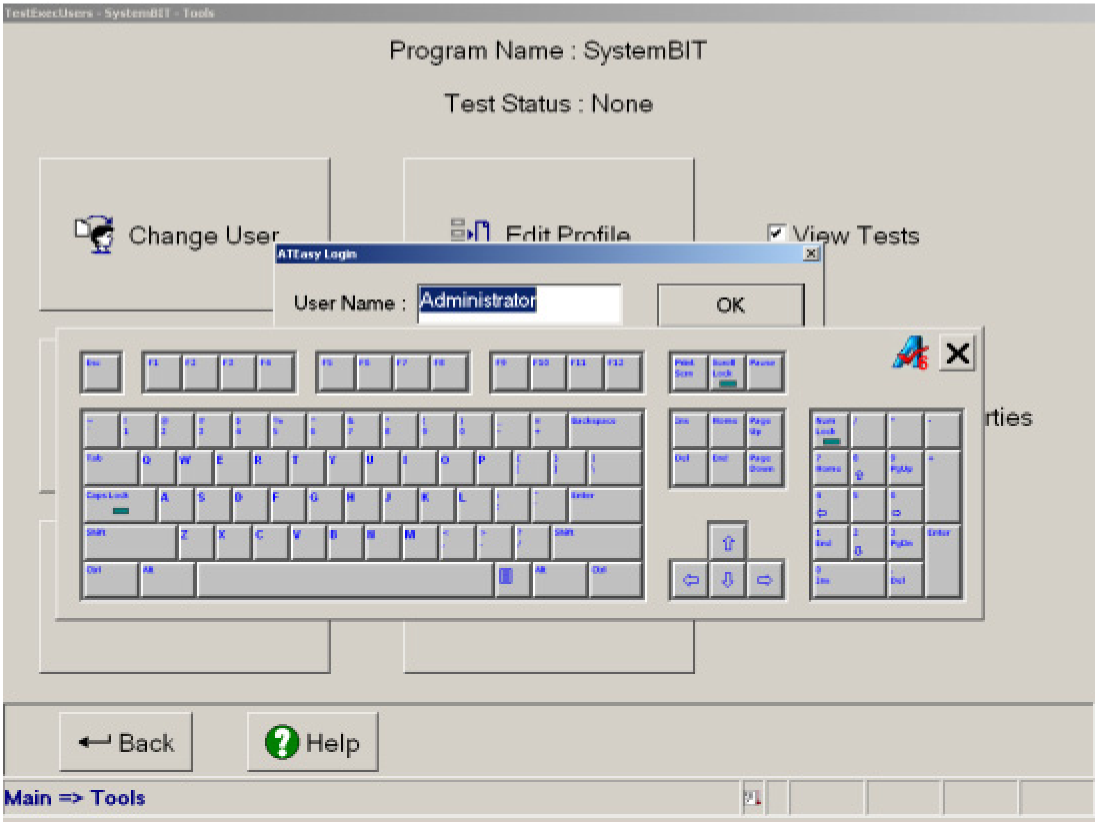
TestExecUsers - SystemBIT - Main

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:



TestExecUsers - SystemBIT - Tools

Program Name : SystemBIT

Test Status : None

User Name : Administrator

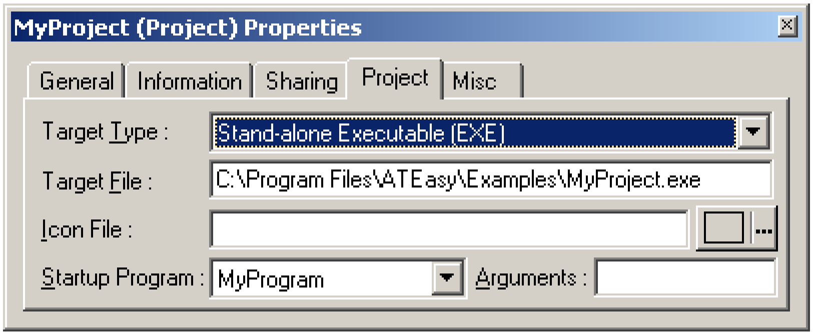
Main => Tools

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:



MyProject (Project) Properties

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:

1. Select the TestExec driver, in the Workspace window.
2. Select **Delete** from the **Edit** menu. The driver is now removed from the system.
3. Click **Save All** from the Standard toolbar to save your work.
4. You will be prompted to enter a file name for a workspace file. Type **C:\MyProject\MyWorkspace** and click **OK**.

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

CHAPTER 5 – 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: **i**, **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 9 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, _.

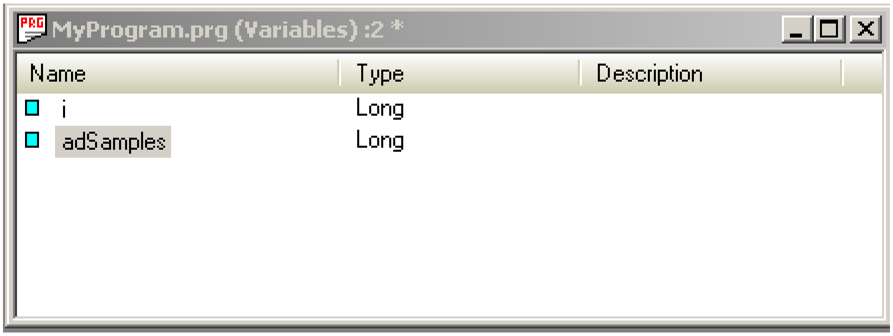
Declaring Variables:

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 **adSamples**.

Your screen should now look similar to the following:



MyProgram.prg (Variables) :2 *

Name	Type	Description
i	Long	
adSamples	Long	

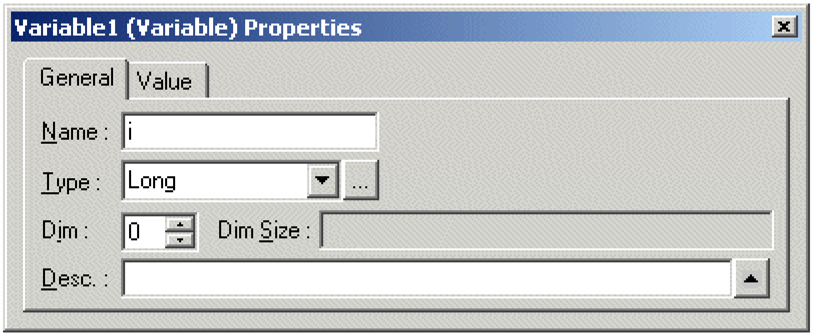
Variable Properties

The next step is to set the properties of variables that you defined. The first variable, **i**, is declared as Long. The second variable, **adSamples**, will be declared as a one-dimensional array holding 20 elements of type Double.

▼ 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.

3. Leave the Properties window open and select the next variable **adSamples**. 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 **adSamples** and type **Array of 20 samples** in the description field.

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

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.

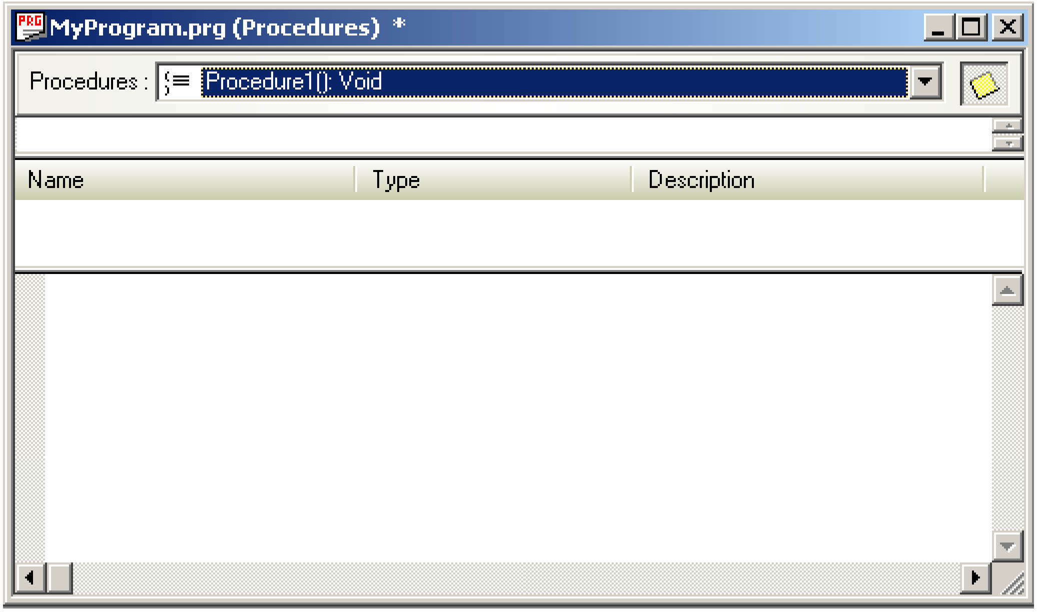
- **DLL** – These are procedures residing in, and exported from, an external library (DLL). You can define and call them in ***ATEasy***.

- **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:



Procedures : Procedure1(): Void

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.

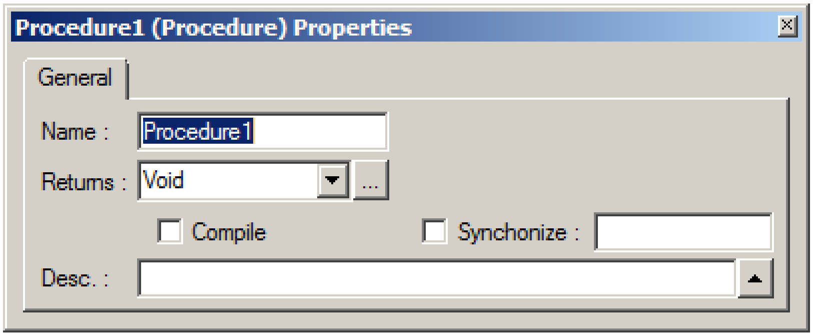
Procedure1 is the newly created procedure. It is the current procedure. The return type was set to **Void** indicating the procedure does not return any value and is therefore a **subroutine**.

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:



Procedure1 (Procedure) Properties

General

Name : Procedure1

Returns : Void

□ Compile □ Synchronize :

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 **Average**.

3. Select or type **Double** from the Returns combo box to set the return value.

4. Type the following description: **Calculates an average value 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, **Average,** has two parameters: **ad,** 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 **ad** 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 **ad** parameter and select **Properties** . Set the following properties:

Name:	**ad**
Parameter:	**Val**
Type:	**Double**
Dim:	**1**
Desc.:	**Array to calc average**

5. 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.

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

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

7. 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:

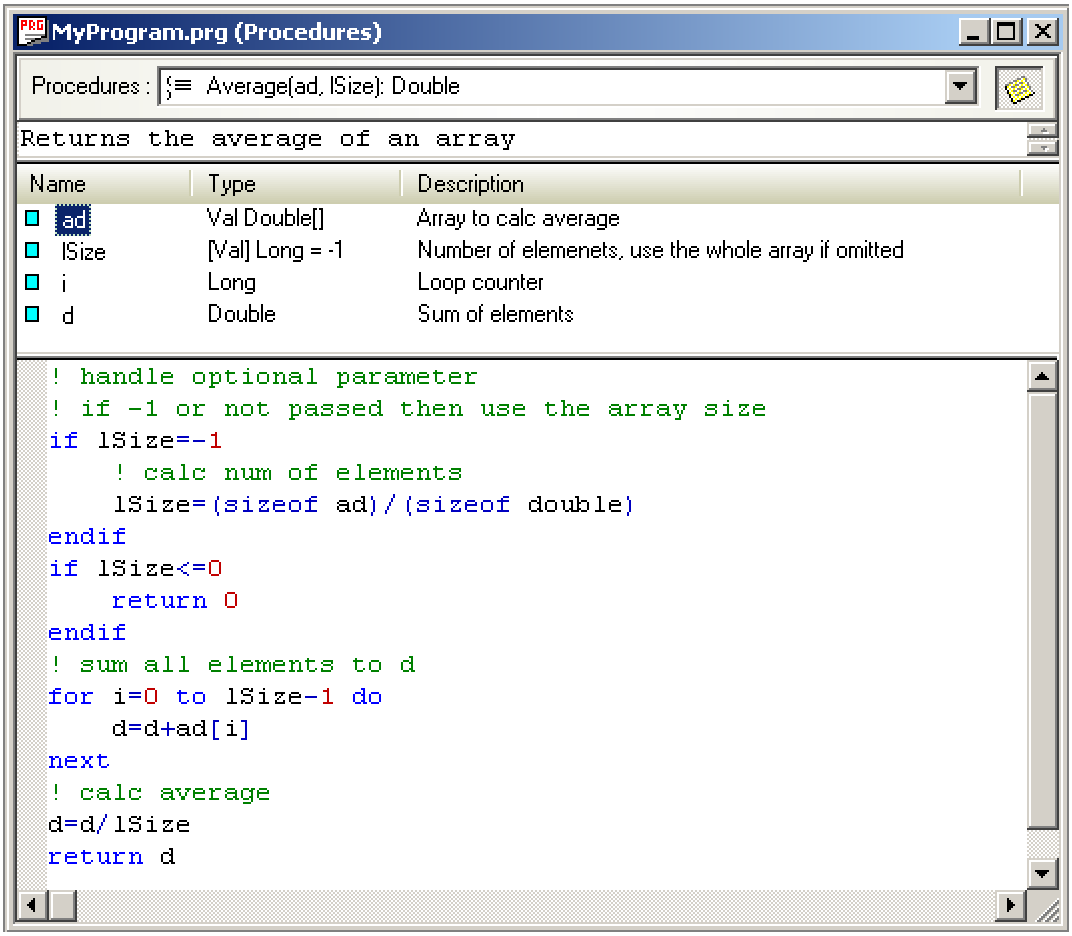
1. Type the following statements in the procedure code area:

```
! handle optional parameter
! if -1 or not passed then use the array size
if lSize=-1
      ! calc num of elements
      lSize=(sizeof ad)/(sizeof double)
endif
if lSize<=0
      return 0
endif
! sum all elements to d
for i=0 to lSize-1 do
      d=d+ad[i]
next
! calc average
d=d/lSize
return 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:



Procedures : Average(ad, lSize): Double

Returns the average of an array

Name	Type	Description
ad	Val Double[]	Array to calc average
lSize	[Val] Long = -1	Number of elemenets, use the whole array if omitted
i	Long	Loop counter
d	Double	Sum of elements

```
! handle optional parameter
! if -1 or not passed then use the array size
if lSize=-1
    ! calc num of elements
    lSize=(sizeof ad)/(sizeof double)
endif
if lSize<=0
    return 0
endif
! sum all elements to d
for i=0 to lSize-1 do
    d=d+ad[i]
next
! calc average
d=d/lSize
return d
```

Calling the Procedure from a Test

To see if the average procedure you just created works, create a test. In the test, set 20 elements in an array with values ranging from 1 to 20. Then, call the Average procedure. The procedure should return 10.5, which is the average of the filled array. Use a precise test type with a required value set to 10.5.

▼ To write a test using the Average procedure:

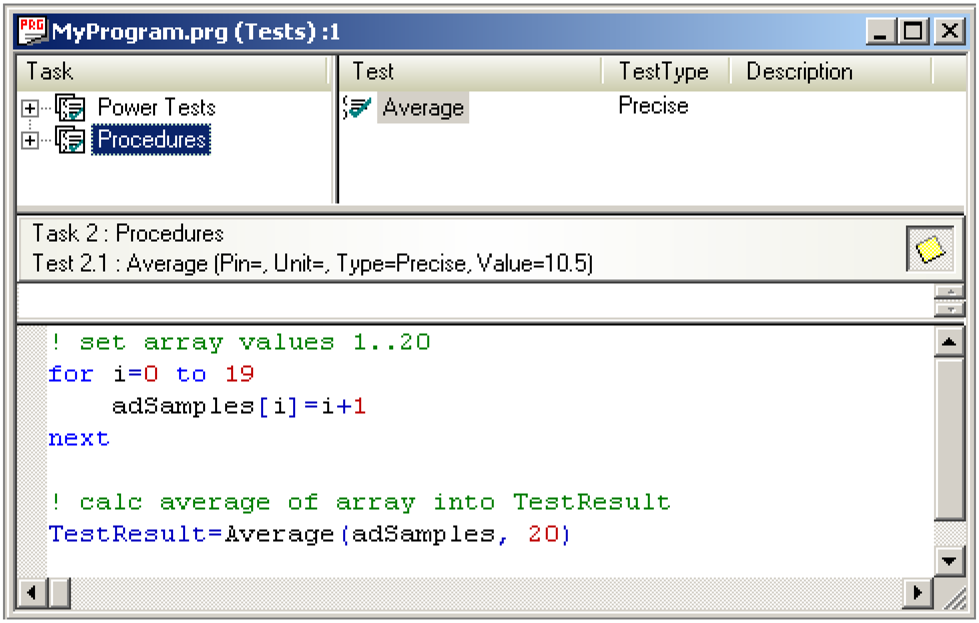
1. Double-click on the **Tests** submodule below MyProgram in the Workspace window. The tests view should appear in a new document view.
2. Right-click on the Power Tests task, and select **Insert Task After** . A new task with a test is inserted.
3. Rename the task to **Procedures** by typing in the edit box or pressing **F2** and typing.
4. Rename the **Untitled Test** to **Average** by clicking on the Untitled Test text and typing **Average**.
5. Right-click on Average and select **Properties** . Change the test type to **Precise** and set the value to be **10.5**.
6. Now enter the following code in the code pane:

```
! set array values 1..20
for i=0 to 19
      adSamples[i]=i+1
next

! calc average of array into TestResult
TestResult=Average(adSamples, 20)
```

The first three statements in this example are a simple For-Next loop filling the array, **adSamples**. The next line calls the average statement and sets the result to be the test result. Note that you *could* call the Average procedure by omitting the second optional argument (**TestResult=Average(adSamples)**), since you are calculating the average of the entire array.

At run-time, after the test is completed, ***ATEasy*** will use that 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 : Average (Pin=, Unit=, Type=Precise, Value=10.5)

```
! set array values 1..20
for i=0 to 19
    adSamples[i]=i+1
next

! calc average of array into TestResult
TestResult=Average(adSamples, 20)
```

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.

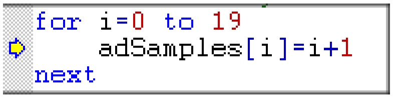
Other debugging commands and windows are available from the **Run** and **Debug** menus and the **View** menus.

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 Average 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:

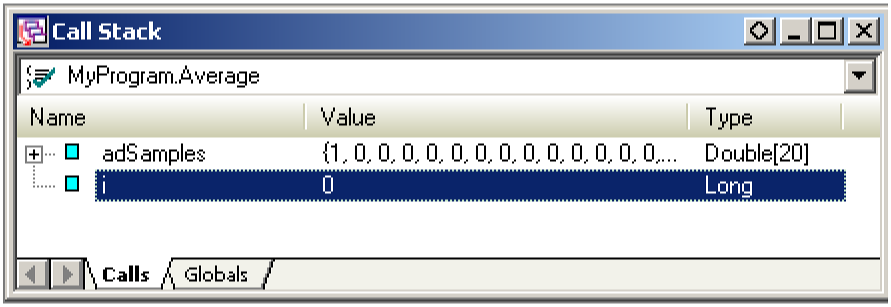


```
for i=0 to 19
    adSamples[i]=i+1
next
```

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 **adSamples** 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:



Call Stack

MyProgram.Average

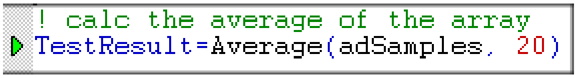
Name	Value	Type
adSamples	{1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,...	Double[20]
i	0	Long

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 Average 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 20. 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 Average procedure and the second one the Average test.

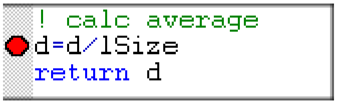
8. Select the second item in the Call Stack combo box. The Test is shown. Notice the green triangle mark next to the line that called your procedure. This mark is the **Call Mark** and it shows the line that called your procedure as shown here:



```
! calc the average of the array
TestResult=Average(adSamples, 20)
```

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

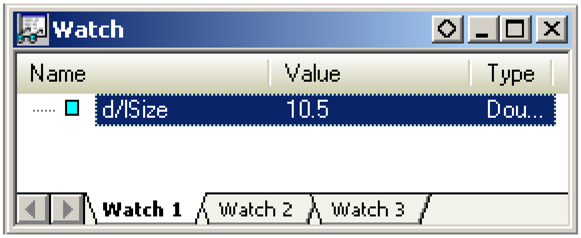
10. Set the insertion point in the line containing the division of **d** with **lSize**. Select the **Toggle Breakpoint** command from the **Debug** Menu or from the Standard toolbar. A red **Breakpoint Mark** will appear next to the line as shown here:



```
! calc average
d=d/lSize
return d
```

11. 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.

12. 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 **10.5** as shown here:



Watch

Name	Value	Type
d/lSize	10.5	Dou...

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

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 6 – 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 GPIB 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 GPIB instrument using the HP34041 Digital Multimeter. 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 Tables	What are I/O Tables?
Creating a SetFunctionVDC I/O Table	How to create an I/O Table.
Creating a SetFunctionVAC I/O Table	How to create a second type of 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.
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.
Using Function Panel Drivers	How to use .fp drivers.

Topic	Description
Using IVI Drivers	How to use the ***ATEasy*** drivers for IVI based instrument drivers

Interfaces, Interface Types, and Drivers

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

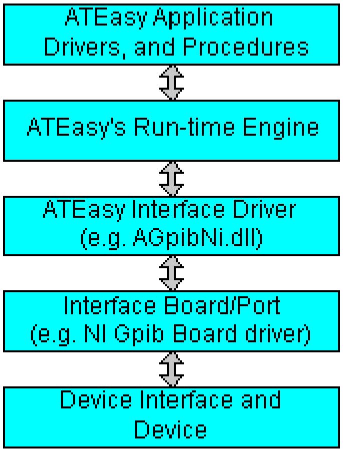
ATEasy supports two types of interfaces: internal (built-in) and external. **Internal** interfaces are built into your machine and do not require any special software or configuration. These interface types include:

- **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.
- **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.

ATEasy's applications are not dependent on the interface vendor. For example, replacing a National Instrument's GPIB board with HP's GPIB board will not result in changes in the application. In addition, when using IO tables to communicate with the device, drivers can be made interface type independent, so one driver can be written to support more than one interface type (for example, GPIB and VXI). Once the driver is added to the system, a driver shortcut is created and configured to contain the currently used interface and the interface address.


ATEasy Application Drivers, and Procedures
ATEasy's Run-time Engine
ATEasy Interface Driver (e.g. AGpibNi.dll)
Interface Board/Port (e.g. NI Gpib Board driver)
Device Interface and Device

Adding an Interface

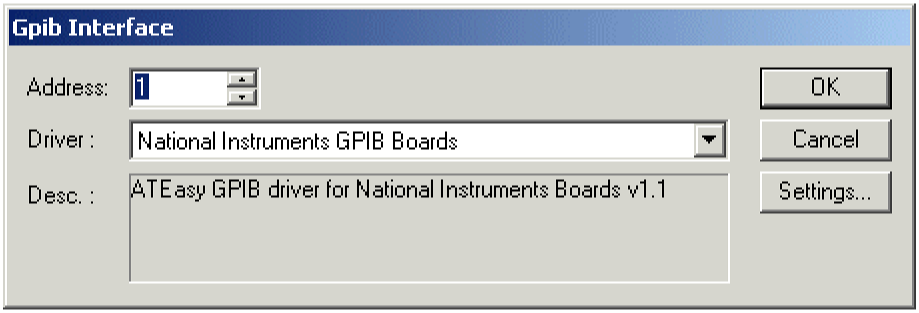
Before ***ATEasy*** can access external interfaces on your computer, they have to be defined and configured. These interfaces include GPIB and VXI. Other interfaces, such as COM ports, are built-in and do not require any further action.

Before defining an interface, you must install the interface board in your computer and install the vendor software driver. ***ATEasy*** uses the vendor driver in order to configure and use the interface board.

In this example, we will be using the National Instruments GPIB board. If you have a different type of board, the choices and displays may be slightly different.

▼ To define a National Instrument GPIB interface:

1. Select **Options** from the **Tools** menu.
2. Select the **Interfaces** page.
3. Select **Gpib** from the interface list and click **Add**. The Gpib Interface dialog appears.
4. Set the Address to be **1** (default). This address will be used by your application to access the GPIB Board. Select the GPIB interface vendor. In our example, select **National Instruments** from the combo box as shown here:



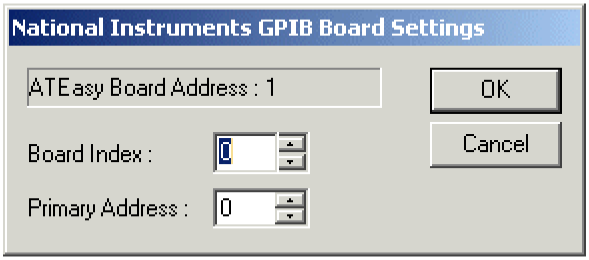
Address: 1

Driver : National Instruments GPIB Boards

Desc. : ATEasy GPIB driver for National Instruments Boards v1.1

Note: If the Description text does not indicate the driver version as shown here, an error message will display in the text box. If an error is displayed, make sure the vendor driver is installed properly and resides in the ***ATEasy*** folder, Windows or Windows System folder, or in the Windows PATH directories.

5. Click **Settings**. The National Instruments GPIB Boards dialog box appears as shown below:



ATEasy Board Address : 1

Board Index : 0

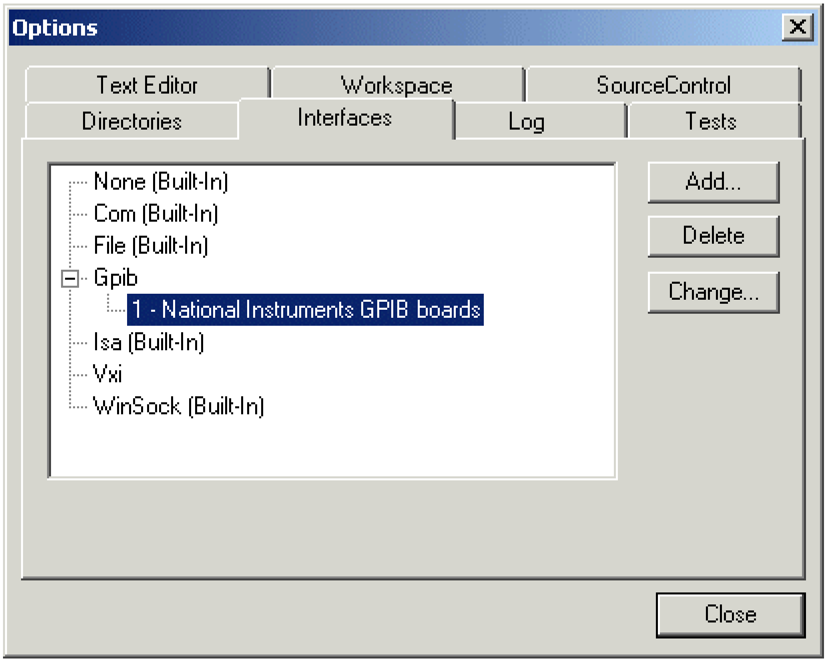
Primary Address : 0

This is a vendor specific dialog box.

For National Instruments, set the board index as configured by the National Instrument GPIB control panel applet: **0** for GPIB0, **1** for GPIB1, etc. If you have only one GPIB board from National Instruments, this is set to 0 by default. The primary address is the device GPIB address that ***ATEasy*** uses to access the board. Typically, it is set to 0, unless you have a device already using that address.

Click **OK** to close this dialog.

6. Click **OK** again to return to the Interfaces dialog box. Your screen should now look like the following:



None (Built-In)
Com (Built-In)
File (Built-In)
Gpib
1 - National Instruments GPIB boards
Isa (Built-In)
Vxi
WinSock (Built-In)

7. Click **Close** to return to the IDE main window. Your interface is now configured and ready to use.

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. In this example, you will be developing a new driver for the HP34401 Digital Multimeter GPIB instrument. For reference, you will add the HP34401 driver, which is supplied with ***ATEasy***, to your system.

▼ 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.

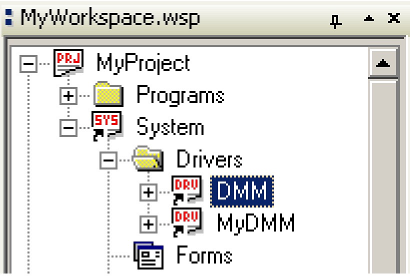
3. Select **HP34401a.drv** and click **Open**. ***ATEasy*** loads the driver and names a **driver shortcut** as **DMM**. This driver is for a Hewlett-Packard GPIB based Digital Multimeter (DMM). The document view for the driver opens in the client area.

▼ To create a new driver in the System:

1. 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.

2. Click on the **Save All** command from the file menu and save the new driver (Driver1) to **MyDMM** in the MyProject folder. Notice that ***ATEasy*** renamed the shortcut from Driver1 to MyDMM.

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.

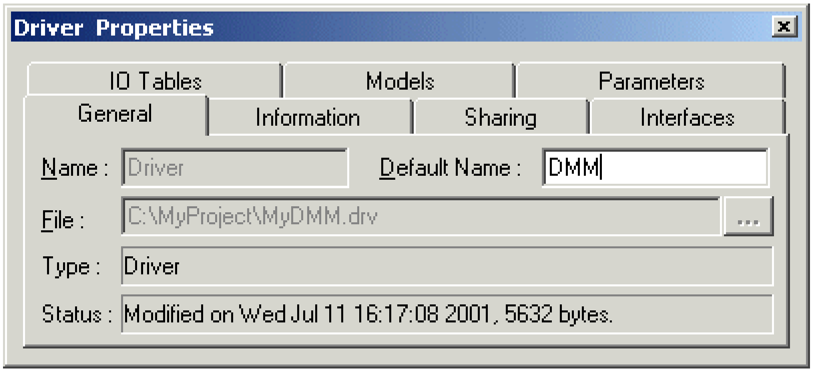
Changes made to the driver will be saved in the driver file, while changes made to the driver shortcut are saved in the System module.

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



Driver Properties

Name : Driver

Default Name : DMM

File : C:\MyProject\MyDMM.drv

Type : Driver

Status : Modified on Wed Jul 11 16:17:08 2001, 5632 bytes.

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

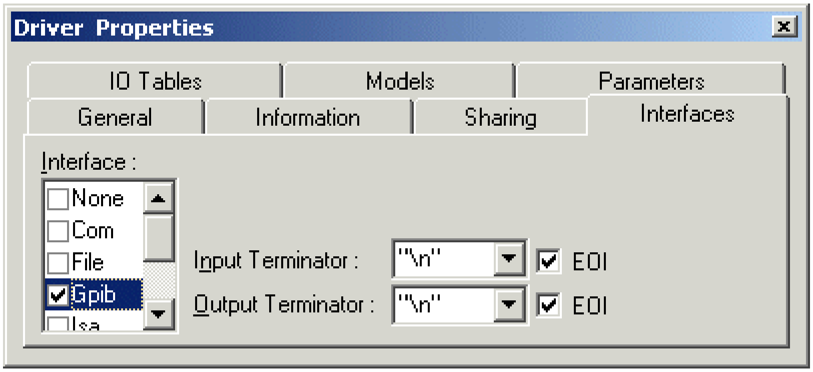
Defining the Driver Interface

The driver interface to be created is a GPIB driver. You will be adding this interface to the **MyDMM** 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 **Gpib** Interface in the list box to add GPIB support to this driver. As shown below, select the **LF** (Line Feed or **"\n"**) for the **Input** and **Output Terminator**. Also, select **EOI** to identify the end of transmission.
3. Uncheck the **None** interface, to make the GPIB interface the only interface supported by this driver.

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



Interface :
□ None
□ Com
□ File
☑ Gpib
□ Isa

Input Terminator : "\n" ☑ EOI

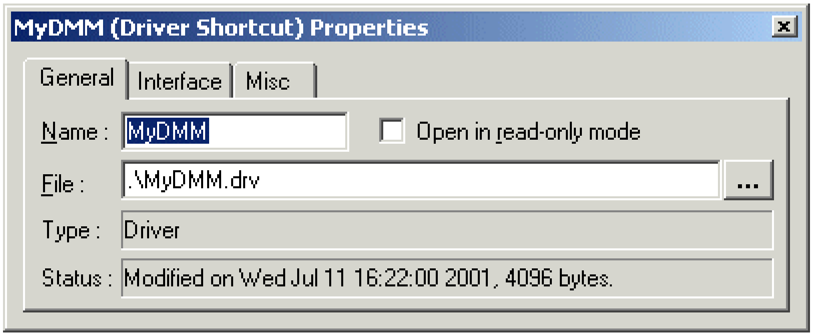
Output Terminator : "\n" ☑ EOI

Configuring the Driver in the System

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

▼ To configure the driver in the system:

1. Select the **MyDMM** 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:



MyDMM (Driver Shortcut) Properties

Name : MyDMM

□ Open in read-only mode

File : .\MyDMM.drv

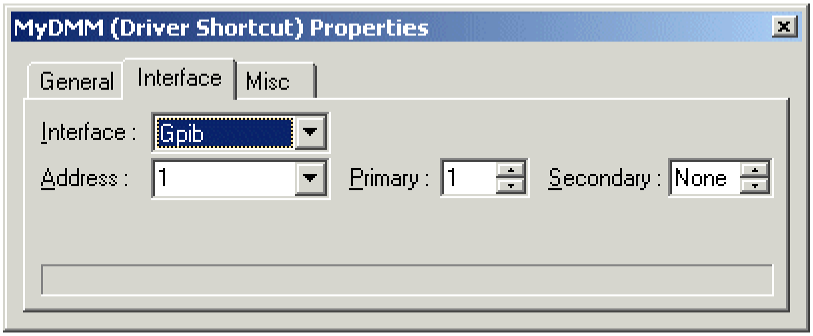
Type : Driver

Status : Modified on Wed Jul 11 16:22:00 2001, 4096 bytes.

2. Select the **Interface** page.

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

4. Change the address to **1**. This indicates that the driver is configured to use the board address 1. Set the GPIB **Primary** address to **1** and the **Secondary** address to be **None** (no secondary is used). Your screen should now look like the following:



MyDMM (Driver Shortcut) Properties

Interface : Gpib

Address : 1

Primary : 1

Secondary : None

At this point, your driver is configured and ready to send and receive data from the GPIB interface board to GPIB address 1 where your instrument resides.

I/O Tables

To communicate with instruments using an Interface such as GPIB, 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, GPIB and RS-232) without the need to write interface-specific code inside the driver.

Creating a SetFunctionVDC I/O Table

I/O Tables perform a variety of functions. The first you will work with is writing data to an instrument. The I/O table you create here will be used to set the DMM to VDC (Volts DC) measurement mode. To do so, you need to send a string to the DMM instructing it to change its measurement mode to VDC. For that, you need one output operation and one send operation.

▼ To create an IO Table:

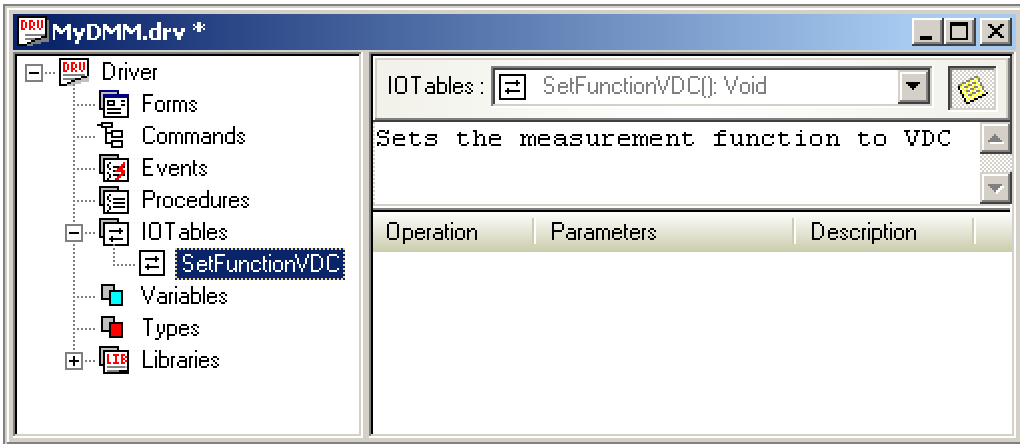
1. Select the **IOTables** submodule from the tree view in the Document View of the MyDMM driver.

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 **SetFunctionVDC**.

4. Type the following description for this operation in the description view: **Sets the measurement function to VDC.**

Your screen should now look similar to the following:



IOTables : SetFunctionVDC(): Void

Sets the measurement function to VDC

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.

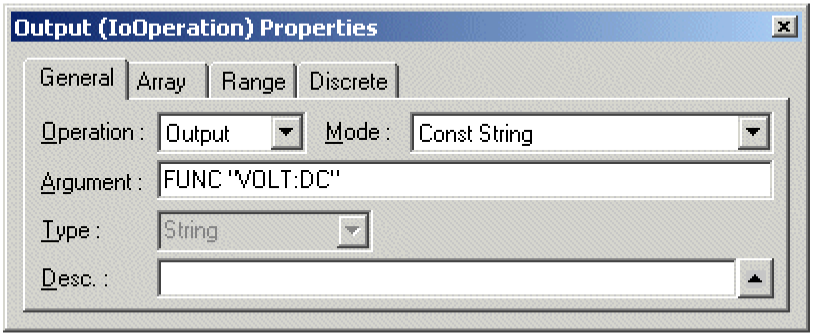
▼ To create the Output operation:

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

2. 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 DMM for VDC mode:

   ```
   FUNC "VOLT:DC"
   ```

   The HP 34401 user's guide specifies that this is the string to be sent in order to set up the instrument for VDC measurement mode. 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 : FUNC 'VOLT:DC"

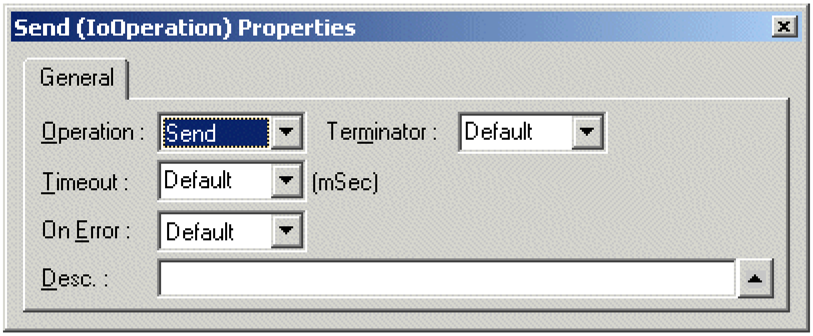
Type : String

Desc. :

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

▼ To create the Send operation:

1. 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.
2. 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

General

Operation : Send

Terminator : Default

Timeout : Default (mSec)

On Error : Default

Desc. :

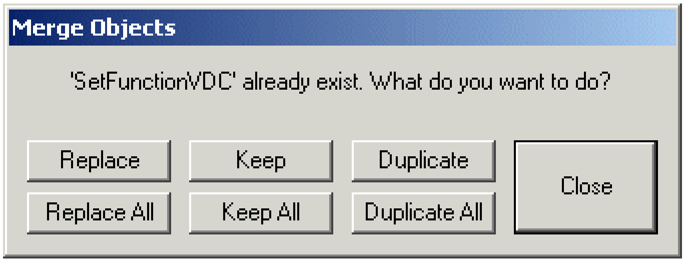
The operations view now shows two operations: Output, followed by the Send operation. Your I/O table is now completed and is ready to use.

Creating a SetFunctionVAC I/O Table

In this section, you will create a similar I/O Table called **SetFunctionVAC**. The SetFunctionVAC 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 SetFunctionVDC I/O Table. Then, you will modify the I/O Table and the Output operation to the VAC functionality.

▼ To create the SetFunctionVAC I/O Table:

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



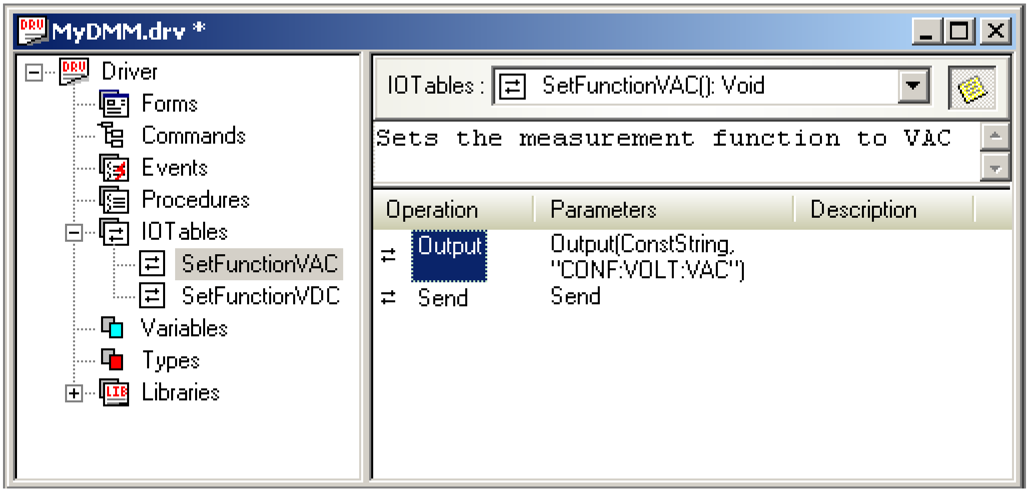
Merge Objects

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

3. Click on the **Duplicate** button. A new I/O Table is created and named **SetFunctionVDC1**.
4. Select the **SetFunctionVDC1** I/O Table and rename it to **SetFunctionVAC**. (Use the **F2** key if you need to.)
5. Type the following description for this table in the description view: **Sets the measurement function to VAC**. (**Hint:** you can just edit VDC to VAC.)
6. Select the Output operation, open the properties window and change the argument to display:

```
CONF "VOLT:AC"
```

When done, the I/O Tables view should be as shown below:



Sets the measurement function to VAC

Operation	Parameters	Description
Output	Output(ConstString, "CONF:VOLT:VAC")	
Send	Send	

If desired, you could now create additional I/O Tables for all the DMM's measurement modes such as current (IAC and IDC), resistance (2-wire and 4-wire), frequency, etc.

Using the Output Discrete Mode

While you could continue to add individual I/O tables for each of the DMM functions, ***ATEasy*** offers another method to further simplify driver development. One of the property pages for an Output operation is *Discrete*. The Discrete mode allows a single I/O Table to accommodate multiple options, which typically require multiple I/O Tables. In this example, you will create a single I/O Table allowing you to set the measurement mode of the DMM to any of the available functions.

▼ To create a new I/O Table using the Discrete mode:

1. Create a new I/O Table, naming it **SetFunction**. Set the I/O Table description to **Sets the measurement function**. For specific steps to create an I/O table, see Creating a SetFunctionVDC I/O Table on page 93.

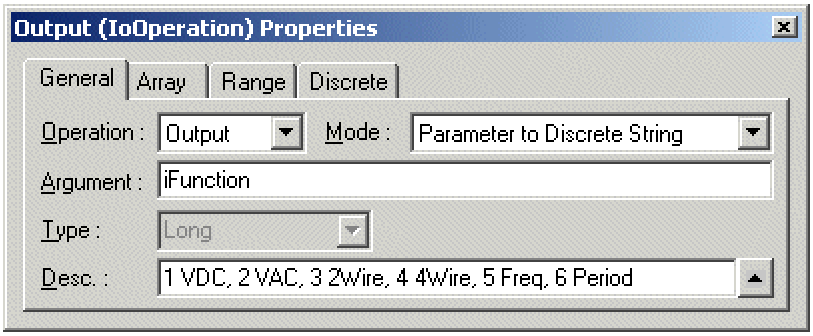
2. Create one Output operation and set the argument to **FUNC "**. No other changes to this operation are required.

3. Add another Output operation and open the properties window for it.

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

5. Set the Argument name to **iFunction**. Enter the description: **1 VDC, 2 VAC, 3 2Wire, 4 4Wire, 5 Freq, 6 Period**

   Your screen should now look similar to the following:



Operation : Output
Mode : Parameter to Discrete String
Argument : iFunction
Type : Long
Desc. : 1 VDC, 2 VAC, 3 2Wire, 4 4Wire, 5 Freq, 6 Period

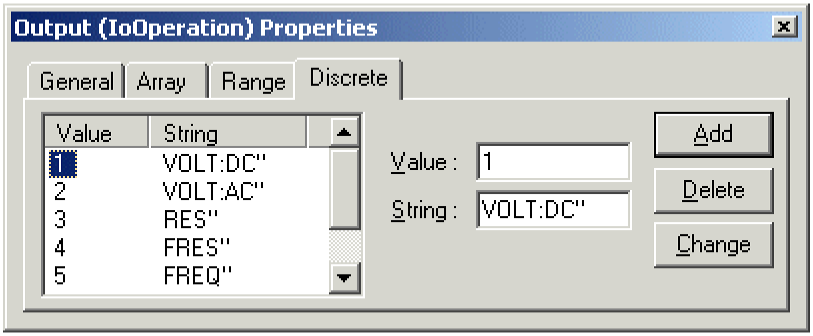
6. Select the **Discrete** page of the Output properties. This page contains a cross-reference table for multiple I/O Table discrete values. You can enter a different string for each value and when the I/O table is called with a specific value, the corresponding string will be sent to the instrument.

7. Enter **1** in the Value field. Type **VOLT:DC”** in the String field. Click **Add**.

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

Value	**String**	**Used for**
2	**VOLT:AC”**	VAC measurements
3	**RES”**	2Wire measurements
4	**FRES”**	4Wire measurements
5	**FREQ”**	Frequency measurements
6	**PER”**	Period measurements

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

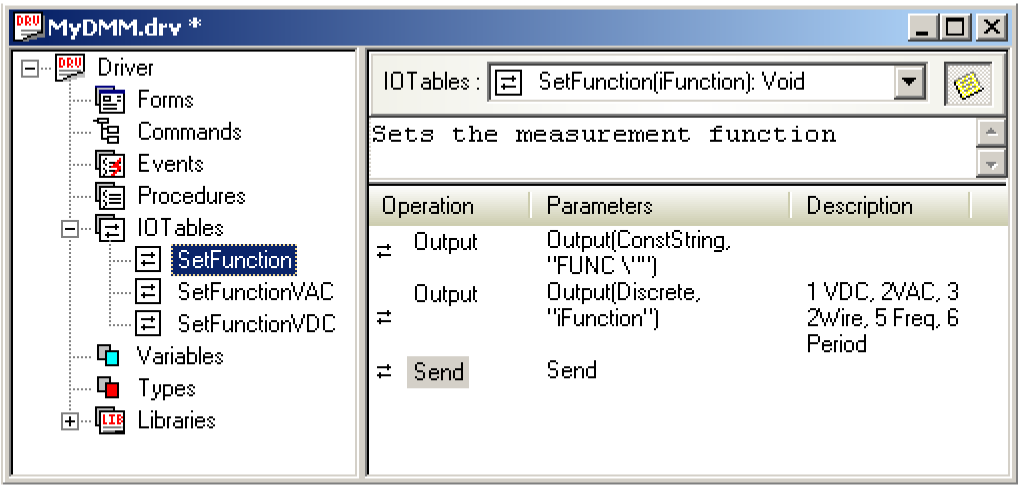


Value	String
1	VOLT:DC"
2	VOLT:AC"
3	RES"
4	FRES"
5	FREQ"

Value : 1
String : VOLT:DC"

9. Add another Send operation.

Your document view should now look similar to the following:



IOTables : SetFunction(iFunction): Void

Sets the measurement function

Operation	Parameters	Description
⇄ Output	Output(ConstString, "FUNC \"")	
⇄ Output	Output(Discrete, "iFunction")	1 VDC, 2VAC, 3 2Wire, 5 Freq, 6 Period
⇄ Send	Send	

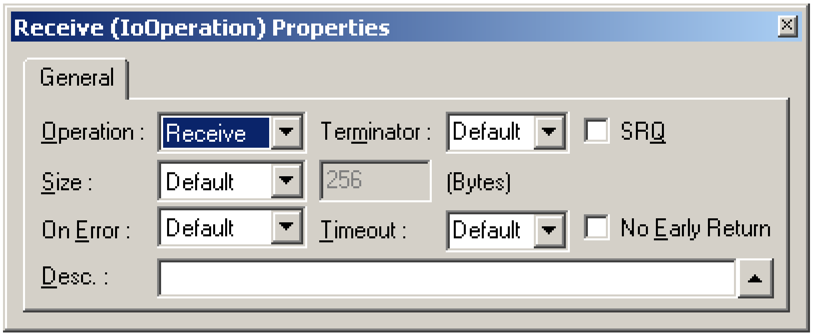
You have created a single I/O table to handle all the DMM measurement functions. When this I/O Table is called with a parameter of 1, the DMM will be set to VDC; when 2 is the argument, the DMM will be set to VAC; and so forth.

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 measure and read data:

1. Create a new I/O Table and name it **Measure**.
2. Create an **Output** Operation and enter **READ?** as the argument.
3. Create a **Send** Operation. These two operations direct the DMM 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:

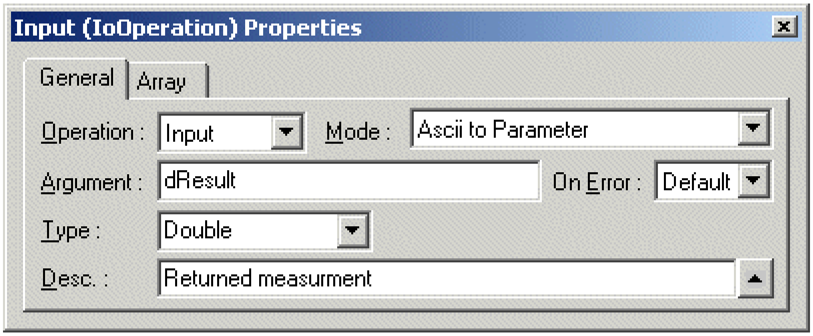


Receive (IoOperation) Properties

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 **dResult** in the Argument. The parameter type should be set to **Double** by default. Enter the description of the parameter as: **Returned measurement**. The Input properties window should look similar to the following:



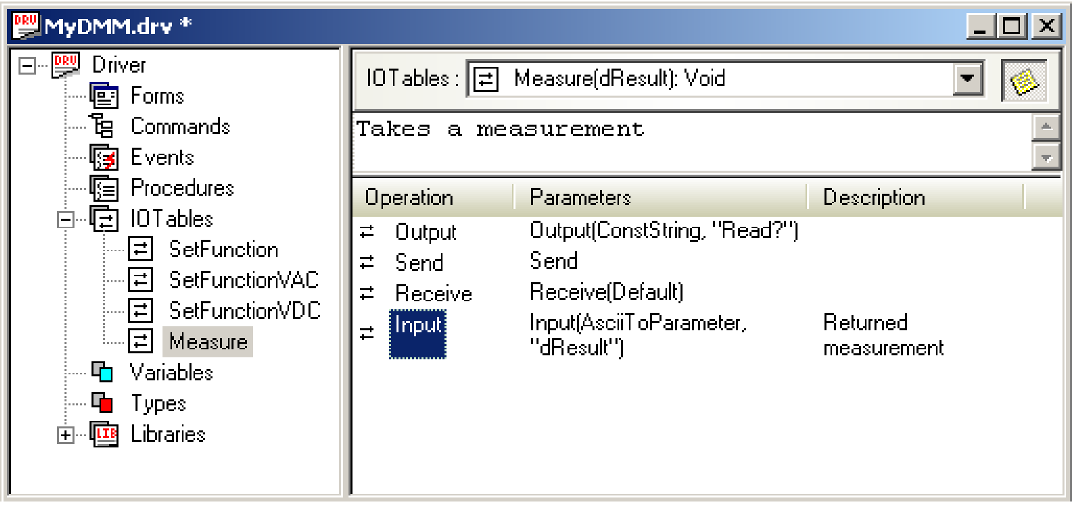
Input (IoOperation) Properties

General | Array

Operation : Input
Mode : Ascii to Parameter
Argument : dResult
On Error : Default
Type : Double
Desc. : Returned measurment

This I/O Table is now complete. When called, ***ATEasy*** first sends a string (:READ?) to the DMM and then reads back data and converts the data from ASCII to the parameter dResult of type double.

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 : Measure(dResult): Void

```
Takes a measurement
```

Operation	Parameters	Description
Output	Output(ConstString, "Read?")	
Send	Send	
Receive	Receive(Default)	
Input	Input(AsciiToParameter, "dResult")	Returned measurement

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:

1. Right-click on the SetFunctionVDC, and select **Properties** from the context menu.
2. Check the **Public** checkbox. This will make the I/O table visible to other modules.
3. 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.

```
SetFunctionVDC()
```

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

```
MyDMM.SetFunctionVDC()
```

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 VDC function.

```
SetFunction(1)
```

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

```
MyDMM.Measure(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 GPIB, RS-232, VXI, WinSock, File IO, and more.

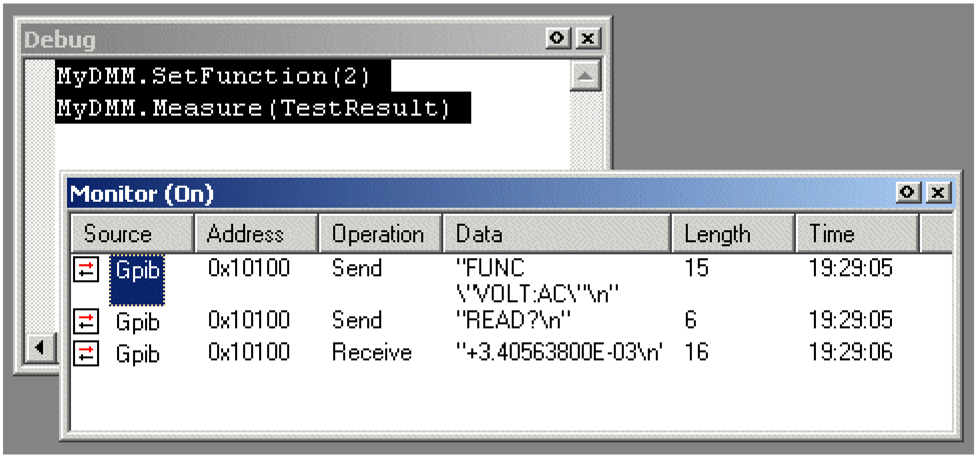
▼ To use the Monitor window:

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 monitor displays information, open the Debug Window from the View menu and type **MyDmm.SetFunction(2)** followed by a line with **MyDMM.Measure(TestResult)**.

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



```
MyDMM.SetFunction(2)
MyDMM.Measure(TestResult)
```

Source	Address	Operation	Data	Length	Time
⇄ Gpib	0x10100	Send	"FUNC \"VOLT:AC\"\n"	15	19:29:05
⇄ Gpib	0x10100	Send	"READ?\n"	6	19:29:05
⇄ Gpib	0x10100	Receive	"+3.40563800E-03\n'	16	19:29:06

Using VXI Plug&Play Function Panel Drivers

ATEasy allow you to use VXI Plug&Play drivers created by instrument vendors. These instrument drivers support various interface types such as GPIB, VXI, Serial or PC based board and more. The function panel driver has a .fp file extension and can be imported to ***ATEasy*** using the File Open command similar to the way you do when you insert an ***ATEasy*** driver. Once the driver is loaded to ***ATEasy*** you will need to save it to ***ATEasy*** driver (.drv or .drt file).

VISA

Prior to using function panel drivers you will need to install the VISA library. Currently two vendors provide the VISA library: Agilent Technologies (http://www.agilent.com) and National Instrument (http://www.ni.com). The VISA library can be download from their web site. We recommend to use the VISA library from the same vendor that manufacturer the GPIB/VXI board you're using. If no GPIB or VXI board is used then any of these libraries will do. The VISA Library Specification (VPP-4.3) is authored by the VXIplug&play Systems Alliance member companies. You can obtain the specification of the VISA library from the alliance's web site at http://www.vxipnp.org. The VISA specification provides a common standard for the VXIplug&play System Alliance for developing multi-vendor software programs, including instrument drivers. This specification describes the VISA software model and the VISA Application Programming Interface (API). VISA gives VXI and GPIB software developers, particularly instrument driver developers, the functionality needed by instrument drivers in an interface-independent fashion for MXI, embedded VXI, GPIB-VXI, GPIB, and asynchronous serial controllers. VXIplug&play drivers written to the VISA specifications can execute on VXIplug&play system frameworks that have the VISA I/O library.

Function Panel Driver Files

You can obtain and download a Function Panel driver (sometimes refer to as **IVI-C** driver) from the instrument manufacturer web site. Most vendors also register their driver at the IVI Foundation web site http://www.ivifoundation.org under the Driver Registry section. Some vendors such as Agilent or National Instruments carry also third party instruments drivers. After installing the Function Panel driver it's Function Panel file along with other files (such as help file, source files and read me files) will be located in a new sub-folder below the **VXIPNP** folder that hold the instrument name (e.g. Agilent/HP 34401A). Additional driver file will be located in the BIN folder (the driver DLL), LIB and Include (.h C language header file) directories. You may use the driver help file to view the function reference. The DLL file will be used by the ***ATEasy*** driver and must shipped along with your application along with the VISA library.

The following table provides an example to the files copied after loading the HP34401A digital multimeter driver from National Instruments web site on a Windows 2000 machine:

Folder	File	Description
VXIPNP\WINNT\ BIN	hp34401a_32.dll	Describes attributes used by the driver function parameters. This file is provided with in newer function panel drivers that are IVI compatible. Used by ***ATEasy*** only when converting the function panel to ***ATEasy*** driver.
VXIPNP\WINNT\ hp34401a	hp34401a.fp	Describes attributes used by the driver function parameters. This file is provided with in newer function panel drivers that are IVI compatible. Used by ***ATEasy*** only when converting the function panel to ***ATEasy*** driver.
	hp34401a.sub	Describes attributes used by the driver function parameters. This file is provided with in newer function panel drivers that are IVI compatible. Used by ***ATEasy*** only when converting the function panel to ***ATEasy*** driver.
	hp34401a.hlp	Windows help file. Contains reference information about the DLL functions.
	hp34401a.c	Source file for the DLL functions. Not used by ***ATEasy***.
	hp34401a.txt	Read me text file. Contains information about the driver and its files. Not used by ***ATEasy***.
	hp34401a _example.c	C example. Shows how to program the board.
VXIPNP\WINNT\ Include	hp34401a.h	C header file. Includes the DLL functions prototypes. Used by ***ATEasy*** only during the convert process.
VXIPNP\WINNT\ Lib\ bc	hp34401a.lib	Borland C++ library. Not used by ***ATEasy***.
VXIPNP\WINNT\ Lib\ msc	hp34401a.lib	Microsoft VC++ library. Not used by ***ATEasy***.

▼ Converting Function Panel Driver to an ATEasy Driver

1. Select **Open** from the **File** menu
2. Select **Driver files** from the File Open dialog **File of type** drop down list and select the FP file from the VXIPNP driver folder.
3. Click **Open.**

ATEasy will start converting the FP file to ***ATEasy*** driver. During the conversion ***ATEasy*** may need to open other file used by the .FP file, these files may be the C header file .H and the .SUB file (if exist). Other files that may be required are additional header files that are included in C header file, these file are typically installed by the VISA library (e.g. IVI.h or VISA.h) or by your C compiler (if you have one). If ***ATEasy*** is unable to find these files it will prompt you to select the path in which these files reside, if you are unable to find these files select **Cancel**, in that case ***ATEasy*** will try to continue with the convert. After the conversion is complete you may need to save the ***ATEasy*** driver file (drv or drt format).

Before using the driver you will need to configure its parameters as follows:

▼ Configuring the Converted Function Panel ATEasy Driver

1. Open the Driver Shortcut properties window.
2. Click to show the Parameters property page.
3. Select the **ResourceName** string parameter and type in the address of the instrument that you are trying to use. This address is typically looks as "GPIB0::14::INSTR" if the address is GPIB Board #0, primary address 14. The next paragraph explains in details how to address a instrument.
4. Select the **IdQuery** numeric parameter and type 0 if you do not want to perform In-System Verification in the address of the instrument that you are trying to use. Type 1 to verify that the instrument exist when Initialize is called.
5. Select the **ResetOnInit** numeric parameter and set it value to 1 if you want to reset the instrument when the driver **OnInit** is called else set its value to 0.
6. Select the **InitializeOnInit** numeric parameter and set it value to 1 if you want to initialize the driver when the driver **OnInit** is called else set its value to 0.

Specifying the ResourceName Parameter

The **ResourceName** string has the following grammar:

Interface	Grammar	Example
VXI	VXI[board]::VXI logical address[::INSTR]	"VXI0::1::INSTR" - a VXI device at logical address 1 in VXI interface VXI0.
VXI	VXI[board]::MEMACC	"VXI::MEMACC" - board-level register access to the VXI interface.
VXI	VXI[board][::VXI logical address]::BACKPLANE	"VXI::1::BACKPLANE" - mainframe resource for chassis 1 on the default VXI system, which is interface 0.
VXI	VXI[board]::SERVANT	"VXI0::SERVANT" - servant/device-side resource for VXI interface 0.
GPIB-VXI	GPIB-VXI[board]::VXI logical address[::INSTR]	"GPIB-VXI::9::INSTR" - a VXI device at logical address 9 in a GPIB-VXI controlled VXI system.
GPIB-VXI	GPIB-VXI[board]::MEMACC	"GPIB-VXI1::MEMACC" - board-level register access to GPIB-VXI interface number 1.
GPIB-VXI	GPIB-VXI[board][::VXI logical address]::BACKPLANE	"GPIB-VXI2::BACKPLANE" - mainframe resource for default chassis on GPIB-VXI interface 2.
GPIB	GPIB[board]::primary address[::secondary address][::INSTR]	"GPIB::1::0::INSTR" - a GPIB device at primary address 1 and secondary address 0 in GPIB interface 0.
GPIB	GPIB[board]::INTFC	"GPIB2::INTFC" - Interface or raw resource for GPIB interface 2.
GPIB	GPIB[board]::SERVANT	"GPIB1::SERVANT" - Servant/device-side resource for GPIB interface 1.
ASRL	ASRL[board][::INSTR]	"ASRL1::INSTR" - a serial device located on port 1.
TCPIP	TCPIP[board][::LAN device name]::SERVANT	"TCPIP::inst0::SERVANT"-Servant/device-side resource for TCPIP device.

Interface	Grammar	Example
TCPIP	TCPIP[board]::host address[::LAN device name]::INSTR	"TCPIP::dmm2301@ki.com::INSTR" -a TCP/IP device dmm2301located at the specified address.
TCPIP	TCPIP[board]::host address::port::SOCKET	"TCPIP0::1.2.3.4::999::SOCKET" - raw TCP/IP access to port 999 at the specified address.

Comments:

- The VXI keyword is used for VXI instruments via either embedded or MXIbus controllers.
- The GPIB-VXI keyword is used for a GPIB-VXI controller.
- The GPIB keyword can be used to establish communication with a GPIB device.
- The ASRL keyword is used to establish communication with an asynchronous serial (such as RS-232) device.
- The TCPIP keyword is used to establish communication with Ethernet instruments.
- Default value for board is 0.
- Default value for secondary address is none.
- Default value for LAN device name is inst0.

Using the Converted Function Panel ATEasy Driver

Once the driver is inserted to the system and configured you can start using its commands. ***ATEasy*** generates a command for each of the functions that are exported by the FP file. Each command is attached to an ***ATEasy*** procedure, which contains a call the driver DLL procedure. The ***ATEasy*** procedure generated also contains checking of the return value from the DLL procedure, if an error occurs ***ATEasy*** will generate an exception using the **error** statement. This allows you to concentrate on using the driver commands and implement error handling in one place (e.g. in **OnError**) without adding code for error checking after each commands used.

The driver command tree typically contains the following commands:

- **Initialize** - Establishes communication with the instrument. Must be called prior to any other command to obtain the instrument session handle. The driver uses this handle when accessing all other commands by the driver.

- [**Configuration**] - Optional. Contains commands to configure the instrument.
    - Attributes - In IVI based driver Contains attributes to set or get the instrument configuration.
- [**Measurement**] - Optional. Contains commands to return a measurement from the instrument.
- **Utility**
    - **Error Message** - Translates the error return value from a VXIplug&play instrument driver function to a user-readable string.
    - **Error Query** - Queries the instrument and returns instrument-specific error information.
    - **Reset** - Places the instrument in a default state.
    - **Self Test** - Causes the instrument to perform a self-test and returns the result of that self-test.
    - **Revision Query** - Returns the revision of the instrument driver and the firmware revision of the instrument being used.
    - ...
- **Close** - Terminates the software connection to the instrument and de-allocates system resources associated with that instrument.

Additional commands may be available to provide control of the instrument configuration and measurements functions as provided by the driver manufacturer. These commands are documented in the driver help file.

Using IVI drivers

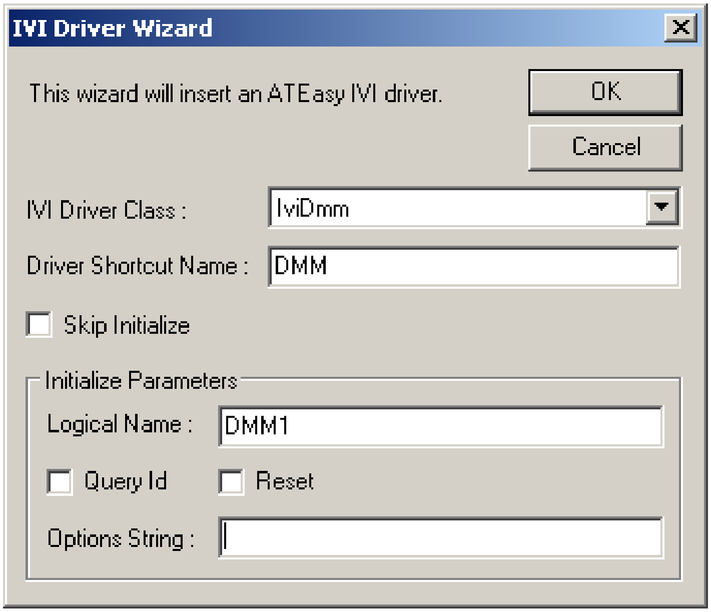
The IVI standard for instrument drivers was created by the IVI Foundation http://www.ivifoundation.org. The foundation defined generic, interchangeable programming interfaces for common instrument classes. Currently the following IVI drivers were released by the IVI foundation: DC power supply, Digital multimeter, Function generator & Arb, Oscilloscope, Power meter, RF signal generator, Spectrum analyzer and Switch. IVI drivers are based on VXI Plug&Play drivers and they require VISA and the IVI libraries that are provided by the IVI foundation members to be installed prior using them. The drivers offer same functions and parameters to different instruments of the same type (i.e. DMM) from different vendors. For example, it allows you to replace Agilent 34401 with a Keithley 2000 DMM in your system without changing your code. ***ATEasy*** provides built in support for theses drivers and provides ***ATEasy*** drivers for all the above IVI drivers.

▼ To Insert an ATEasy IVI driver to your system

1. Before using the driver you must install the instrument manufacturer IVI driver. The IVI engine uses this driver to control the instrument.

2. Configure the driver **address** and its **logical name** using the IVI configuration utility such as National Instruments Measurement & Automation Explorer. The Logical Name which is used to identify the instrument and entered by you (in that utility, the logical name must be displayed below **IVI Drivers/Logical Names** and the manufacturer IVI instrument driver should be link to that logical name and appears under **IVI Drivers/Driver Sessions** which also contain the instrument address, if you don't see **IVI Drivers** in that utility IVI engine is not properly installed). The instrument address is a **VISA resource name** (see earlier in this chapter).

3. Select **Insert IVI Driver…** command from the **Insert** menu. The IVI Driver Wizard is displayed as shown here:



IVI Driver Wizard

This wizard will insert an ATEasy IVI driver.

IVI Driver Class : IviDmm

Driver Shortcut Name : DMM

□ Skip Initialize

Initialize Parameters

Logical Name : DMM1

□ Query Id □ Reset

Options String :

4. Select the IVI class from the list.

5. Type in the driver shortcut name.

6. Type in the instrument Logical Name as explained above.

Programming using ***ATEasy*** IVI drivers is similar to Function Panel imported drivers. ***ATEasy*** provides several examples for the various IVI classes; the IVI workspace file Ivi.wsp contains these examples.

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:	`DMM`	`Set Function VDC`	
	`DMM`	`Measure`	`(dResult)`

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 (DMM). 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.

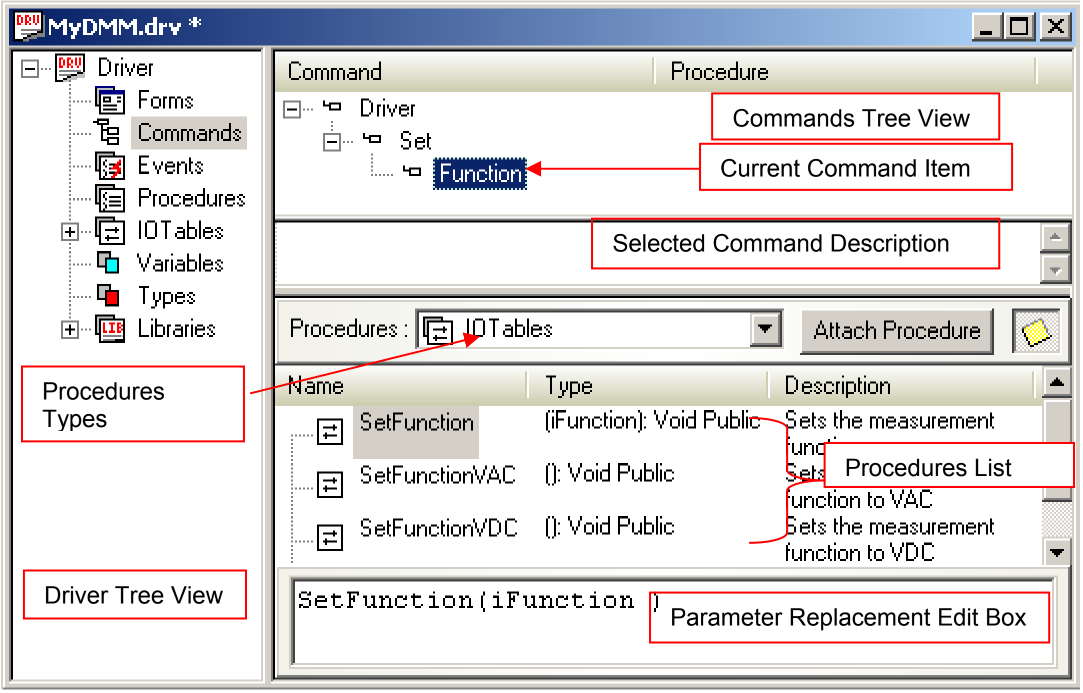
- **Program commands** improve programming by creating language elements specific to a UUT for repeated actions unique to a specific test program. While System and Driver commands are accessible by all modules in a given project, the program statements can only be used within that program. Program commands 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:



Name	Type	Description
SetFunction	(iFunction): Void Public	Sets the measurement func...
SetFunctionVAC	(): Void Public	Sets ... function to VAC
SetFunctionVDC	(): Void Public	Sets the measurement function to VDC

```
SetFunction(iFunction )
```

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.

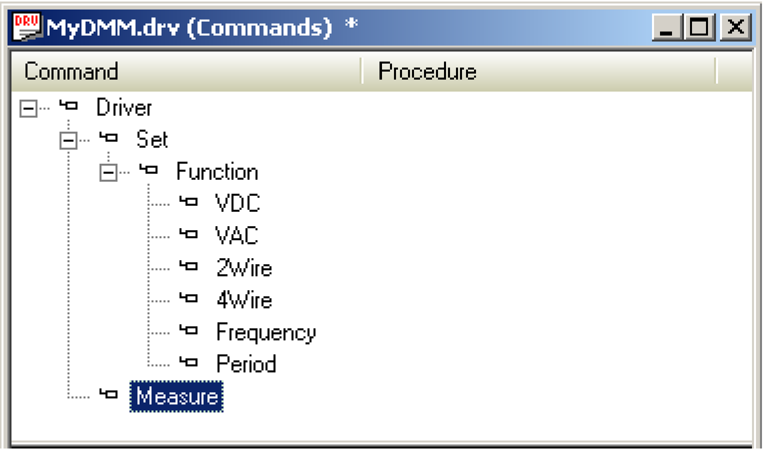
Creating Driver Commands

In Chapter 6, you created several I/O Tables for the DMM driver. Now create Driver Commands for these I/O Tables.

▼ To create driver command items:

1. Double-click on the **Commands** submodule under **MyDMM** 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 **Set** in the edit box to rename the command item.
3. Repeat step 2 and insert a new item below **Set**. Rename it **Function**.
4. Repeat step 2 and insert six items below **Function**. Rename them **VDC, VAC, 2Wire, 4Wire, Frequency,** and **Period.**
5. Create an additional item below **Driver**. Rename it **Measure.**

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


MyDMM.drv (Commands) *

Command	Procedure
Driver	
Set	
Function	
VDC	
VAC	
2Wire	
4Wire	
Frequency	
Period	
Measure	

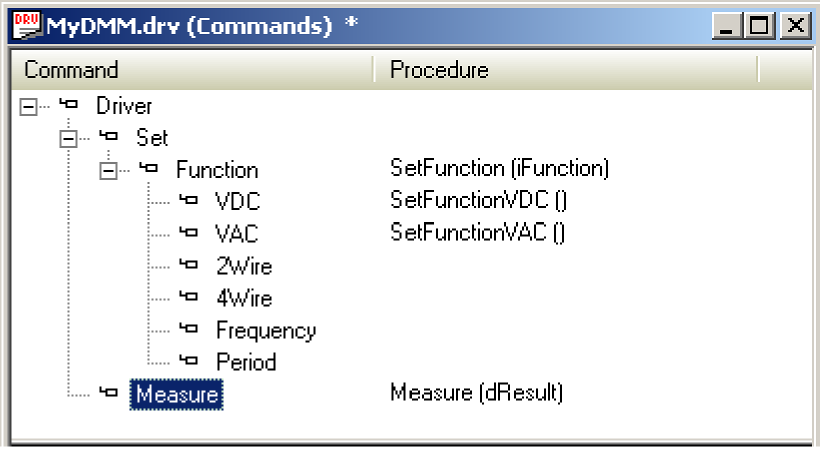
Attaching Procedures and I/O Tables to Commands

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:

1. Select the **Function** command item in the command items view.
2. Select **IO Tables** from the Procedures combo-box. The available I/O tables are displayed in the list below.
3. Select **SetFunction** from the procedures list.
4. Click **Attach Procedure**. The Procedure is now displayed next to the command item in the command items view.
5. Repeat steps three and four for the **VDC**, **VAC** and **Measure** command items.

The commands view should look as shown here:



MyDMM.drv (Commands) *

Command	Procedure
Driver	
Set	
Function	SetFunction (iFunction)
VDC	SetFunctionVDC ()
VAC	SetFunctionVAC ()
2Wire	
4Wire	
Frequency	
Period	
Measure	Measure (dResult)

Four commands were created:

```
Set Function (iFunction)
Set Function VDC
Set Function VAC
Measure (dResult)
```

Replacing Parameters with Arguments

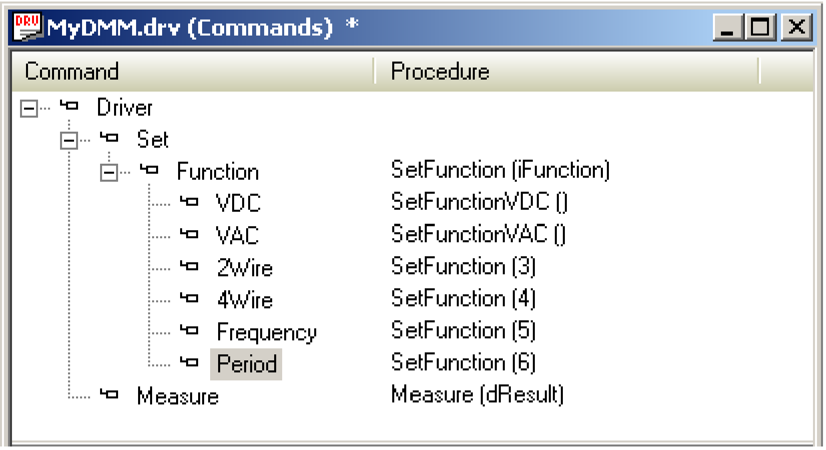
When attaching procedures with parameters to command items, you can replace the parameter with an argument. The argument you specify will be used instead of the parameter and the command will not require the user to supply an argument. Parameters are usually replaced with literals that you supply, but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

In the following example you will use the **SetFunction** I/O Table to implement the remaining commands under Set Function: **2Wire**, **4Wire**, **Frequency,** and **Period**.

▼ To implement the remaining command:

1. Attach **SetFunction** to the **2Wire** command item.
2. Select the **iFunction** text in the Parameter Replacement edit box (at the bottom of the commands view) and type **3** to replace the text. The command items view is automatically updated to display SetFunction (3).
3. Repeat steps one and two for **4Wire**, **Frequency**, and **Period** and use **4**, **5**, and **6** respectively.

The commands view should look as follows:



MyDMM.drv (Commands) *

Command	Procedure
Driver	
Set	
Function	SetFunction (iFunction)
VDC	SetFunctionVDC ()
VAC	SetFunctionVAC ()
2Wire	SetFunction (3)
4Wire	SetFunction (4)
Frequency	SetFunction (5)
Period	SetFunction (6)
Measure	Measure (dResult)

Four commands were created:

```
Set Function 2Wire
Set Function 4Wire
Set Function Frequency
Set Function Period
```

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 MeasureMyDmmVdc(dResult). The procedure will call two commands defined in the MyDMM 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 DMM to implement the procedure.

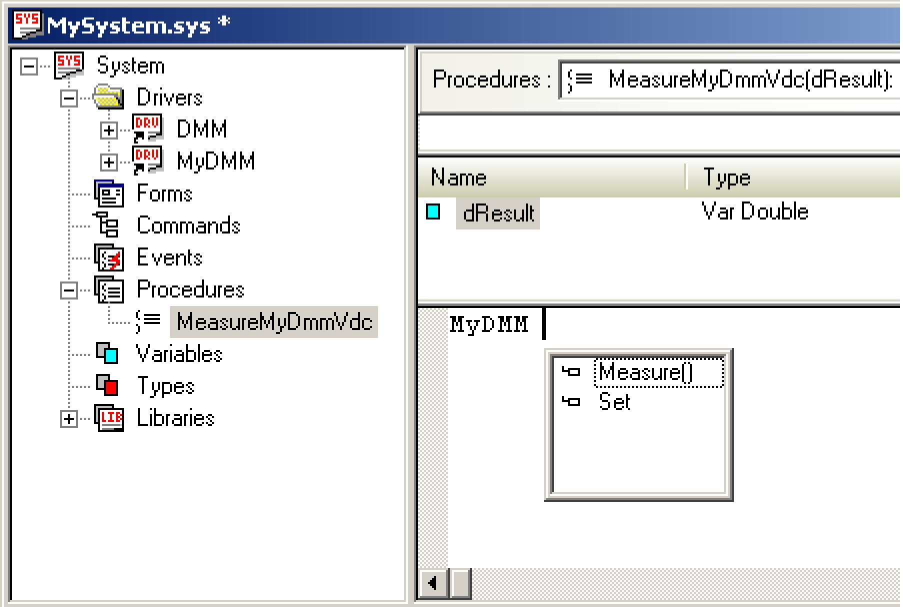
▼ To create a System procedure:

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 **MeasureMyDmmVdc**.
4. Right-click on the procedure variables view and select **Insert Parameter/Variable At**. A new variable is inserted. Rename it to **dResult**.

5. Right-click on **dResult**. Select **Properties** and change the variable type to **Double**.

6. Select the **Var** parameter type from the **Parameter** combo box.

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



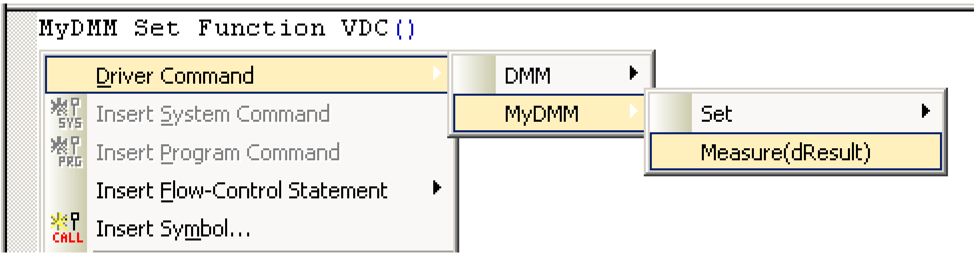
Procedures : MeasureMyDmmVdc(dResult):

Name	Type
dResult	Var Double

```
MyDMM 
```

8. Press the down arrow key to select **Set.** Press ENTER and continue to select **Function** and **VDC**.

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 **MyDMM**, and then select **Measure(dResult)** as shown here:



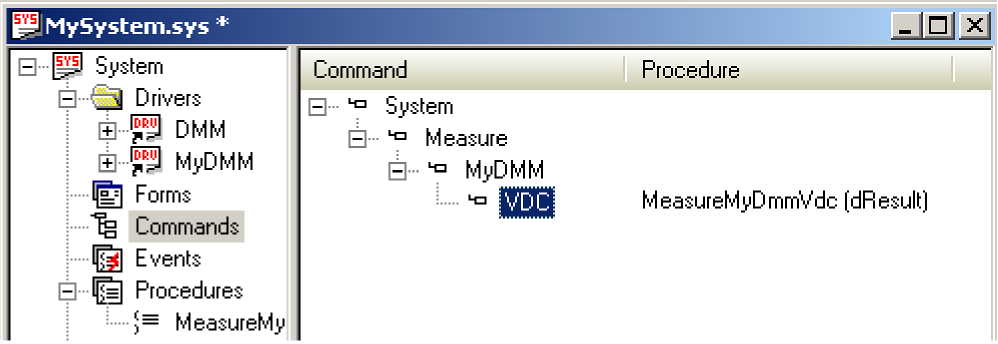
```
MyDMM Set Function VDC()
```

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:

1. Select **Commands** from the tree view in the system document view. The commands view displays in the right pane.
2. Right-click on **System** and select **Insert Command Below** from the context menu. A new command item is created.
3. Rename the command item to **Measure**.
4. Right-click on **Measure** and select **Insert Command Below** from the context menu.
5. Rename the new command item to **MyDMM**.
6. Insert another command item below **MyDMM** and rename it to **VDC**.
7. Select **MeasureMyDmmVdc** from the procedures list and click on the **Attach Procedure** button. The view should look as follows:



Command	Procedure
System	
Measure	
MyDMM	
VDC	MeasureMyDmmVdc (dResult)

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.

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 display a waveform in a chart control. 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	Shows how to add controls and to align them on the form.
Setting Control Properties	Explains control properties and how to set them from their property pages.
Setting Controls Tab Order	Explains what the Tab Order is and how to set it.
Testing the Form Layout	Explains how to use the Test Form command.
Using Events	Explain what are events and how ***ATEasy*** calls them.
Writing an Event for the Close Button	Implements the **OnClick** event for the **Close** button.
Adding Variables	Adds variables to a form.
Writing an Event for the Acquire Button	Implements the **OnClick** event for the **Acquire** button.
Writing Procedures	Writes the **AcquireData** procedure to fill the arrays for the chart.

Topic	**Description**
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.

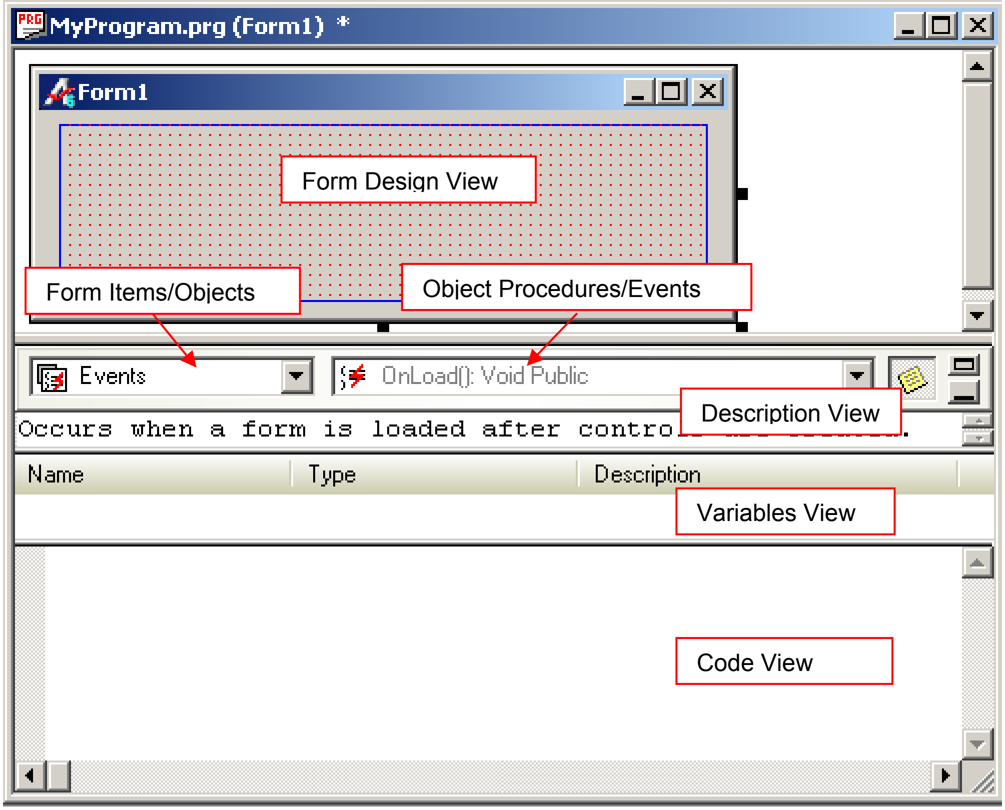
Creating a Form

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.

▼ To create a form:

1. Right-click on the **Forms** submodule under **MyProgram** and select **Insert Form Below** from the context menu. A new Form called Form1 is created.

Creating the new form causes ***ATEasy*** to display the Form View as shown here:



Occurs when a form is loaded after contro[…]

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)

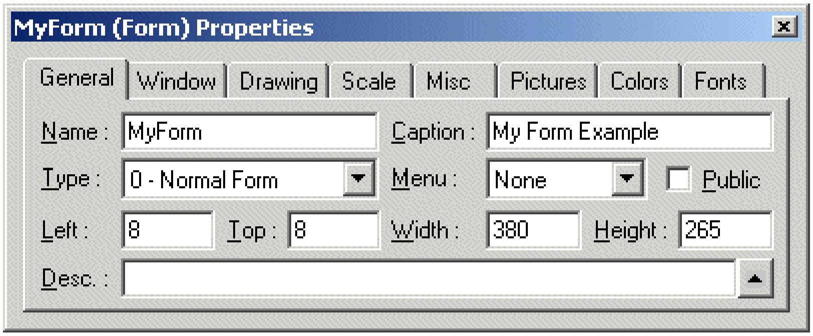
Additional Form properties pages include:

- **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:

1. Open the MyForm Properties window by clicking the right mouse button on **MyForm** and selecting **Properties**. The properties window opens.
2. Rename the form to **MyForm**.
3. 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:

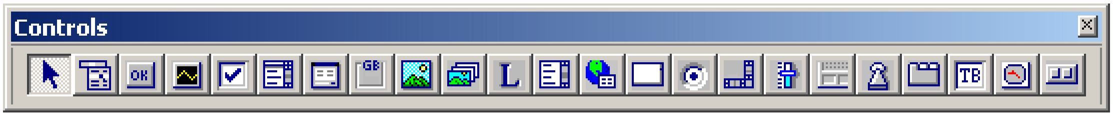


Name : MyForm
Caption : My Form Example
Type : 0 - Normal Form
Menu : None
□ Public
Left : 8
Top : 8
Width : 380
Height : 265
Desc. :

Form Controls

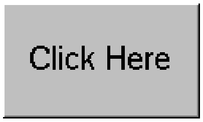
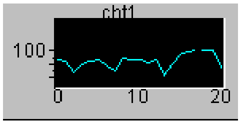
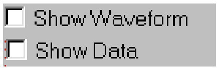
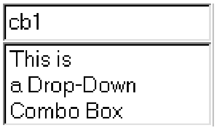
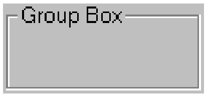
Forms need to be populated with controls and menus in order to be useful. Controls are the interface elements through which the end user makes choices or obtains information. Buttons, text boxes, checkboxes, list boxes, and charts are all examples of controls supplied with ***ATEasy***. Controls have 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.

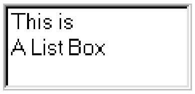
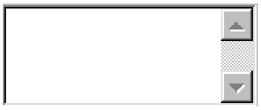
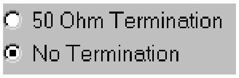
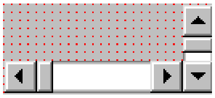
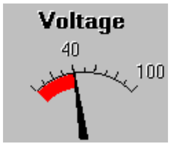
Controls and menus are added from the **Controls** toolbar. The Controls toolbar appears on the screen whenever the form view is active. It contains all the available controls that can be placed on a form as shown here:

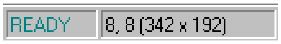
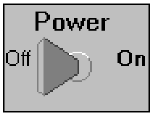
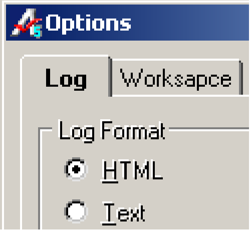
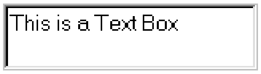
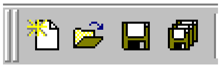


The first button in the toolbar (appears on the left side) is the **selection tool**. The second button (showing a menu) is used to add a **menu** to the form. The rest of the buttons are controls. To add any one of these controls, click on it, then click in the form client area, and drag the control to the appropriate size.

ATEasy provides the following controls:

Type	Description	Appearance	Tool
AButton	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.	 This is A List Box	
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.	 ○ 50 Ohm Termination ● No Termination	
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).		

Type	Description	Appearance	Tool
AStatusBar	Used for displaying a status bar on a form. The status bar contains panes that can display text, images, keyboard state and more.	 READY | 8, 8 (342 x 192)	
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.	 This is a Text Box	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

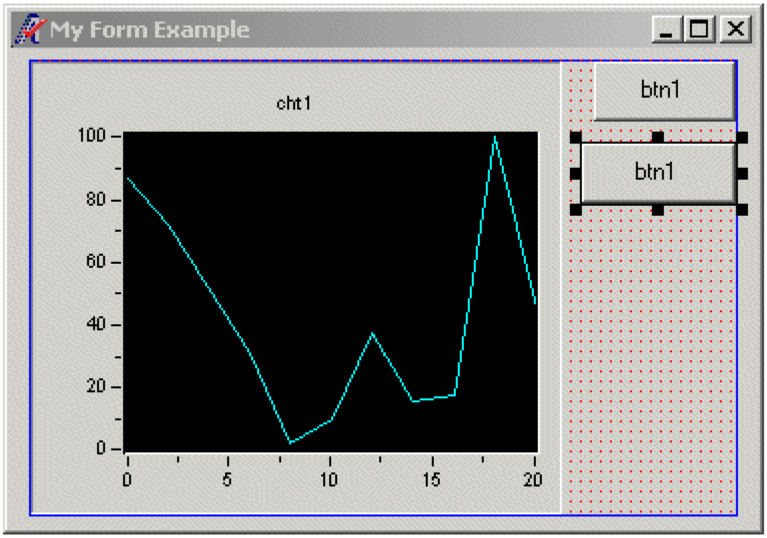
In your example, you will be adding a chart control on the left and two buttons on the upper right of your form.

▼ To add the controls to the form:

1. Click the **AChart** control on the Controls toolbar. Place your mouse pointer at the upper left side of the form. Click and drag a rectangle that is approximately two thirds of the size of your form.

2. Click the **AButton** control on the Controls toolbar. Place your mouse pointer at the upper right side of the form. Click and drag a rectangle. When you let the mouse up, the button is labeled **btn1**.

3. Copy the button. 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:

1. Click on **btn1** 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.

2. Once btn1 is in the correct position, you can make **btn2** align equal to the left edge of **btn1**. To do that you must select multiple controls. Click on **btn2** to select the control, press the **SHIFT** key, and then click on **btn1**. Notice that the two controls are now selected. The last control that you select is called the **pivot control**. Click on the **Align Left** command from the **Form Design toolbar** or from the **Arrange** menu under the **Edit** menu. Notice that btn2 is now aligned equal to the left edge of btn1.

3. While the two buttons are still selected, make btn2 have the same width and height of btn1. 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 btn1 and btn2 are aligned and are the same size.

4. Next, make the chart and top button align to the top edge. Using the same technique as step two, select the chart (**cht1**). Holding the SHIFT key down, select **btn1. C**lick the **Align Top** command. Both controls are aligned to the top.

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 "**btn**" for all Button Controls. This way, when you see a reference in the program to **MyForm.btnClose**, you know it refers to a Button Control probably called "**Close**."

▼ To set the control properties for the buttons:

1. Double-click on **btn1** to display its properties. The button property page displays.
2. On the General properties page, change the Name to **btnClose**.
3. Check the **Default** checkbox to make the button the default button. Selecting default causes the button to be pressed when the user presses the ENTER key (and perform the procedure assigned to that button). Only one button in a form can be designated the default. Notice that the page also has a **Cancel** check box. This designates a button to be a cancel button accepting the ESC key when the form is active as the user pressed the button. This is usually used for a Cancel button on a form.
4. Click on the **Control** page to activate the page. Change the Caption to **Close**. 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.
5. Select **btn2** and repeat step two, changing the Name to **btnAcquire.**
6. Repeat step four for **btn2**, changing the Caption to **&Acquire**. Adding the "**&**" in front of a character allows the user to press ALT and the character immediately following the "&" in order to select the button. The user can select buttons with one key stroke instead of either using the mouse or pressing Tab until the control (button) has focus and then pressing the space bar to press the button. Note the A now appears underlined (**A**) in the form.

Note: If you want to select a character *other* than the first character of a button (such as in the case that you have two buttons with the same first letter, for example, Acquire and Add), you may place the "&" immediately in front of the character to use with the Alt key. For example, to use the first "d" of Add, you would enter A&dd. ALT + D would trigger the button, and the text on the button face would appear as **Add**.

▼ To set the control properties for the chart:

1. Select the chart and activate the General page. Change the control's Name to **chtData**.

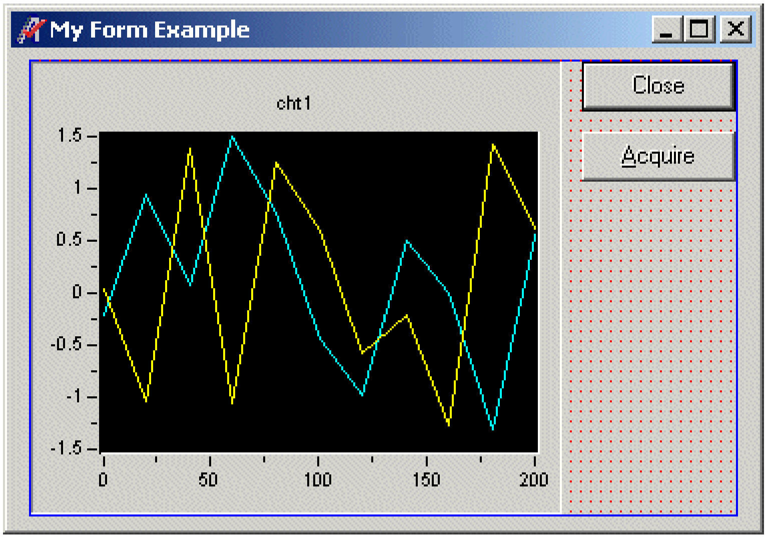
2. Activate the **Axes** page. The page displays a list of all the axes available via a list box. Click on the X-axis in the list and set Max to **200**. Click on the Y-axis and set the Min to **–1.5** and the Max to **1.5**. This causes the chart to display 200 samples (maximum) and the values of the data will be between –1.5 to 1.5.

3. Activate the **Ticks** page and change the X-Axis Spacing to **1 – By Division**. Change the **MajorDivision** to **4** and the **MinorDivision** to **2**. Do the same for the Y-Axis, changing the Spacing to **1 – By Division**, the **MajorDivision** to **6**, and the **MinorDivision** to **2**.

4. Activate the **Plots** page and add a second plot by clicking on the **New** button . Your chart will contain two plots, each displaying its own data in a line on a chart. Name the second plot, **Plot2**. Move it below Plot1 on the list by clicking the move down button ↓. Change the **LineStyle** and **PointStyle** color to yellow, by using the color drop down button to the right of each . Each plot now displays in a different color **Plot1** in cyan and **Plot2** in yellow.

5. Change the background of the chart to green by activating the **Colors** page. Select **BackColor** from the Property Name list box and select green.

Your form should appear as below:



Setting Controls Tab Order

Your next step is to set the **Tab Order** of the controls in the form. The tab order sets the order of controls in the form. This order is used when the user presses the TAB key to move the focus from one control to the next. Typically, form controls tab order is organized from top to bottom, then, from left to right. Your form controls may already be in the correct order; however it is a good habit to verify that the tab order is correct.

▼ To set and verify the form controls tab order:

1. Right-click on the Form and select Arrange, then Tab Order , from the context menu. The controls are displayed with blue labels indicating their order.
2. Click on the chart label (**0**) set it to be the first in the tab order, Next click on the Close button (**1**) and then click on the Acquire button (**2**).

Testing the Form Layout

Your form is now ready for a test of its layout. ***ATEasy*** has a special tool allowing you to execute the form in order to test the form's initial display without writing code to load or display the form.

▼ To test the form layout and initial display:

1. Click on **Test Form!** from the Form Design toolbar. This tool is also available below **Arrange** in the **Edit** menu. The form is now displayed.
2. Press **ALT + A** to check if the Acquire button is pressed. Press ENTER to check if the default button **Close** is pressed. Press the **TAB** key a few times to check the tab order.
3. Press **ESC** to exit the Test Form! mode.

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 Close Button

You will now add a simple event to MyForm. This event controls what happens when the Close button is pressed.

▼ To write the btnClose.OnClick Event:

1. From the form view, select **btnClose** from the form items combo box. The right combo box now displays the available events for the **AButton** control.

2. Select the **OnClick** event. Click in the procedure space and begin typing:

   ```
   Unload Form
   ```

   The Unload statement will close the form window on the user’s screen. The Form object will still exist after that line, and you can still use the form variables and procedures after this line, however the visible form is erased from the screen. When the user closes the form from the title bar by clicking on the X button, ***ATEasy*** will unload (close) the form automatically.

Now, if you collapse the right combo box you will see that all events are shown with gray text while **OnClick** now appears normal text, showing that the event is used.

Before you can add the code for the Acquire button and chart, you need to add some variables.

Adding Variables

Add a form variable to a form just as you do with any other module. You will add three variables, two arrays, and one integer that stores the number of times the user clicked the Acquire button.

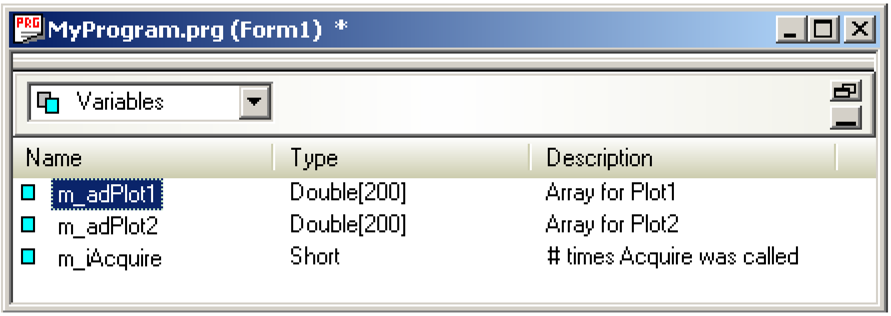
▼ To add form variables:

1. From the form view, select **Variables** from the form items combo box. The view below now shows a variables view.
2. Right-click on the variables view and insert the following variables and properties:

Name	**Type**	**Dim**	**Dim Size**	**Description**
m_adPlot1	Double	1	200	Array for Plot1
m_adPlot2	Double	1	200	Array for Plot2
m_iAcquire	Short	0	n/a	# times Acquire was called.

Note the **m_** is used to tell the user that the variable is a member variable of the form.

Your variables view should look similar to the following:



Name	Type	Description
m_adPlot1	Double[200]	Array for Plot1
m_adPlot2	Double[200]	Array for Plot2
m_iAcquire	Short	# times Acquire was called

Writing an Event for the Acquire Button

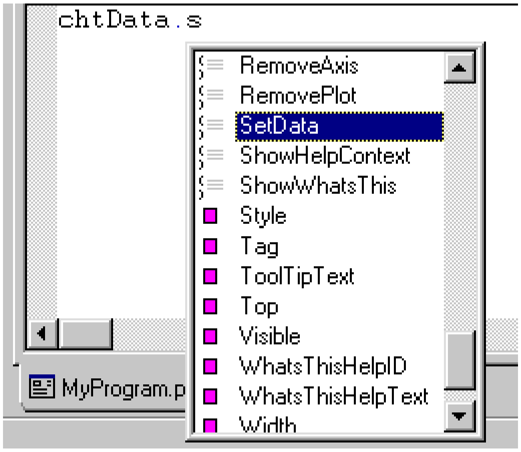
Now you can write the code for the **OnClick** event of the Acquire button. This event fills the form arrays by calling a form procedure, then it refreshes the form chart to display the array content. The event also increments the m_**iAcquire** form variable and displays that number in the chart caption.

▼ To add an Event with Variables:

1. From the form view, select **btnAcquire** from the form items combo box. The right combo box now displays the available events for the **AButton** control.
2. Select the **OnClick** event. Click in the procedure space and begin typing:

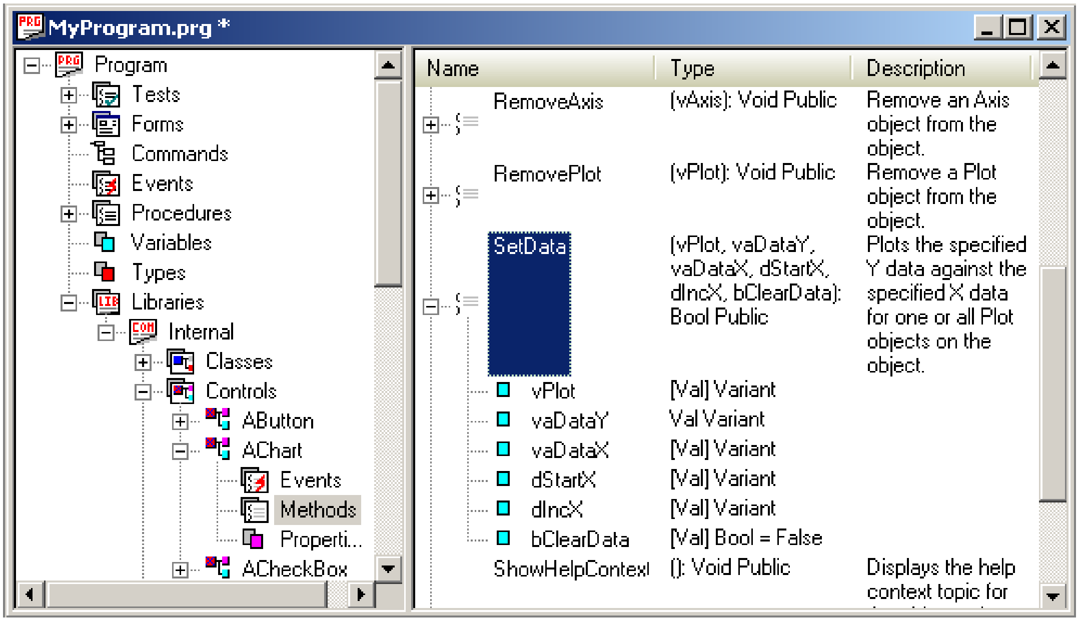
```
! increment the # of times acquire was called
m_iAcquire=m_iAcquire+1
! set chart caption
chtData.Caption="# of Sin Cycles:"+str(m_iAcquire)
! acquire and show data
AcquireData(m_adPlot1, m_adPlot2)
chtData.SetData("Plot1", m_adPlot1,,,, True)
chtData.SetData("Plot2", m_adPlot2,,,, True)
```

Note that while you are typing the member operator '**.**', a code completing member list pops up to display the available control members as shown here:



```
chtData.s
```

The user can view this list by browsing in the internal library **AChart** control entry. The Caption is a property of the chart control while **SetData** is a method performing an action on the control as shown here:



Name	Type	Description
RemoveAxis	(vAxis): Void Public	Remove an Axis object from the object.
RemovePlot	(vPlot): Void Public	Remove a Plot object from the object.
SetData	(vPlot, vaDataY, vaDataX, dStartX, dIncX, bClearData): Bool Public	Plots the specified Y data against the specified X data for one or all Plot objects on the object.
vPlot	[Val] Variant	
vaDataY	Val Variant	
vaDataX	[Val] Variant	
dStartX	[Val] Variant	
dIncX	[Val] Variant	
bClearData	[Val] Bool = False	
ShowHelpContext	(): Void Public	Displays the help context topic for

The next step will be to write the **AcquireData** form procedure, which will be used to fill the plots' arrays with sample data.

Writing the AcquireData Procedure

Just as you have written procedures before in other modules, you will write a procedure within your form. This procedure initializes the arrays with a dummy waveform. The first array will display a sin wave with increasing frequency. The second array will be filled with a sin wave that cycles **m_iAcquire** times.

▼ To write the form procedure:

1. From the form view, select **Procedures** from the form items combo box. The right combo box now displays a list of the form procedures, which is initially empty.

2. Select **Procedure Below** from the **Insert** menu. A new procedure is created.

3. From the Properties window, change the name to **AcquireData** and the description to **Initialize arrays with a dummy waveform**.

4. Create the following procedure variables:

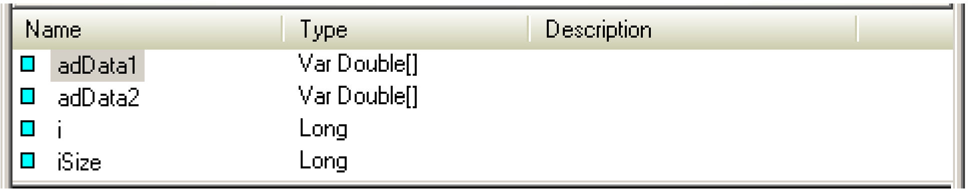
Name:	**adData1**
Parameter:	**Var**
Type**:**	**Double**
Dim**:**	**1**

Name:	**adData2**
Parameter:	**Var**
Type:	**Double**
Dim:	**1**

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

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

Your screen should now look similar to the following:



Name	Type	Description
□ adData1	Var Double[]	
□ adData2	Var Double[]	
□ i	Long	
□ iSize	Long	

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

```
iSize=sizeof(adData1)/sizeof(double)
for i=0 to iSize-1 do
      ! freq increase over samples
      adData1[i]=sin((PI*i)/(iSize-i+1))
      ! m_iAcquire # of sin cycles
      adData2[i]=sin ((PI*i*m_iAcquire*2)/iSize)
next
```

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.

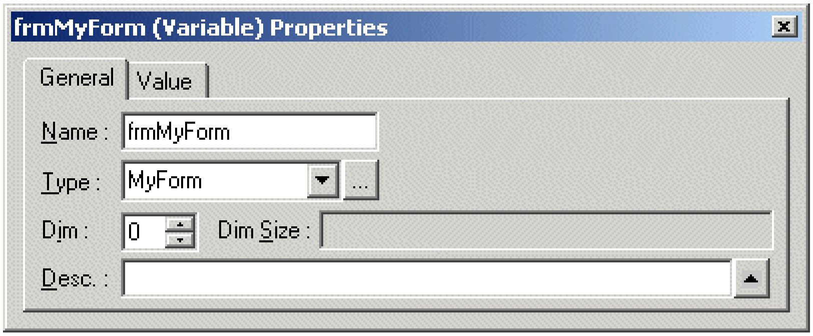
- The third and optional parameter is the form's parent **window handle**. A window handle is a 32-bit number used to uniquely identify the window in the current process. Each form has a handle that can be retrieved using the **hWnd** property. Passing 0 as a handle (or omitting the argument) uses the 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

General | Value

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).

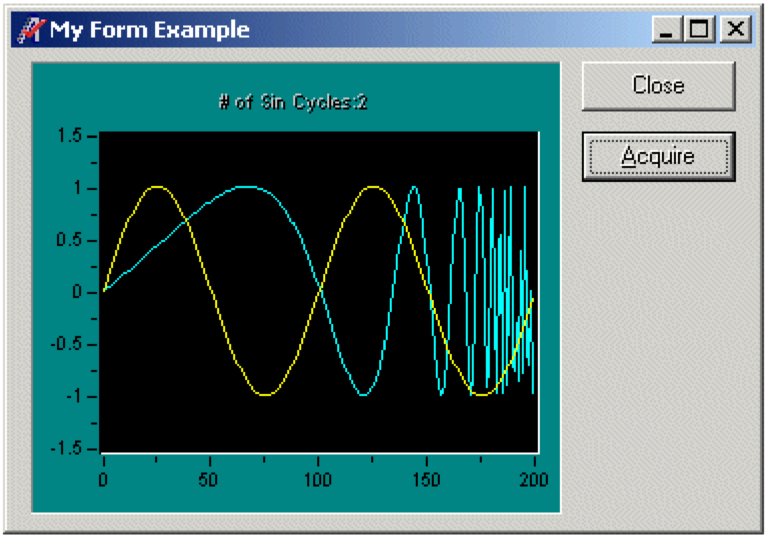
Testing the Form

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.

2. Click twice on the **Acquire** button. The form should now display two sine cycles as shown here:



3. Click on the **Close** button.

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.

About External Libraries

This chapter discusses extending ***ATEasy*** functionality by adding and interfacing with external libraries. It provides an example of how to write an ISA PC board driver that uses a Dynamic Link Library to program the board. It provides more details of how to initialize a driver. It shows how to handle errors in drivers and in the application. This chapter explains how to use Type Libraries and COM objects and classes, as well as providing an example for using a Microsoft Excel COM object.

Topic	**Description**
Overview of Libraries	Explains two types of libraries that can be used in an ***ATEasy*** module: DLLs and Type Libraries.
Creating a DLL Based Driver	Explains how to create a driver and to add a DLL. Also shows how to use and configure a DLL-based driver.
About the GXSW.DLL	Provides an example for a DLL procedure and provides an example of a C header file.
Declaring DLL Procedures	Explains how to define DLL procedures and parameters.
Importing C/C++ header file	Explain how to import C/C++ header file (.h) to declare DLL procedures, types and constants
Using DLL Procedures	Shows how to call DLL procedures.
Driver Initialization	Shows how to use the driver's **OnInit** module event to initialize the driver and how to hide the board handle parameter used to identify the board to the driver.
Handling Errors in a Driver	Explains how to handle errors in the driver.
More about Error Handling	Provides more details regarding ***ATEasy*** error types and about error handling in an application using the **OnError** event.
COM Objects and Type Libraries	Provides details regarding using COM objects and external Type Libraries.
Using the Excel Type Library	Provides an example using the Excel Type Library.
Using the Object Data Type	Provides an example using the Object Data Type.
.NET Assemblies	Using .NET assemblies

Topic	Description
Using VI	Using LabView VI and LLB files

Overview of Libraries

Libraries are external modules containing procedures, classes and other programming elements. ***ATEasy*** can use two kinds of libraries:

- **Dynamic Link Libraries** (DLL) – which are files, typically with .DLL file extensions, containing procedures. Most PC-based instrument drivers are now shipped as a DLL library. Microsoft Windows is also built from a set of DLLs that provide access to its services. DLLs do not contains type definition of the procedures and their parameters they hold, ***ATEasy*** can make use of C/C++ header files (.h) that is usually supplied with the DLL and create the procedures, types and constants definitions by reading them from the header file and creating ***ATEasy*** equivalents.

  ATEasy version 7.0 supports creating a DLL by setting the project target type to DLL. With the Build Command, an ***ATEasy*** DLL will be created, and it can in turn be used by other ATEasy or other programming languages projects. For more information, refer to ***ATEasy*** Users' Guide.

- **Type Libraries** – which contain classes, procedures, and other programming elements and are based on Microsoft component technology (COM). Type libraries allow you to make use of classes exposed by external libraries or applications. Examples of type libraries are: Active X controls or MS-Excel. Unlike DLL where you are required to define the programming elements included in it, a type library contains a complete definition of the programming elements exported by the library.

- **.NET Assemblies** – which contain classes, procedures, and other programming elements and are based on Microsoft .NET technology. Unlike DLL where you are required to define the programming elements included in it, a .NET assembly contains a complete definition of the programming elements exported by the library.

In this chapter, you will learn how to use an instrument driver DLL for the GX6115, a 3U PXI high current relay board. The driver uses a DLL GXSW.DLL provided with ***ATEasy.*** More robust examples for using DLLs can be found in the examples provided with ***ATEasy*** in the Language.prj and DLL.prj (along with the DLL sources written in C) files located in your \Examples subfolder.

You will also learn how to use a type library. We will use the MS-EXCEL type library to use Excel. A more complete example for this can be found in Excel.prg located in your \Example subfolder. Also provided in your examples folder is a .NET assembly including sources and ATEasy application DotNet.prj and C# programming language .NET sources for the assembly used.

Libraries are added to any module under the **Libraries** submodule. Each module can have its own libraries, which, if made public, can be used by other modules. Once a library is added to a module, you can call procedures and use classes residing in the library. In addition, if a type library contains an Active X control, then the control will be added to the Controls toolbar. It can be used by dragging it onto an ***ATEasy*** form in a similar way to the way you use ***ATEasy*** built-in controls.

Creating a DLL-Based Driver

In this example, create a driver, **MyGx6138.drv**, which uses a DLL **GXSW.DLL**. The driver is similar to the **GX6138.drv** driver for the 38 Relay (switching) board from Geotest that is part of the GXSW package that can be download from Geotest web site. We will create ATEasy DLL procedure to describe the procedures residing in the DLL. These procedures will be used to program the GX6138 board. For reference, you will also add the **GX6138.drv** driver that is provided in the ***ATEasy*** Drivers folder.

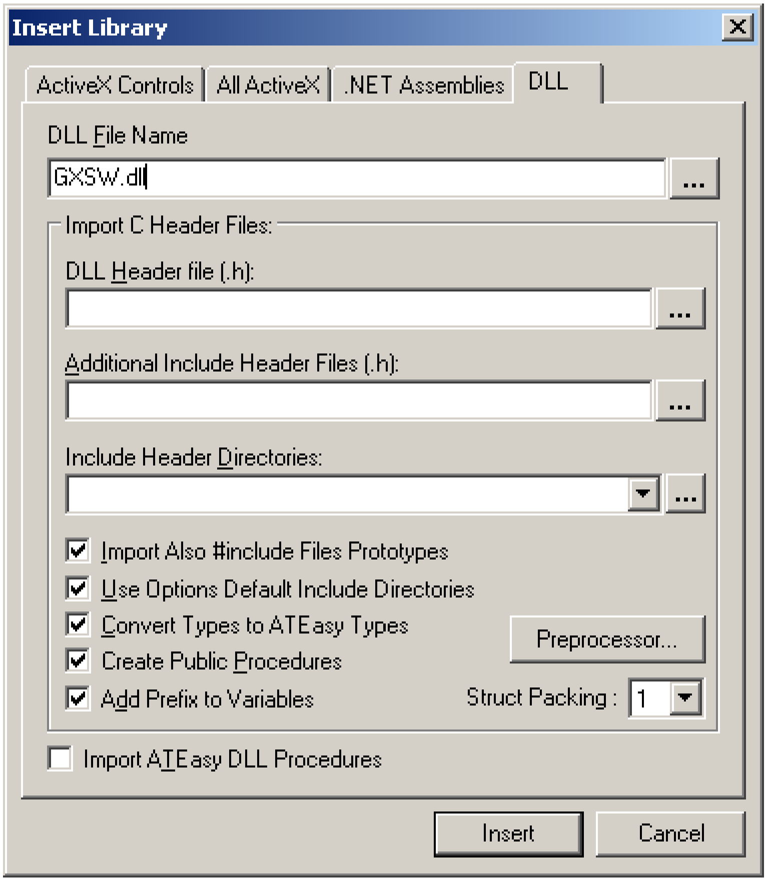
▼ To create a driver using GXSW.DLL:

1. Right-click on the **Drivers** submodule in MySystem. Select the **New Driver** command from the context menu. A new driver is created. Rename the driver name to **MyRELAY** from the workspace window.

2. Right-click on **Libraries** submodule and select **Insert Library Below** from the context menu. The **Insert Library** dialog is displayed. This dialog is used to insert a Type Library, a .NET assembly or a DLL.

3. Activate the **DLL** tab and click on the **Browse** button. The Select File dialog appears.

4. Select **GXSW.DLL** from the Windows System folder (System32 folder on Windows NT/2000/XP/VISTA or System Folder under Windows 9x/Me) and click **Open.** Alternatively, you can type GXSW.dll; ***ATEasy*** will find it since it's in the Windows default search path for DLLs. The DLL file name is now displayed in the edit box as shown here:



DLL File Name: GXSW.dll

DLL Header file (.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 Procedures

☑ Add Prefix to Variables

Struct Packing : 1

☐ Import ATEasy DLL Procedures

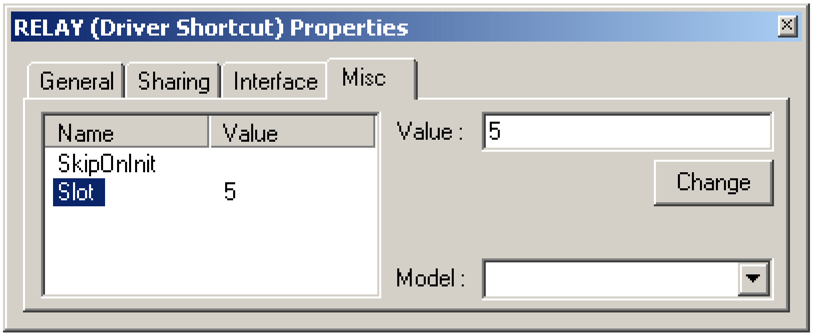
5. Click **Insert**. A GXSW library is added under the new driver **Libraries** submodule with the default name of **GXSW**.

   NOTE: the Import C Header Files fields will be used later in this chapter to import the DLL procedures, for now, we will declare these procedures manually.

6. Right-click on the library and select **Properties** from the context menu. The library properties page is displayed. Check the **Public** checkbox. By default, DLL procedures are attached to commands and need not be public. Since you will be using the DLL procedure within the program, you should make it public so other modules can use the procedure.

7. Right-click on the **Drivers** submodule and select **Insert Driver Below** from the context menu. The Insert Driver dialog displays. Select **GX6138.drv** from the ***ATEasy*** Drivers folder and click **Open**. The GX6138 driver will be inserted to the system type with a name of **RELAY** for the driver shortcut (taken from the driver default name property).

8. Before you use the RELAY driver, you must configure its interface. Right-click on the Driver Shortcut symbol in the Workspace window and select **Properties** . The Driver properties appear. Activate the **Misc** page. Select the **Slot**, type in the GX6138 PXI slot number as appears in the **PXI-PCI Explorer** application. The PXI-PCI explorer can be start by selecting it from the **Geotest, HW** from the Windows Start menu. The **Misc** page should display as shown here:



RELAY (Driver Shortcut) Properties

Name	Value
SkipOnInit	
Slot	5

Value : 5

Model :

The **Slot** parameter is used to supply the driver the PXI slot number used when initializing the board. The **SkipOnInit** parameter will be use to signal the driver to not initialize the driver when the driver **OnInit** event is called, leaving it empty will cause the driver to be initialize when you start running the application. The parameters can be retrieved by the driver using the **Driver.Parameters("Slot")** expression. This driver uses DLL to communicate with the instrument and does not use I/O Table to communicate with the instrument and therefore they have the **None** driver interface (default for new drivers).

9. Click **Save All** to save your work. When prompted save the new driver to **MyGx6138.drv** in the **MyProject** folder.

Once the library is added to the driver, you need to define the procedures residing in the DLL since a DLL does not contain type information about its procedures. To define these procedures you need to look into the DLL documentation or use the C programming language header file that is sometimes provided with the DLL. Your next step will be to declare the procedures under the library you just inserted. Before you start, let’s look at the GXSW.DLL header file.

About the GXSW.DLL

You can download and install the GXSW package from Geotest web site. The package contains driver for all Geotest PXI switching boards and is supplied with the C programming language header file to describe its procedures. The header file GXSW.H can be found in the GXSW driver folder (C:\Program Files\ GXSW). Opening the file with a text editor and browsing will show you the prototypes of many of the boards supported by the GXSW package. The GX6138 most relevant sections in this file are shown here:

```
//*********************************************************************
// GXSW General purpose functions
//*********************************************************************
VOID WINAPI GxSWGetErrorString(SHORT nError, PSTR pszMsg, SHORT
      nErrorMaxLen, PSHORT pnStatus);
VOID WINAPI GxSWGetDriverSummary(PSTR pszSummary, SHORT nSummaryMaxLen,
      PDWORD pdwVersion, PSHORT pnStatus);

// GTXXXPanel() constants
// nMode for panel functions
#define GXSW_PANEL_MODELESS                        0
#define GXSW_PANEL_MODAL                           1
// Relay State
#define GXSW_STATE_OPEN                            0
#define GXSW_STATE_CLOSE                           1

//**************************************************************
// Gx6138 functions
//**************************************************************
VOID WINAPI Gx6138Initialize(SHORT nSlot, PSHORT pnHandle, PSHORT
      pnStatus);
VOID WINAPI Gx6138Reset(SHORT nHandle, PSHORT pnStatus);
VOID WINAPI Gx6138Panel(PSHORT pnHandle, HWND hwndParent, SHORT nMode,
      HWND * phwndPanel, PSHORT pnStatus);
VOID WINAPI Gx6138GetBoardSummary(SHORT nHandle, LPSTR pszBoardSum,
      SHORT nSumMaxLen, PSHORT pnStatus);

VOID WINAPI Gx6138Close(SHORT nHandle, SHORT nChannel, PSHORT
      pnStatus);
VOID WINAPI Gx6138Open(SHORT nHandle, SHORT nChannel, PSHORT pnStatus);
VOID WINAPI Gx6138GetChannel(SHORT nHandle, SHORT nChannel, PSHORT
      pnState, PSHORT pnStatus);
VOID WINAPI Gx6138SetChannels(SHORT nHandle, LONG lHighChannelStates,
      LONG lLowChannelStates, PSHORT pnStatus);
VOID WINAPI Gx6138GetChannels(SHORT nHandle, PLONG plHighChannelStates,
      PLONG plLowChannelStates, PSHORT pnStatus);

// First and Last channels numbers
#define GX6138_CHANNEL_FIRST                       1
#define GX6138_CHANNEL_LAST                        38
```

Declaring DLL Procedures

Declaring procedures in a DLL is divided into two steps. The first step is to define the procedure name and return value. The second step is to define the procedure parameters. In most cases, the parameters are described in C programming language, that require you to translate the C types mentioned in the header file to ***ATEasy*** data type.

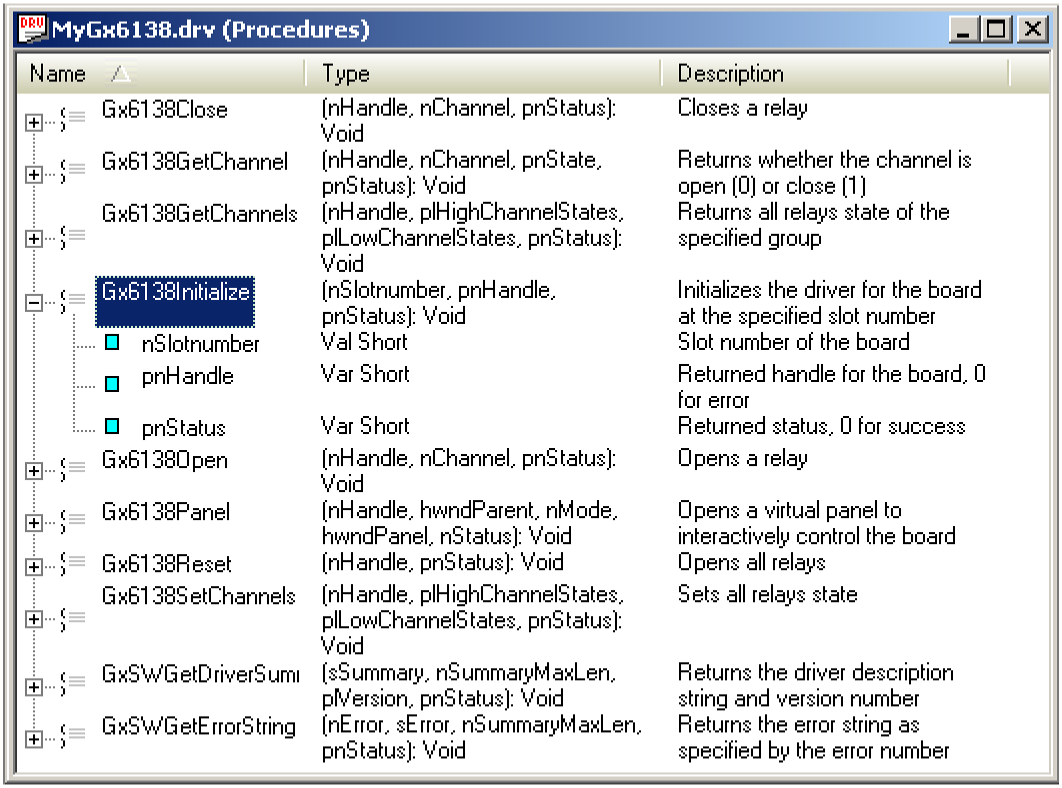
In general, C and ***ATEasy*** types are very similar in the type name and in internal representation of the type. Here are some guidelines:

- C pointers can be converted to VAR parameters. For example the C data type **short *** can be converted to ***ATEasy*** data type **VAR Short**. If no pointer is used, use the ***ATEasy*** **VAL** parameter.
- Arrays and Strings are pointers in both C and in ***ATEasy***. You can use both **VAL** and **VAR**. For example **char *** (or **LPSTR**) can be **VAL String** or **VAR String**. You should use **VAR** if you plan to change the string and reflect the change to the caller. When declaring **VAL, *ATEasy*** creates a copy of the variable and passes it to the DLL procedure. Calling with **VAR** is faster.

▼ To declare the DLL procedures:

1. Right-click on the Procedure below the GXSW Library in the tree of the document view and select **Insert Procedure Below** . A new procedure is inserted.
2. Right-click on the new procedure. Select **Properties** from the context menu. The DLL procedure property page displays. Type the procedure name: **Gx6138Initialize** and check the **Public** checkbox. Set the description to: **Initializes the driver for the board at the specified base address**.
3. Right-click on **Gx6138Initialize** and select **Insert Parameter Below** . A new parameter is inserted.
4. Rename the parameter to **nSlot**. The new parameter has a type of **VAL Short** – this matches the C Type **SHORT**.
5. Repeat steps three and four for the **pnHandle** and **pnStatus** parameter. Their parameter type must be change to **VAR** since they are declared as **LPSHORT** (**short ***). At this point, the procedure is declared and can be used.
6. Repeat steps one through five for each of the procedures that are defined in the previous page. You can use **Copy** and **Paste**, or **Drag** and **Drop** to expedite your work. Make sure the string parameters are defined as **VAR** since they return a value. You can use the Gx6138.drv driver as a reference or for copying the procedures defined there instead of typing them.

At this point, the DLL procedures are defined and ready to use. Your DLL procedures view should look similar to this:



MyGx6138.drv (Procedures)

Name	Type	Description
Gx6138Close	(nHandle, nChannel, pnStatus): Void	Closes a relay
Gx6138GetChannel	(nHandle, nChannel, pnState, pnStatus): Void	Returns whether the channel is open (0) or close (1)
Gx6138GetChannels	(nHandle, plHighChannelStates, plLowChannelStates, pnStatus): Void	Returns all relays state of the specified group
Gx6138Initialize	(nSlotnumber, pnHandle, pnStatus): Void	Initializes the driver for the board at the specified slot number
nSlotnumber	Val Short	Slot number of the board
pnHandle	Var Short	Returned handle for the board, 0 for error
pnStatus	Var Short	Returned status, 0 for success
Gx6138Open	(nHandle, nChannel, pnStatus): Void	Opens a relay
Gx6138Panel	(nHandle, hwndParent, nMode, hwndPanel, nStatus): Void	Opens a virtual panel to interactively control the board
Gx6138Reset	(nHandle, pnStatus): Void	Opens all relays
Gx6138SetChannels	(nHandle, plHighChannelStates, plLowChannelStates, pnStatus): Void	Sets all relays state
GxSWGetDriverSum	(sSummary, nSummaryMaxLen, pVersion, pnStatus): Void	Returns the driver description string and version number
GxSWGetErrorString	(nError, sError, nSummaryMaxLen, pnStatus): Void	Returns the error string as specified by the error number

As you can see, the DLL procedures and their parameters are documented. This is used when the user uses the procedure.

Importing C Header File

Declaring procedures, types and constants manually can take considerable time. ***ATEasy*** can make use of C header file supplied with DLL to import procedures, types (struct, enum or typedef) and constants. C data types that are specified in a header file can have various interpretations. For example the C data type ‘unsigned int *’ when specified as a procedure parameter can be interpreted as a parameter that returns a DWord (Var DWord) or array of DWord (Var DWord []) or accept array of DWord (val DWord []). During the Import process ***ATEasy*** will ask the user to resolve Ambiguous data types by presenting the possible options. Selecting the right option is essential.

In the following example we will be using the Gx6138.h as an example for importing a C header file. The file was extracted from GXSW.h that is part of the GXSW package that can be installed from Geotest web site. The Gx6138.h file is installed by ***ATEasy*** in the Examples folder.

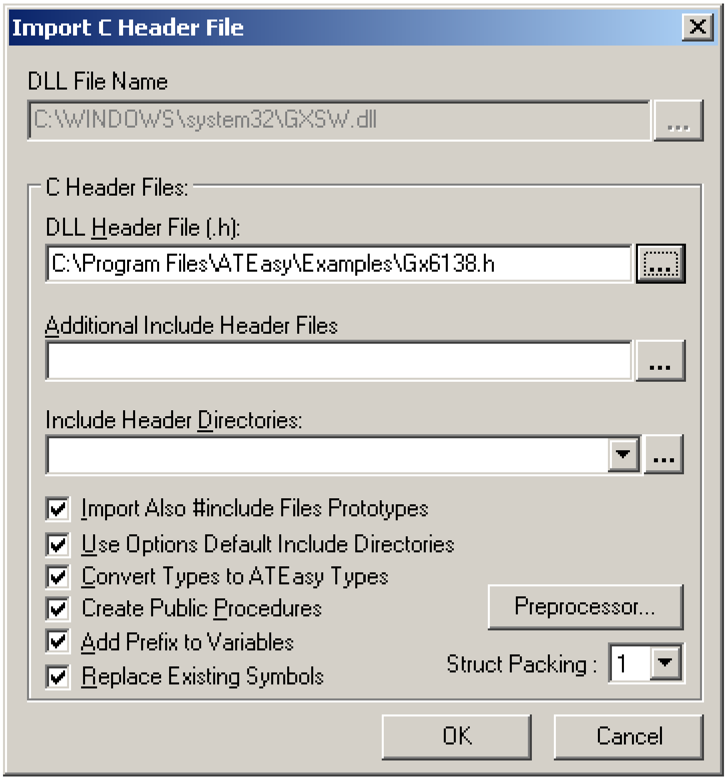
▼ To import C header file:

1. Create a new driver by right click on the system from the workspace window and select New Driver

2. Under libraries insert the GXSW.DLL as explained earlier in this chapter.

3. Right click on GXSW DLL under procedures and select **Import C Header File (.h)…**

4. Select the **C:\Program Files\ATEasy\Examples\Gx6138.h** and check the following options as shown here:



Import C Header File

DLL File Name: C:\WINDOWS\system32\GXSW.dll

C Header Files:

DLL Header File (.h): C:\Program Files\ATEasy\Examples\Gx6138.h

Additional Include Header Files:

Include Header Directories:

☑ Import Also #include Files Prototypes

☑ Use Options Default Include Directories

☑ Convert Types to ATEasy Types

☑ Create Public Procedures

☑ Add Prefix to Variables

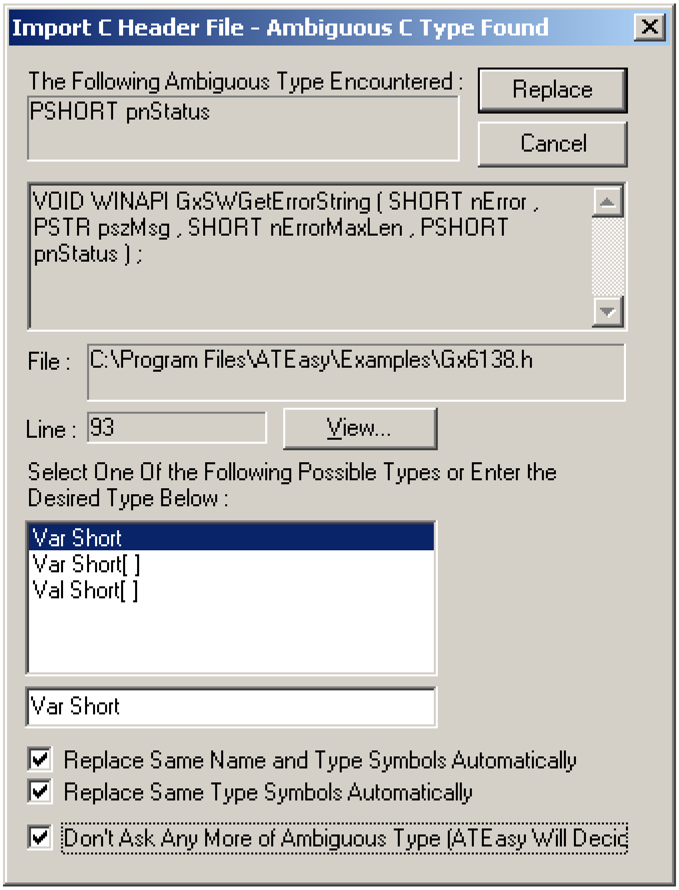
☑ Replace Existing Symbols

Struct Packing : 1

The **Additional Header Files (.h):** is used when you need to specify additional header files needed to import (separated by semicolons). The **Include Header Directories** is used to specify include directories that are needed when trying to resolve data types defined in other header files. The preprocessor button displays a header file that contains common definition and always parsed before the actual header file. This file can be edited to add or change types. The other options are documented in the On-Line help.

5. Click **OK** to import the header file.

6. ***ATEasy*** will prompt with the following Ambiguous C Type dialog:



Import C Header File - Ambiguous C Type Found

The Following Ambiguous Type Encountered : PSHORT pnStatus

VOID WINAPI GxSWGetErrorString (SHORT nError , PSTR pszMsg , SHORT nErrorMaxLen , PSHORT pnStatus) ;

File : C:\Program Files\ATEasy\Examples\Gx6138.h

Line : 93

Select One Of the Following Possible Types or Enter the Desired Type Below :

Var Short
Var Short[ ]
Val Short[ ]

Var Short

☑ Replace Same Name and Type Symbols Automatically

☑ Replace Same Type Symbols Automatically

☑ Don't Ask Any More of Ambiguous Type (ATEasy Will Decid

This dialog will be displayed for each C data type that ATEasy cannot find exact match. In this case the **PSHORT** data type is offered with 3 options **Var Short** and two array types. Since the status is returned from the procedure as a single number and not an array select the **Var Short**. To accelerate the migration check **Replace Same name/same type** and **Don't ask** checkboxes as shown here and click **Replace**.

The newly created GXSW library will now be created and will show similar procedures to the one declared manually earlier in this chapter. You can remove the new driver created from the system.

Using DLL Procedures

At this point, the DLL procedures are defined. Since we made the procedures public, you can call the procedures directly from any of your application modules, procedures, or tests. You need to define the variables **nStatus** and **nHandle** as short. Then, you need to call the **Gx6138Initialize** procedure as shown here with the PXI slot number (5) as displayed in the PXI-PCI Explorer:

```
Gx6138Initialize(5, nHandle, nStatus)
```

This will be used to initialize the board to retrieve the board handle, later it will be used with all other commands such as **Gx6138Close,** which closes relay #2 as shown here:

```
Gx6138Close(nHandle, 2, nStatus)
```

Alternatively, you can use the GXSW library to call the procedure:

```
GXSW.Gx6138Close(nHandle, 2, nStatus)
```

Or, by using the driver name:

```
MyRELAY.GXSW.Gx6138Close(nHandle, 2, nStatus)
```

Or even:

```
MyRELAY.Gx6138Close(nHandle, 2, nStatus)
```

Note that the first two examples call the RELAY driver instead of the MyRELAY driver if the RELAY driver is defined as the first driver in the system. To avoid this, the last two methods are preferred.

As you can see, calling a DLL procedure is very similar to calling an ***ATEasy*** procedure. Typically, after defining the DLL procedure, you will create Commands to provide the user a better interface than procedures. In addition, use commands to provide a simpler interface to hide the **nHandle** and **pnStatus** parameters each of the DLL procedures have. So, instead of calling Gx6138Close with tree parameters as shown previously, enter the command as:

```
RELAY Close (2)
```

This code is a lot simpler; it hides the implementation details from the user. The user does not need to know about **nHandle** or about error handling and **nStatus** as explained in the next section.

Driver Initialization

As you can see from these calls, you need to supply the board handle **nHandle** every time you call the driver procedures. Instead, you can define a driver variable **m_nHandle** that will have the driver handle. Every time you need to access the driver, you can use that variable. The variable can be initialized in the driver **OnInit** event that is called only once when the application starts. This relieves the user from calling any initialization code before calling the driver commands.

Looking over the **Gx6138** driver **OnInit** event will show you just that: **OnInit** has the following code:

```
if nSlot=0 then nSlot=Driver.Parameters("Slot")
if nSlot=0 then
   error -1, "Driver Parameter 'Slot' not set to the board PCI
      slot number.\nPlease set the parameter value from the
      driver shortcut property page."
endif
```

The Gx6138 PXI slot number if not supplied as a parameter to the Initialize is retrieved from the driver shortcut using the Parameters ADriver property. When you insert a driver into your system, you should set the Slot number parameter of the board.

Looking over at the driver procedure to see the code for the **Initialize** procedure, you will see the following code:

```
Gx6138Initialize(nSlot, m_nHandle, nStatus)
CheckError(nStatus)
```

As you can see, the **Gx6138Initialize** accepts the **nSlot** and returns the handle to **m_nHandle** and the status to **nStatus**. **nStatus** is passed to a driver procedure called **CheckError,** which provides error handling for the driver as explained in the next section.

Handling Errors in a Driver

As you can see from the GXSW DLL procedures, all functions have a parameter **nStatus** that is used to return a status indicating if an error occurs (<0). Ideally, you should check the return value after each call as shown here:

```
Gx6138Close(m_nHandle, 2, nStatus)
If nStatus<>0 then
   ! error do something.. like abort or MsgBox()

endif
```

Placing the return-value-checking code after each call in the test program makes the program very long and hard to read. Instead, you can call a single error-checking procedure, the **CheckError** driver procedure, to perform that test. Call **CheckError** with the **nStatus** parameter returned from the DLL procedure as an argument.

CheckError shows the following code:

```
if nStatus<0 then
   m_lLastError=nStatus
   GxSWGetErrorString(nStatus, sError, 256, nErrorStatus)
   error nStatus, sError   ! generate exception
endif
```

The function checks to see if **nStatus** contain an error code. If it does, it retrieves the description of the error and calls the ***ATEasy*** statement **error**. The error statement generates an ***ATEasy*** run-time error displaying the error message returned from the DLL.

Looking through the RELAY driver commands such as **RELAY**, **Close**, and more reveals that when the user uses the driver commands to program the board, an internal procedure is called. It performs the action such as **Gx6138Close** and then checks to see if an error occurs.

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.

COM Objects and Type Libraries

ATEasy provides extensive support for using COM objects. **COM Objects** are software components based on the Microsoft's Component Object Model. In this model, one application can create or use other application objects by using the **CreateObject** or **GetObject** functions as shown here:

```
ob=CreateObject(”Excel.Application”)
```

The **CreateObject** receives a **Program ID** string, "Excel.Application," to identify the class the user wants to create. Every COM object has a unique program ID to identify itself. Once the object is created, it is assigned to the **ob,** object variable, which you define in your application.

The object **ob** variable can be defined in ***ATEasy*** in two ways. You can use the **Object** data type or you can add a Type Library to the ***ATEasy*** Libraries submodule and use the object class name (or control name) as the data type. Using the first method is called **Late Binding**. The second method is called **Early Binding**. In **Early Binding**, ***ATEasy*** uses the type library and the variable type to check for compiler errors when you build your application. In **Late Binding**, the object properties and methods are assumed to be correct at design time and can be checked only at run-time.

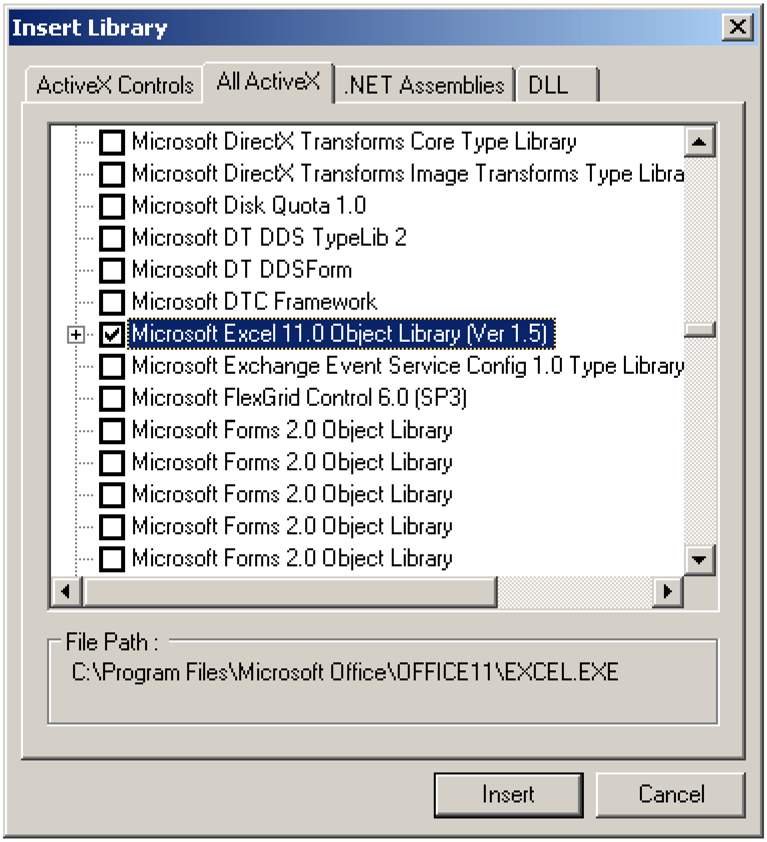
Using the Excel Type Library

You are now ready to create and use the object using the Excel type library (early binding).

▼ To load the Excel Type Library:

1. Right-click on MyProgram **Libraries** submodule and select **Insert Library Below**. The Insert Library is displayed showing the Controls page. The controls page displays the ActiveX controls type libraries installed on your system.

2. Activate the **All ActiveX** Page to display all the available type libraries. Browse through the list of libraries displayed in the list box and select and check the **Microsoft Excel Objects Library**. (If the library is not displayed in the list, you do not have Excel installed on your computer.) The type library file path and file name is displayed below the list box as shown here:



- □ Microsoft DirectX Transforms Core Type Library
- □ Microsoft DirectX Transforms Image Transforms Type Libra
- □ Microsoft Disk Quota 1.0
- □ Microsoft DT DDS TypeLib 2
- □ Microsoft DT DDSForm
- □ Microsoft DTC Framework
- ☑ Microsoft Excel 11.0 Object Library (Ver 1.5)
- □ Microsoft Exchange Event Service Config 1.0 Type Library
- □ Microsoft FlexGrid Control 6.0 (SP3)
- □ Microsoft Forms 2.0 Object Library
- □ Microsoft Forms 2.0 Object Library
- □ Microsoft Forms 2.0 Object Library
- □ Microsoft Forms 2.0 Object Library
- □ Microsoft Forms 2.0 Object Library

File Path :
C:\Program Files\Microsoft Office\OFFICE11\EXCEL.EXE

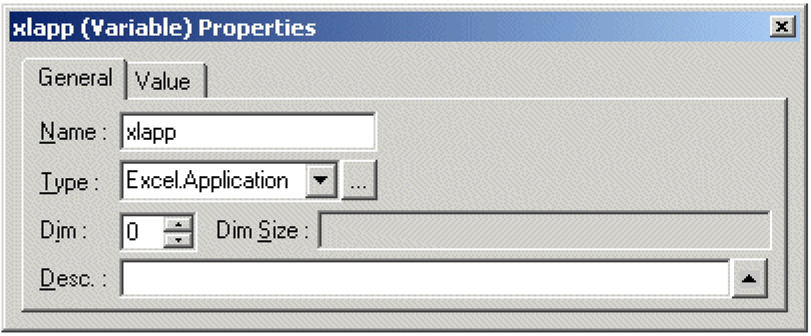
3. Click **Insert**. A new library is created under the Libraries with the default name of **Excel**.

Looking through the Excel library under the Libraries submodule you can see the library contains **Classes**, **Controls** (ActiveX controls)**, Modules** (global procedures similar to the one found in the Internal library), **Types** (Enum, Struct and TypeDef) and **Variables**. **Modules**, **Controls** and **Variables** sub modules are not display since they are empty for this type of library. Expanding **Classes**, you will find the **Application** class, which contains **Methods**, **Properties** and **Events**. You will use the **Caption** and **Visible** property to set and make visible the Excel window caption. You will also use the **Cells** property to read and write to the spreadsheet cells.

Your next task is to create an Excel object from the Application class and to call properties and methods belonging to that class.

▼ To use the Excel Application class:

1. Right-click on the MyProgram **Variables** submodule and select Insert Object Below . Open the variable properties window and rename the variable to **xlapp**. Change its type to **Excel.Application** by selecting it from the browse button as shown here:



xlapp (Variable) Properties

Name : xlapp

Type : Excel.Application

Dim : 0

Dim Size :

Desc. :

.

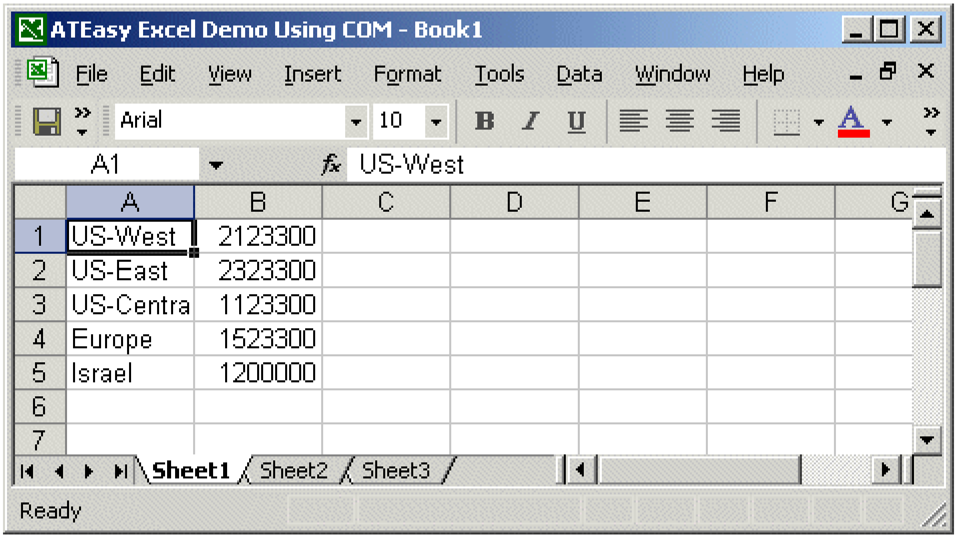
2. Declare the following program variables: **i** as **Long; iSize** as **Long**; **ad** as a one-dimensional array of 5 elements of type **Double**; and **as** as a one-dimensional array of 5 elements of type **String.**

3. Insert a new Task under **MyProgram**. Name the task **Excel** and the test as **Using Early Binding.**

4. Type the following lines of code in the test code view:

```
xlapp=CreateObject("Excel.Application")

xlapp.Visible=TRUE
xlapp.Caption="ATEasy Excel Demo Using COM"
xlapp.Workbooks.Add()

! prepare data
as={"US-West", "US-East", "US-Central", "Europe", "Israel"}
ad={2123300.00, 2323300.00, 1123300.00, 1523300.00,
1200000.00}

iSize=sizeof(as)/sizeof(as[0])

! fill cells
for i=1 to iSize
      xlapp.Cells.Item(i, 1).Value = as[i-1]
      xlapp.Cells.Item(i, 2).Value = ad[i-1]
next

! check if cells got the data
if xlapp.Cells.Item(1, 1).Value<>as[0]
      TestStatus=FAIL
endif
```

5. Click **Doit!** to test the code you have just written. You should see the following Excel window:


ATEasy Excel Demo Using COM - Book1
File Edit View Insert Format Tools Data Window Help
Arial
10
B I U
Sheet1
Sheet2
Sheet3
Ready

A1 US-West

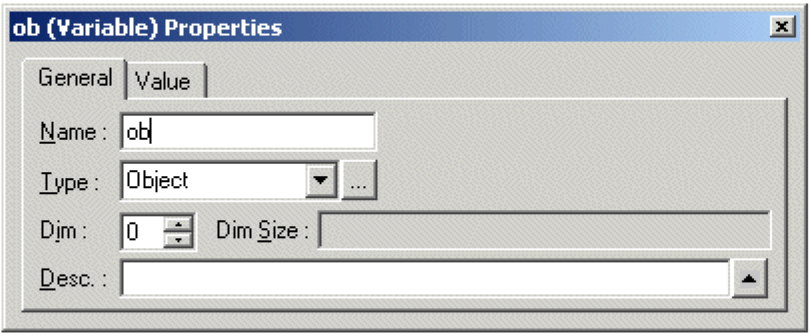
	A	B	C	D	E	F	G
1	US-West	2123300					
2	US-East	2323300					
3	US-Centra	1123300					
4	Europe	1523300					
5	Israel	1200000					
6							
7							

Using the Object Data Type

Your next example will be to create the same test using late binding.

▼ To create an Excel application using the generic object type:

6. Right-click on the MyProgram **Variables** submodule and select **Insert Object Below**. Open the variable properties window and rename the variable to **ob** and change its type to **Object** as shown here:



Name : ob

Type : Object

Dim : 0

Dim Size :

Desc. :

.

7. Insert a new test under the **Excel** task and name it as **Using Late Binding.**

8. Type the following lines of code in the test code view:

```
ob=CreateObject("Excel.Application")

ob.Visible=TRUE
ob.Caption="ATEasy Excel Demo Using COM"
ob.Workbooks.Add()

! prepare data
as={"US-West", "US-East", "US-Central", "Europe", "Israel"}
ad={2123300.00, 2323300.00, 1123300.00, 1523300.00, 
1200000.00}

iSize=sizeof(as)/sizeof(as[0])

! fill cells
for i=1 to iSize
      ob.Cells.Item(i, 1).Value = as[i-1]
   ob.Cells.Item(i, 2).Value = ad[i-1]
next

! check if cells got the data
if ob.Cells.Item(1, 1).Value<>as[0]
      TestStatus=FAIL
Endif
```

As you can see the code for early and late binding is almost identical. The only noticeable difference is the use of the cells property, which in early bind returns a **Range** object and in late bind accepts parameters directly in order to access the worksheet cell.

9. Click **Doit!** to test the code you have just written. You should see the same Excel window as shown in the previous topic.

Using .NET assemblies

.NET assemblies are .NET libraries that contain classes and their definition. Using .NET assemblies from ATEasy is similar to using COM type libraries. You first insert the assembly using the **Insert Library Below** command. Then you activate the **.NET Assemblies** page, this page displays the .NET assemblies and their classes taken from assemblies that are stored in the following folders:

1. Private Assembly can be stored in your application folder or subfolders.
2. Global Assembly Cache (GAC). Located under Windows Assembly folder. This folder is used to store third party assemblies.
3. .NET Framework folder. Located under Windows Microsoft.NET\Framework folder.

ATEasy will also use these folders to locate the assembly at run-time if the assembly file name stored in the library sub-module contains no path (i.e. assemblyfilename.dll). After locating the assembly that you wish to use, check the assembly or individual classes from the assembly that you wish to use and click Insert. ***ATEasy*** then, will create a library that contains the classes, methods, variables and properties as selected. Additional classes and assemblies that are referenced by the classes or assembly you selected are also loaded and displayed below the assembly in the **Referenced Libraries** sub module. ***ATEasy*** support many programming language features that that are supported by .NET including inheritance, multiple constructors, overloading of methods and properties, virtual members, static members, early and late bound objects and more. These features are shown in the **DotNet** example (see **DotNet.prj** in the ***ATEasy*** Examples and Examples\DotNet folders). The example contains a .NET assembly along with it's C# source code that demonstrate some of the programming concepts used by .NET.

Using .NET classes is similar to using COM classes, you create a variable using the **new** operator and assign the result to an object of type **Object** or the class you created as shown here:

```
obCls1_1=new DotNetClass1()
obCls1_2=new DotNetClass1(True)
```

The first line create an object from the DotNetClass1 and the second one creates another one but uses a different constructor that receives True as parameter.

Calling a method or a property or variable of the class is similar to using COM classes (e.g. ob.Method(), or ob.Property or ob.Variable).

Destroying an object is also similar to COM. The following statement will destroy the object:

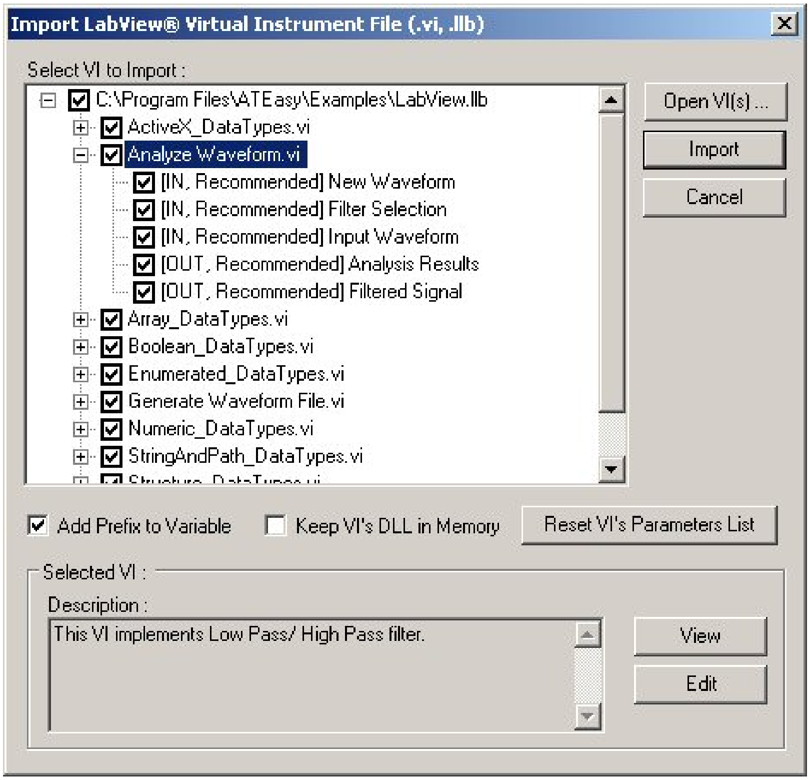
```
obCls1_2=Nothing
```

Using LabView® VI and LLB files

ATEasy allow you to use National Instruments LabView® Virtual Instrument Files from within your application. The two available files types are supported: VI file that holds a single Virtual Instrument panel and LLB that hold multiple VIs.

To use a VI you must import it first using the **Import LabView Virtual Instrument File (.vi, .llb)…** command . Once you import the LLB or VI files ATEasy will create a procedure for each one of the VI that was imported. These procedures contain code that is using the ATEasy internal function **GetVi** that returns a LabView ActiveX object used to call the VI. Currently ATEasy supports VI's created with LabView version 7.0 and 7.1 and you must have a matching version of LabView run-time version in your machine as well as the VI referenced libraries in order to call the VI. The LabView development environment is not required in order to call the VI. The procedures created by the wizard contain parameters as required by the VI. Calling these procedures is similar to any ATEasy procedures.

The **LabView.prj** and the **LabView.llb** in the ATEasy Examples folder provides an example for using LabView. The following dialog displays the **Import LabView Virtual Instrument File** dialog importing the LabView.llb provided with ATEasy:


Import LabView® Virtual Instrument File (.vi, .llb)
Open VI(s) ...
Import
Cancel
Reset VI's Parameters List
View
Edit

Select VI to Import :

- ☑ C:\Program Files\ATEasy\Examples\LabView.llb
  - ☑ ActiveX_DataTypes.vi
  - ☑ Analyze Waveform.vi
    - ☑ [IN, Recommended] New Waveform
    - ☑ [IN, Recommended] Filter Selection
    - ☑ [IN, Recommended] Input Waveform
    - ☑ [OUT, Recommended] Analysis Results
    - ☑ [OUT, Recommended] Filtered Signal
  - ☑ Array_DataTypes.vi
  - ☑ Boolean_DataTypes.vi
  - ☑ Enumerated_DataTypes.vi
  - ☑ Generate Waveform File.vi
  - ☑ Numeric_DataTypes.vi
  - ☑ StringAndPath_DataTypes.vi
  - ☑ Structure_DataTypes.vi

☑ Add Prefix to Variable ☐ Keep VI's DLL in Memory

Selected VI :

Description :

This VI implements Low Pass/ High Pass filter.

CHAPTER 10 – WHERE TO GO FROM HERE

About Where to Go from Here

This chapter describes how and where to find information about topics not covered in this manual.

Topic	Description
More about ***ATEasy***	Describes some of the features available that are not covered in this manual and provides information where to find additional information.
Examples	Describes a quick overview of the examples that are provided with ATEasy.

More about *ATEasy*

ATEasy contains many features not covered in this manual. The following list describes some of the features available that are not covered in this manual:

- Configuring and using the COM (serial), GPIB, VXI, File and WinSock interfaces
- Using TCP/IP communication
- Using DDE to communicate between applications
- Using the WIN32 API
- Creating and using threads and synchronization objects
- COM/ActiveX Objects and Multithreading
- Multiple UUTs, parallel and sequential execution modes
- The Internal Library classes, controls, procedures, variables and types
- Internal Variables
- Complete description of the ***ATEasy*** programming language and statements
- Using interrupts and Interface Events
- Error and exception handling
- Managing users
- Document version control and text formats

- Using module events
- Form types: MDI, MDI Child and Normal forms
- Adding and Using Menus with Forms
- Drawing on forms
- HTML and the Log control
- Logging and customizing your test results
- ATML support
- Creating and using ***ATEasy*** DLLs

The ***ATEasy*** *User's Guide* and the ***ATEasy*** online help contain a complete coverage of these topics. Additionally, you can look at the examples and drivers that were copied to your Examples and Drivers directories during Setup.

ATEasy Examples

ATEasy is provided with several examples that can be used to see how to implement various application and features as required. This topic will describe briefly the available examples provided in the ***ATEasy*** Examples folder. You can load the examples described here by opening one the workspace files in that folder as described here:

Examples Workspace Files

Workspace File	Description
Examples.wsp	Workspace file for all example projects.
Basic.wsp	Workspace file for Language, Forms and Test Executive projects.
Communication.wsp	Workspace file for ComChat and WsChatMT projects.
Excel.wsp	Workspace file for Excel project.
ATEasy2.wsp	Workspace file for ATEasy2 examples.

Other workspace files and examples projects may be available in that folder.

Examples

The following examples are provided; Additional examples may be available, see the ***ATEasy*** examples folder:

AdoDB Example

This example demonstrates the use and programming of databases using a Microsoft ADO Active Data Objects ActiveX library. The example create read and write a data base, tables and records. The example is using Microsoft Access and can be easily changed to use any data base format that has ODBC driver. You must install the Microsoft ActiveX Data Objects 2.8 Library from www.microsoft.com.

Files included: AdoDB.prj and AdoDB.prg.

ATML Example

This example demonstrates ***ATEasy*** support for IEEE Standard 1671- Automatic Test Markup Language (ATML) for Exchanging Automatic Test Equipment and Test Information via XML support. The support is provided using the ***ATEasy*** ATML.drv. The example generates ATML Test Results (.xml file) from an ***ATEasy*** program, ATML Test Description (.xml file) from an ATEasy program. It also transforms ATML TestResults to formatted test log using Style sheets (XSL files): AtmlToText.xsl, AtmlToHtml.xsl or AtmlToHtmlStyle1.xsl.

Files included: ATML.prj, ATML.prg, ATML.sys and the ATML.drv and its support files (xsd schema files and xsl style sheet files).

ComChat Example

This example demonstrates the use and programming of COM or serial ports. The application creates a window that is used to type text used to send to a COM port. Any data received from the port is appended and displayed in the window. The example uses ***ATEasy*** COM interrupts to display the data received back. The example also shows how to read and write from Windows INI files using windows API.

Files included: ComChat.prj, ComChat.sys and ComChat.drv.

DLL Example

This example demonstrates how to write DLL using Visual Studio (C) and use it from ***ATEasy***. Two DLLs are provided with this example. The first DLL, written in C with Microsoft Visual Studio (dll.dll). The second DLL (ATEasyDll.dll) is written using ***ATEasy*** and sources are provided in a separate ATEasy project (ATEasyDll.prj). The provided program contains several tests that call DLL functions with various parameters types.

Files included: Dll.prj, Dll.prg. DLL files (in the Examples\DLL folder): Dll.dll, Dll.dsp, Dll.cpp, Dll.h, StdAfx.cpp and StdAfx.h. ATEasyDll files (in the Examples\ATEasyDLL folder) – see the following paragraph.

ATEasyDll Example

This example demonstrates how to create an ATEasy DLL (ATEasyDll.dll) and use it from ATEasy and from other languages such as C and VB. This project contains a driver that includes several exported functions, thus illustrating how to export functions via ATEasy DLL. The example includes three additional projects that shows how to call the ATEasy DLL from Microsoft Visual Basic application (ATEasyDllVb.exe), Microsoft Visual C project (ATEasyDllC.exe) and ATEasy application (see DLL example).

Files included: ATEasyDll.prj, ATEasyDll.sys and ATEasyDll.drv. Application files for VB and C in the Examples\ATEasyDll folder.

DotNet Example

This example demonstrates how to write and use a .Net assembly classes and how to use the classes from ATEasy. The assembly is created using Microsoft Visual Studio .NET using C# programming language. The provided program contains several tests that create and use the various exported classes in the .NET assembly with various parameters types. You must have the Microsoft .NET framework installed in your computer. It is recommended to use Windows XP or newer since prior versions of Windows, OLE does not support 64 and unsigned types commonly used in .NET.

Files included: DotNet.prj, DotNet.prg.

DotNet assembly source files (in the Examples\DotNet folder): DotNet.dll, DotNet.dsp, DotNet.cs

Excel Example

This example demonstrates how to use Microsoft Excel using ActiveX/COM and DDE (requires that you will have Excel installed in your machine). The example creates an excel workbook with two spreadsheets, the first spreadsheet contain data and the second contains a chart that plots the data from the first sheet. The example shows three ways of implementing this:

- The first task, test is using the Excel COM object using late binding.
- The first task, second test is using the Excel COM type library using early binding.
- Second task, implement the example using ***ATEasy*** DDE functions.

You must have the Microsoft Excel installed in your computer for this example to run.

Files included: Excel.prj, Excel.prg.

FaultAnalysis Example

This example demonstrates the use of the fault analysis driver (FaultAnalysis.drv) along with the TestExec.drv and Profile.drv. The example contains a program with many dummy tests and preset test result set inside the tests. A condition file (FaultAnalysis.cnd) used by the Fault Analysis driver contains conditions that can be analyzed, displayed and print to the test log at the end of the run of the program.

Files included: FaultAnalysis.prj, FaultAnalysis.prg, FaultAnalysis.sys (contains the FaultAnalysis.drv, Profile.drv and TestExec.drv) and FaultAnalysis.cnd.

Fl884x Example

This example demonstrates GPIB programming and how to build a GPIB interface instrument driver. The example uses a Fluke 8840A digital multimeter. The example demonstrates the use of IO Tables, Commands, GPIB interface and programming using IO Tables or using the internal library GPIB functions.

Files included (in the Drivers folder): Fl884x.prj, Fl884x.prg and Fl884x.drv.

Forms Example

This example demonstrates how to create and use ***ATEasy***'s Forms. The example is divided to tests each creates a different form with different sets and controls and menus.

Files included: Forms.prj and Forms.prg.

HW Example

This example demonstrates the use of the Geotest Hardware Access Driver (HW). The HW driver features can be accessed from the Windows **Start, Geotest, HW** menu in the taskbar. The example shows how to read the computer PCI bus configuration and resources. The HW driver can be used to write prototype driver and to test PCI/PXI/ISA boards.

Files included: HW.prj, HW.prg HW.sys and HW.drv.

Hp34401aFP Example

This Example demonstrates Function Panel driver for DMM loaded with the HP34401A digital multimeter driver from National Instruments .fp file. It contains a program utilizing FP functions.

Files included: Hp34401aFP.prj, Hp34401a-FP.prg, Hp34401a-FP.sys and Hp34401a-FP.drv

IOTables Example

This example demonstrates creating and using of IO Tables and the File driver interface. The example driver IoTables.drv contains many IO tables with different operations and modes. The example uses a temporary file to perform the input and output operations.

Files included: IoTables.prj, IoTables.prg, IoTables.sys and IoTables.drv.

IviDmm Example

This example demonstrates using the standard IVI-C and IVI-COM class DMM drivers. The example contains two tasks; the IVI-C task requires installing National Instrument driver for HP34401A while the IVI-COM task requires the Agilent driver. You will need to download and install the drivers from these vendors. VISA resource manager also needs to be configured with the DMM logical name before using the example.

Files included: IviDmm.prj, IviDmm.prg, IviDmm.sys and IviDmm.drv.

LabView Example

This example demonstrates using and calling LavView VI and LLB files from ATEasy.. The example program contains procedures generated by importing the ViExamples.llb VIs to ATEasy suing the Insert LabView VIs command. The example requires National Instruments LabView 7.0 run-time (www.ni.com) installed in your computer (ATEasy will work with LabView v6.0 or above).

Files included: LabView.prj, LabView.prg and LabView.llb.

Language Example

This example demonstrates the ***ATEasy*** programming language. The examples contains several tasks to demonstrate Control statements, expressions and assignments, user defined procedures, using variants variables, internal library, using DLLs, using the log window and customizing the log output, interrupts, multi-threading, conditional compilation and error and exception handling.

Files included: Language.prj and Language.prg.

MMTimer Example

This example demonstrates how to use the Windows Multimedia timers to perform 1ms accurate timing events. The example is provided with DLL sources (using MS VC++ 6.0) that can serve as a template for an application that requires performing operations in an accurate timing manner.

Files included: MMTimer.prj and MMTimer.prg

DLL Files (in Examples\MMTimer folder): MMTimer.dll, MMTimer.dsp, MMTimer.cpp, MMTimer.h, StdAfx.cpp and StdAfx.h.

ModuleEvents Example

This example demonstrates ***ATEasy***'s program, system and driver module events. Running or using the debug command on this example will generate trace output from each of the ATEasy's module event procedures, which shows the sequence in which these events are called by ***ATEasy***.

Files included: ModuleEvents.prj, ModuleEvents.prg, ModuleEvents.sys and ModuleEvents.drv.

MultiPad Example

A complete text editor with MDI user interface (Multi Document Interface) implemented with ATEasy. This example is provided to show how to use the MDI user interface, MDI Form, MDI Child Form, AMenu, AToolbar, ACommonDialog (file open, save and print, font selection), AImageList, ATimer and more.

Files included: Multipad.prj, Multipad.sys and Multipad.drv.

MyProject Example

This example is provided as a reference to the files created in this manual. It contains all the examples created in the ***ATEasy*** getting started manual.

Files included: MyProject.prj, MyProgram.prg, MySystem.sys, MyDMM.drv (HP34401a driver) and MyRELAY.drv (GX6138 driver).

ProcessDiagnostics Example (v8)

This example is provided as a tool to diagnose memory, threads or file handle leaks created by calls to external libraries such as DLLs. .Net or COM. The example is using the ProcessDiagnostics.drv driver provided with ATEasy. The driver can report the current process memory consumption, handles count,; thread count, modules and CPU usage.

Files included: ProcessDiagnostics.prj, ProcessDiagnostics.prg, ProcessDiagnostics.sys, and ProcessDiagnostics.drv.

Profile Example

This example shows how to create profile programmatically and dynamically. First it selects the second profile of the given profile file, Profile.prf in the ATEasy Examples folder. When the Test Executive driver displays the selected profile in its tree view, you can run the profile. At the end of a program run, it asks you to perform its diagnostics if there are any failed tests – upon which if you respond Yes, then it creates a diagnostic profile and a file, "Diagnostics.prf" in the current folder and runs it. The newly created profile includes all failed tests.

Files included: Profile1.prg, Profile2.prg, Profile.sys, Profile.drv and TestExec.drv.

StdIoProcess Example (v8)

This Example shows how to control a console application. The example shows 3 console applications. The windows Command prompt (cmd.exe), TCL interpreter and PERL interpreter. The PERL interpreter used in this example is Strawberry Perl (http://www.strawberryperl.com). The TCL interpreter used in this example is ActiveState ActiveTcl (http://www.activestate.com/activetcl). The example uses the ATEasy StdIoProcess driver that can be used to control console applications by redirection their standard input, output and error pipes. The driver offers synchronous and asynchronous way to redirect the input to the console application.

The driver uses .NET System.Diagnostics.Process class).

Files included: StdIoProcess.prj, StdIoProcess.prg, StdIoProcess.sys, StdIoProcess.drv, StdIoProcessAdd.tcl, StdIoProcessAdd.pl

TestExecMini Example

This example can be used as a basic building block for building your own test executive. It displays a form that allows the user to select a test from the program tests and run it.:

Files included: TestExecMini.prj, TestExecMini.prg and TestExecMini.drv.

TestExec Example

This example demonstrates the use of the TestExec and the Profile drivers and their user interface. A Profile file TestExec.prf is provided and contain several profiles ready to run.

Files included: TestExec.prj, TestExec-1.prg, TestExec-2.prg, TestExec.sys (uses Profile.drv, TestExec.drv and TestExec.prf).

TestExecMutipleUUTs Example (v8)

This example demonstrates the use of the TestExec to run multiple UUTs in parallel and sequential mode. The example is similar to the TestExec example with the exception of additional code that was added to the System.OnInit() event to setup and run multiple UUTs

Files Included: TestExecMultipleUUts.prj, TestExec-1.prg, TestExec-2.prg, Language.prg, TestExecMultipleUUts.sys, Profile.drv, and TestExec.drv.

TestExecUser Example

This example demonstrates the use of Test Executive users file. The users file allows you to create Multi users environment where an administrator sets each user group's privileges.

Files included: TestExecUsers.prj, TestExec-1.prg, TestExec-2.prg, TestExecUser.prg , TestExecUser.sys, Profile.drv and TestExec.drv.

VB (Visual Basic) Example

This example demonstrates how to use an ActiveX object, ActiveX control, enumerated type and structures created using Microsoft Visual basic. The example is provided with sources safe for the Visual Basic project. You should have VB 6.0 in order to compile the VB examples and the VB 6.0 run-time in order to run it.

Files included: VB.prj, VB.prg.

VB Files (in Examples\VB folder): VB.ocx,, ATEasyVB.vbp, Class1.bas, CPShapeL.ctl, CPShapeL.ctx and CPShapeL.pag.

WsChatIE Example

This example demonstrates how to use WinSock and TCP/IP communication protocol to communicate between applications. The application creates a window that is used to type text used to send to a TCP/IP port. Any data received from the port is appended and displayed in the window. The application can be set to act as a client or as a server. The example uses ***ATEasy*** Interface Events to receive data from the port (client or server) or accept the client connection (server).

Files included: WsChatIE.prj, WsChatIE.sys and WsChatIE.drv.

WsChatMT Example

This example demonstrates how to use WinSock and TCP/IP communication protocol to communicate between applications. The application creates a window that is used to type text used to send to a TCP/IP port. Any data received from the port is appended and displayed in the window. The application can be set to act as a client or as a server. The example uses a separate thread to receive data from the port (client or server) or accept the client connection (server). This example is similar to the WsChatIE, however, the communication is done here in separate thread instead of using interface events.

Files included: WsChatMT.prj, WsChatMT.sys and WsChatMT.drv.

INDEX