Instruction Manual for Motion Server & Servo Application Workbench

March, 1999

Douloi Automation, Inc.
3517 Ryder Street
Santa Clara, CA 95051-0714

Voice (408) 735-6942
Fax (408) 735-6946
EMail support@douloi.com

Table of Contents

Chapter 1

1) Introduction

Welcome!

Welcome to Motion Server and Servo Application Workbench, tools to simplify and accelerate the creation of motion control applications.

Douloi Automation wants to encourage your project's success. Free technical support is available to answer your questions, assist you through trouble-spots in product use, and to recommend strategies and approaches for solving different aspects of a motion control problem. Sample code, application prototypes, and application notes can be provided to response to specific questions you may have. We would much rather have you call and get answers than to be frustrated or slowed in your automation project. Please feel free to contact us at:

- voice (408) 735-6942
- fax (408) 735-6946
- EMail support@douloi.com

Motion Server is a hardware component that can communicate through several methods. The easiest environment to use Motion Server is Microsoft Windows. In Windows, users can describe real-time activities that execute on the Motion Server card independent of the host processor. These behaviors can then be retained in Motion Server for use in other operating systems. Motion Server can be used with any Windows language system which communicates to Windows Dynamic Link Libraries (DLLs) such as Visual Basic or Turbo Pascal for Windows. Servo Application Workbench is a particular language system which greatly simplifies the use of Motion Server by providing convenient access to the functionality of Motion Server as well as providing components for creating control panels for servo or stepper motor controlled machines. After creating a motion control application with SAW, the system can be configured to start the application independently, from the Program Manager or AUTOEXEC.BAT file as if the application was a conventional Windows application.

Objective of Document

The purpose of this document is to provide information in the order you will need it in the course of setting up and using a system. Specific details on features are found in the SAW on-line help reference manual accessed through the Help menu selection in SAW or the Help icon in the Servo program group. The on-line help serves as the Reference Manual for the system. This document serves as the User Manual. Information can be found by either following information paths from the index or searching by keyword or topic.

The primary method of explaining Motion Server and SAW is through interactive tutorials and application "sketches" which illustrate how different system features are used. A section is included illustrating how to access Motion Server from other language systems such as Visual Basic and Visual C++.

Motion Server Specifications

Motion System

- 486 DX5 128 MHz 32 bit processor
- 4, 8, 12, or 16 axes per system
- Servo or Stepper on per-axis basis
- Multiple independent coordinated axis groups
- Trapezoidal and S-Curve profiling
- Custom profiling at application level
- 32 bit position management
- Sample rates from 1 to 4 kHz
- Linear, circular, curve interpolation
- Electronic gearing with phase adjust
- Electronic camming
- Tangent servo
- Master/slave coordination
- High speed registration
- Kinematics
- Motion superposition
- Coordination tailoring
- On-board real-time operating system supporting 12 seperate activities as well as motion control
- 48 general purpose configurable I/O

- 1 Capture signal per axis
- User Disable signal
- 2 amp enable signals per axis, one active high, the other active low
- watchdog safety system

Servo Capabilities

When configured to run a servo motor the hardware provides:

- 4 MHz quadrature inputs with 3 bit filters for 4 axis, 1 MHz quadrature rate for 16 axis
- high speed position capture
- high speed position compare
- +/- 10 volt command signal with 12 bit resolution

Stepper Capabilities

When configured to run a stepper motor the hardware provides:

- 2 Mhz step rate for 4 axis, 500 kHz step rate for 16 axis
- configurable step pulse polarity

Timer Event

Motion Server provides motion control functions by responding to a timer which occurs generally at 1 kHz although the frequency is programmable. This times event performs three major functions.

The first function is control law execution. Servo control is accomplished with the familiar zero, pole, integrator filter used in many motion control systems. This PID control law operates at a 1 kHz sample rate providing comfortable closed loop system frequencies of 100 Hz and below. Stepper motor control is accomplished by updating pulse generating electronics at a frequency of of 1 kHz providing continuous velocity control of stepper motors. The second function of the timer event is motion profiling. Motion Server is able to profile motion for up to 16 physical axes. These axes can be combined in different arrangements to form various coordinated multi-axis groups. Any particular axis group can perform coordinated motion along an arbitrary path. Multiple axis groups can perform motion concurrently and independently. The motion profiler uses a dynamic profiling technique which permits changing profile parameters on the fly including acceleration, deceleration, slew speed, and in some cases destination and motion type. This permits motion mode "splicing" without stopping. For example a positioning move can be changed to a jog at a new speed on the fly.

The third timer event function is multitasking. Multiple user-written motion application programs can be resident in Motion Server. The timer event contains a multitasker which activates and manages the operation of these programs.

Multiple Motion Application Threads

As many as 12 separate motion application "threads" or programs (which are distinct from motion profiles) can be running concurrently and independently at any particular time. These programs are written in Douloi Pascal, a dialect of Object Pascal. Programs can communicate to each other through shared data structures. They can access the motion control system, communicate to auxillary analog input and serial port I/O boards attached to Motion Server, communicate with Windows applications created by the Servo Application Workbench, and to the disk file system if SAW is present.

Microsoft Windows '95 and Windows NT

Microsoft Windows serves as the most common development and target environment for motion control applications using Douloi products. The familiar interface aids both developers and users of the resulting applications reducing the developers learning curve and the operators training time. Motion Server can be used with other operating systems through various communication methods available.

Long-Slot ISA Format

Motion Server occupies a single ISA "long" slot. The end of the Motion Server card furthest from the mounting bracket holds the on-board 486 processor. The heat sink and fan assembly protrudes from the board further than the board-to-board spacing preventing the placement of another long-slot card immediately in front of Motion Server, however shorter cards can fit if necessary. Host performance does not effect Motion Server's real-time performance, however some operations are performed by the host on behalf of Motion Server when Motion Server sends "mail" requesting that these operations are done on its behalf. Performance of these "non-real-time" operations is enhanced with a faster host.

Servo Application Workbench

Servo Application Workbench (SAW) is a Windows application which greatly simplifies the creation of multithreading motion application programs and operator control elements to direct them. Applications may

contain conventional Windows controls such as buttons and text items as well as more specialized controls such as components available in the software catalog browser.

Inside Servo Application Workbench is a high level language compiler. The compiler changes the descriptions of the motion applications into native 32 bit 486 object code which executes on Motion Server very quickly. The compiler “knows” about the motion system, the multithreading system, and Windows. This permits the application developer to access different system resources in a consistent way without having to worry about how these resources are being provided.

Servo Application Workbench allows the developer to construct motion applications in a “clip art” fashion by pasting pre-fabricated parts and assemblies into the application. After “screen painting” the application and filling in the program’s behavior Servo Application Workbench compiles the motion application programs and creates the associated Windows application to operate them. This ability to create new real-time behavior and download into Motion Server is constrained to the Windows environment because the language compiler is a Windows DLL. However, new motion controller capabilities (beyond the standard command set) can be created in SAW, downloaded into Motion Server, and remembered in "flash" memory for use under another operating system.

Methods Of Use

Motion Server can be used in a number of ways. Certain capabilities are available only in certain development methods. The following sections describe resources available.

Servo Application Workbench

Servo Application Workbench is the easiest method for development of real-time machine behavior. This behavior can be "downloaded" into the controller and invoked from a control panel also written in SAW, from other Windows programs, or from binary or ASCII commands.

32-bit Dynamic Link Library

Dynamic Link Libaries are a common and simple method of adding features to any Windows language system. A dynamic link library provides procedures and functions to control the Motion Server hardware. The dynamic link library uses Binary Commands which are described in detail in the Binary Command Manual.

ASCII Commands

ASCII commands provide a simple method of accessing the basic functions of Motion Server including single axis and multiple axis coordinated motion and I/O. Characters are sent through an RS-232 port on an accessory card which connects to Motion Server.

Binary Commands

Binary Commands provide low-level "register" access to Motion Server without requiring that position information and command parameters be converted into an ASCII format. Binary Commands are constructed by the 32 bit Dynamic Link Library or can be constructed directly with register reads and writes for use with non-Windows operating systems.

Chapter 2

2) Setting Up the System

Purpose

The first step in preparing to use Motion Server and the Servo Application Workbench is configuring and installing the software and hardware. A procedure for hooking up Motion Server to other motion elements is provided along with checks to insure the integrity of your work.

If at any point during the setup process you have a question please feel free to call Douloi Automation for advice on how to best proceed.

System Requirements

Servo Application Workbench requires an IBM or compatible PC ("Wintel" computer) running Microsoft Windows version 3.1, 3.11 (Windows for Workgroups) or Windows '95. Also required is a VGA monitor, 8 megabytes of memory, mouse, 2 megabytes of available hard disk space.

This document presumes that you are familiar with Microsoft Windows. Pascal familiarity is helpful but not necessary.

Software Installation

Software needed to use Motion Server and SAW may be installed in your system by running double clicking the SETUP.HTM file in the root directory of the CD-ROM. Instructions specific to particular browsers are described in this file.

Servo Application Workbench Installation

If you've licensed Servo Application Workbench install the SAW Full Development Environment. If you have not licensed SAW, Douloi Automation recommends that you install the SAW Run-Time Edition. This provides access to diagnostic software and allows Douloi Automation to guide you through diagnostic procedures if required. The Run-Time Edition allows you to load SAW applications but not to save any of your work. Make sure you choose the appropriate operating system.

Think & Do Installation

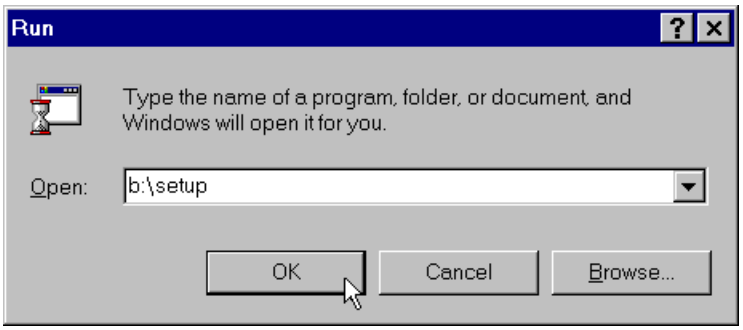
If you are using Think & Do Software, install the Binary Command Interpreter under the Windows NT section. Think & Do communicates to Motion Server through binary commands. Details of binary commands are explained in the Binary Command Manual.

Visual C++ and Visual Basic

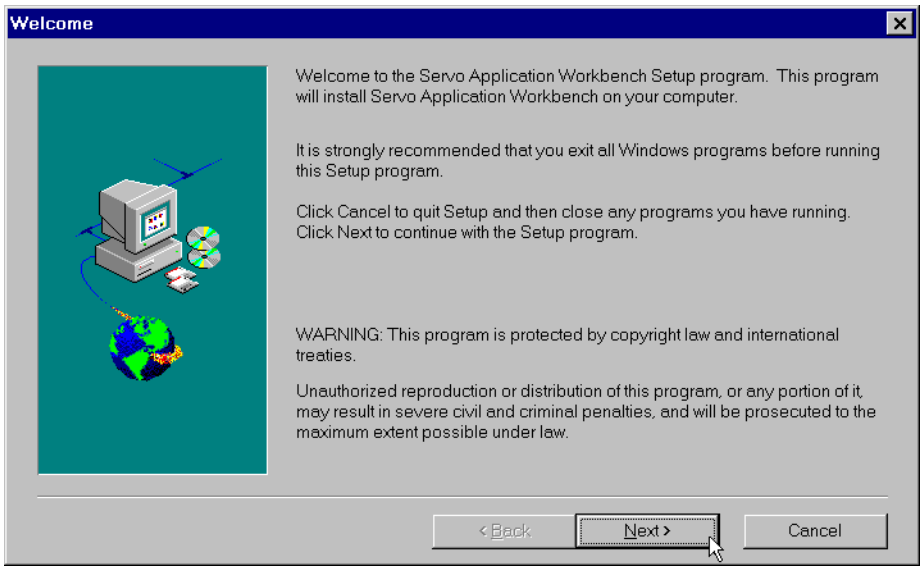
If you are using Visual C++ or Visual Basic you should install the Binary Command Interpreter for the operating system you intend to use. Details of binary commands are explained in the Binary Command Manual.

Example Setup Procedure

After choosing a particular setup option a setup procedure similar to the following will appear. Different selections have difference dialogs however this is representative.

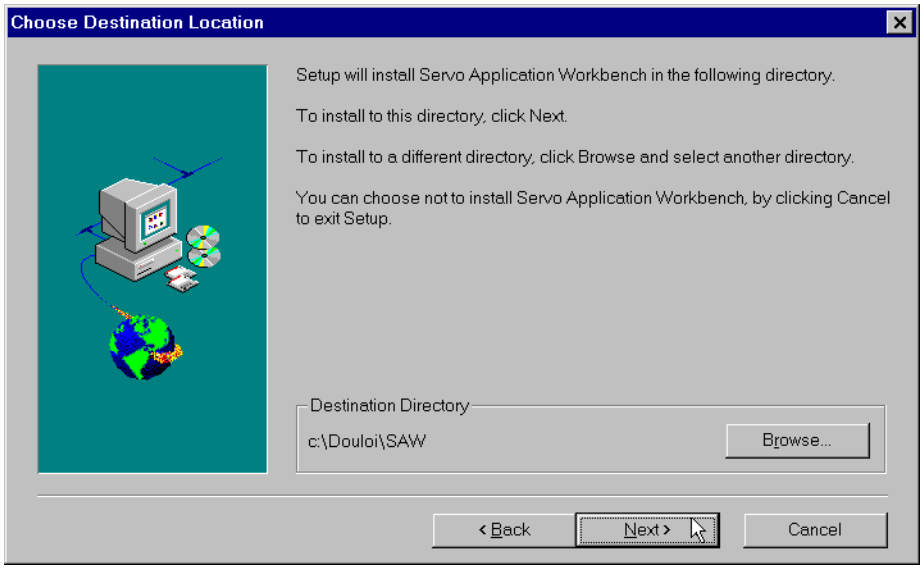


An installation screen will appear. You will be shown the following introductory dialog:

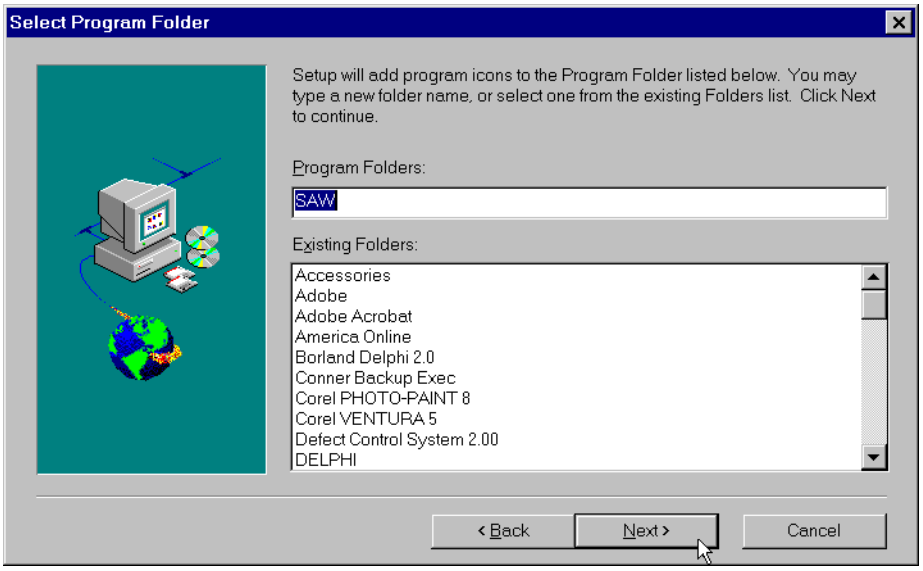


Select the "Next" button to continue.

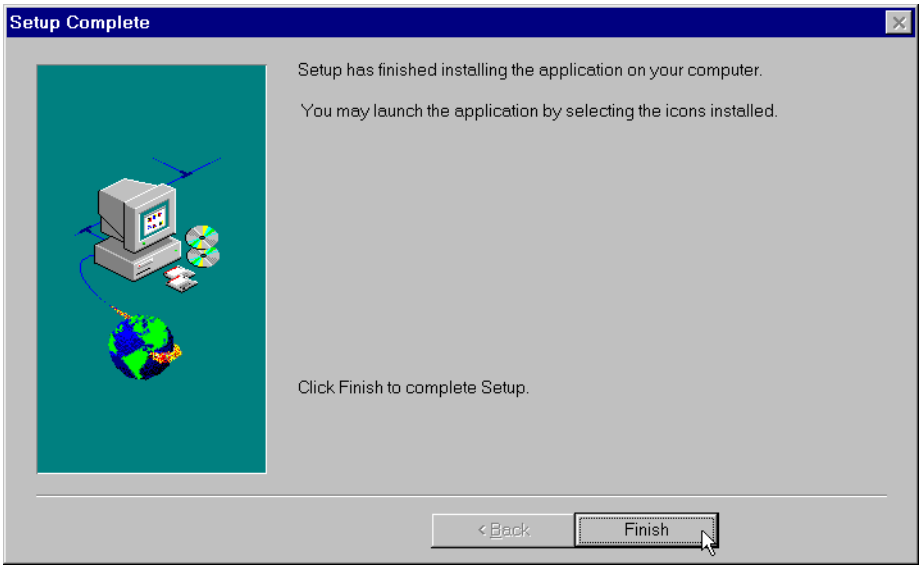
The next decision is where should the software be placed. The following dialog suggests c:\Douloi\SAW as the default directory:



A default program folder named SAW is offered:



After files are installed onto the hard disk the following dialog concludes the installation:



Hardware Installation

Detailed descriptions of each Motion Server connector and signal are found in the "Connectors" chapter towards the end of the manual. Different amplifier vendors choose different names for signal functions. If you have any confusion regarding how to hook up Motion Server to your servo amp, stepper drive, or IO module please contact Douloi Automation for advise. Faxing a data sheet of the signals definition for the device in question helps us best help you.

The introduction of any new piece of hardware in a PC has the possibility of causing hardware problems. Before installing Motion Server perform a complete backup of your computer. If you are unfamiliar with backing up your system contact Douloi Automation for a recommendation. The value of your work can easily be lost by a hard disk failure or hardware conflict. Safeguard you work by having a consistent pattern of backups After backing up your system install the hardware into your PC.

External Connections

Connections from Motion Server to other parts of the system can be made with IDC to terminal breakout modules, or by using D-style IDC connectors on the split-apart axis cables creating identical axis connectors.

During initial setup it is most desirable to work with a motor system which is not attached to a machine, or if attached has the freedom to spin without encountering limits. It is not a significant problem if this is not the case however additional care should be exercised when setting up a limited-motion system.

External connections will be made “incrementally”, checking out the correctness of the connections as you move along. An example will be walked through for a single axis. Repeat the following procedure for each axis in the system.

Servo Motor Setup

Hooking up the Encoder

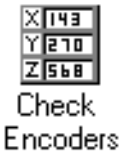
Motion Server expects to interpret quadrature encoders. A +5 voltage is provided on the axis encoder connector to power the encoder. If you have a +12 volt encoder the signals can be properly interpreted however you'll need to supply the encoder power from an alternate location such as a disk-drive connector plug.

The encoder may be two channel or three channel, single ended or differential. The type, if not explicitly known, can be inferred from the types of signals being returned. Channels are typically labeled A and B for the main quadrature information, and I for the index pulse, a once-per-revolution signal useful for initialization or position integrity checks. If the encoder has differential outputs there should be corresponding /A, /B, and /I signals, one complement signal for each line. **If your encoder does not have differential outputs simply leave the differential inputs disconnected. An internal voltage reference provides a suitable level for these lines in the absence of an external signal. Do not ground the /A, /B, or /I inputs. Ground is not a suitable voltage reference and may cause the encoder to appear to not work.**

Confirming Proper Encoder Operation

Regardless of whether or not Servo Application Workbench has been purchased for development, a run time version has been provided to run some installation instruments to assist you in the course of setting up your system.

Start Windows and double click on the Check Encoder icon in the SERVO group. The icon looks like this:



After a few moments a position display instrument should appear indicating the position of the encoders. Rotate the motor and note the change in the position display window for that axis. Press the "reset" button to zero the current position and arm the index capture. Rotating the motor should

cause a position change, and if rotated at least one rev, should also indicate that the index pulse was detected by changing the "Armed" status to "Tripped" (assuming you have a three channel encoder).

Repeat this encoder installation procedure for each axis in your system. After checking the encoders close the application by double clicking on the system menu.

Connecting Motion Server to Servo Power Amplifiers

Motion Server takes position information from the encoders and in combination with directives from the application program creates a motor command which is a request to the power system for a certain amount of motor torque. This motor command is expressed as an analog voltage in the range +10 volts to -10 volts, plus voltage indicating a request for positive torque and negative voltage indicating a request for negative torque. This is a common torque request format for many amplifier suppliers. Connect the motor command signal to the torque command input of the amplifier. If the amplifier input takes a differential pair then use the Motor Command signal for the "plus" side of the differential receiver on the amp and a ground wire for the "minus" side. Use a twisted pair for these two motor command and ground signals.

Modern amplifiers are quite versatile and often have a number of different operating modes. For use with Motion Server the desired mode is most likely called by one of the following names in the amplifier documentation:

Torque Control
Current Control
Transconductance Mode
Position Control Configuration

The objective of the mode is to provide a motor current proportional to the analog input voltage. It is important to disable any “integration” feature of the amplifier. These are most often encountered with amplifiers designed to directly support PI control internally. Motion Server provides PID control but will conflict with any other agents attempting to do the same.

Often the default gain of an amplifier, i.e. amps output/volts input, is quite high. The most useful amplifier gain setting is provided by setting the amplifier to produce its maximum positive desired output at +10 volts, and the maximum negative desired output at -10 volts. This gives Motion Server the maximum possible numeric range for working with your amplifier system. Consult amplifier documentation for gain values and the possible need to adjust the amplifier gain. Note that adjusting the amplifier gain on a live system may result in sudden unexpected movement. Gain

settings are best adjusted on the bench with an inductive load rather than the actual motion system. Consult your amplifier documentation and vendor for information.

The safety of the system will be enhanced by using the amplifier enable outputs from Motion Server if the amplifiers you are using have amplifier enable inputs. Amplifier enable is a signal used to instruct the amplifier when it should interpret the analog signal. If the amplifier enable line is not active, the amplifier disregards the analog signal. This provides Motion Server with two different ways of requesting 0 amplifier power, through expressing 0 voltage to the analog input and by disabling the amplifier. If the amplifier has an amplifier enable input Douloi Automation recommends using it to improve the safety of your system. Use of the amplifier enable signals is particularly important if the host computer power can be turned off while the amplifier power remains on as the analog command voltages can drift during the shutdown of the computer power supplies. If you have a choice, it is much better to choose amplifiers which default "off" without being deliberatly turned on by the controller, rather than amplifiers which are active by default.

Confirming Proper Amplifier Operation

Double click the Check Amps icon. The Check Amps icon looks like:

This instrument is useful for checking the basic stability of the servo system. The main concern at this point is whether the motor wires are hooked up correctly or are backwards. In a closed loop servo system the commands sent to the motor direct the motor towards a desired goal reducing an error and converging towards the goal. If wiring in the system is backwards the motor will be directed away from the desired goal increasing error and diverging. "Check Amps" identifies which of these two cases is currently configured. Select the axis to test with the “Select Axis” button and perform the test with the “Perform Test” button.

Things can get “backwards” in many different places in a closed loop servo system and it is generally easier simply to measure what you have rather than try to determine ahead of time what you have. If "Check Amps" determines that something is backwards you have several options on what to change. Inverting the A and B channels of the encoder can solve the

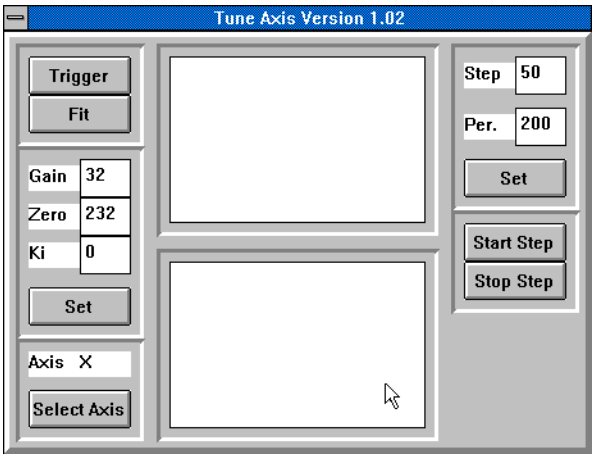
problem, however this can be inconvenient to do if you are using plug in cables and modules. It is not a good idea to swap wires inside an encoder cable since this makes the cable “special” in a somewhat invisible way and replacing the cable with a straight through cable can cause the system to “mysteriously fail” if that swapped wire cable is not carefully marked. The most suitable place to switch things is to invert the motor wires on the motor. Do not attempt to invert the plus and minus voltages on the amplifier. In a closed loop system the “positive” direction is established by the encoder, not by the motor leads so there is no penalty for reversing the motor leads.

It is also possible to “reverse” Motion Server in software using the SetLoopInversion command. However it is better to solve the problem through a wiring change. If the system setup is somehow lost the system will default to no loop inversion. It is better for the hardware system to instrinsicaly be stable rather than having to “help” the system be stable with a software configuration.

Tuning the System

In order for your servo system to operate at its full potential the servo system needs to be “tuned”. This adjustment process is performed with the SAW Application TUNEAXIS.SAW which implements the classic step response test. The Tune Axis icon is a tuning fork and looks like this:

Double click on the Tune Axis icon to start this utility. You should see a screen similar to the following:



The motor is instructed to “step” back and forth a small displacement “instantly” in the TUNEAXIS application. How well the motor is able to perform this sudden step is a measure of its performance. No physical system can exactly track a step change in displacement. Typical responses show the motor moving towards the desired step position, “overshooting” the goal and returning to the desired point.

Starting the Test

Select the axis you would like to tune by pushing the "Select Axis" button. Start the stepping motion by pushing the "Start Step" button. then push the "Trigger" button to trigger the storage scope and collect information

Interpretation

The red line indicates the commanded or theoretical step being requested. The black line indicates the actual response which is being performed by the motor. The green line indicates the torque being requested of the motor during the step. The period of the step, displacement, and compensation parameters can be changed by altering the numbers in the appropriate box and pushing the “set” button.

Avoid Saturation

When using the step response test it is important to not “saturate” any of the system elements. Digital control, as well as linear control, requires “room to move the elbows”. If the calculations or implementation of the control system encounters boundaries the effectiveness of the system will be at least reduced and perhaps, in extreme situations, unstable. The green

line indicating commanded torque is an example of this. The red lines on the commanded torque plot indicate the limits of the torque's expression. If the green line "pegs" into the red lines for extended periods then the system is saturating. The controller would really like to ask for more torque however it can't request more torque with the implementation it has. This type of saturation confuses the controller which presumes that the system is not encountering such boundaries.

Tuning Guidelines

The ability of the motor to track the step can be improved by adjusting the compensation, that is, the gain, zero, and integrator terms of the control law. Initially it is best to leave the integrator at 0 and study the behavior of the system in light of just the zero and gain. These two parameters are the main influences on the system's dynamic response. Generally a zero value of 232 or so is useful. Increase the gain value, pushing the "set" button after each new value, and observe the effects on the step response plot. The motor will most likely sound "snappier" as it more closely follows the step with higher gains. However there will come a point where the gain becomes excessive and the motor may audible or visually begin to vibrate. This can usually be seen on the plots as an oscillation well above the step frequency. At this point you can increase the zero (which limits at 255) or more likely reduce the gain to move below this vibrating instability.

Achieving 0 Steady State Error

At this point you can include the integrator to reduce steady state error. Start with a value of 1 and increase by 1. The initial effect will appear small, however the steady state value will more accurately converge on the desired value. As the integrator value is increased the motor will more quickly converge to the desired value. However, as the integrator gets too large in value, another instability is likely to occur causing the system to again oscillate. In this case "back off" the integrator to a lower value to restore the benefits of zero steady state error with stable behavior. In general the integrator value should not be greater than 1/4 of the gain value.

Record the values of gain, zero, and integrator for each axis. These values characterize your system and will be helpful when constructing applications to return the system to this current compensation.

At this point you should be ready for some preliminary servo controlled movement.

Stepper Motor Setup

Stepper motor drives are connected to signals on the axis-group connectors. Stepper motors and servo motors user the same amplifier enable signals. Stepper motors user the step and direction signals. Often stepper motors use opto-isolators on their inputs. The step, direction, and enable signals are all open collector signals.

Subsequent sections will describe preliminary motor movement. If the motor turns the wrong way you can SetCoordinateInversion command to cause movement in the opposite direction.

Chapter 3

3) Introduction to Motion

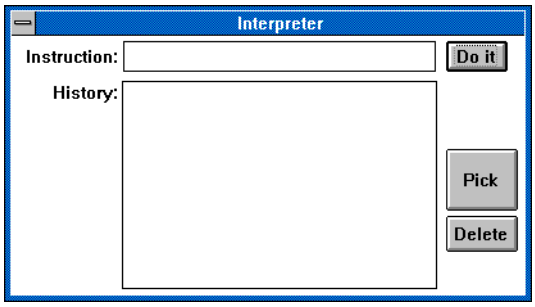
Purpose

This section will review the different motion capabilities of Motion Server. In the course of describing motion capabilities some groundwork for the language system will be made in preparation for describing motion control applications.

This document describes motion related commands from an overview vantage point. Details of commands outlined here are found in the on-line help which can be invoked from the Help menu item under SAW or from the Help icon in the Servo group directly.

Preliminary Motion

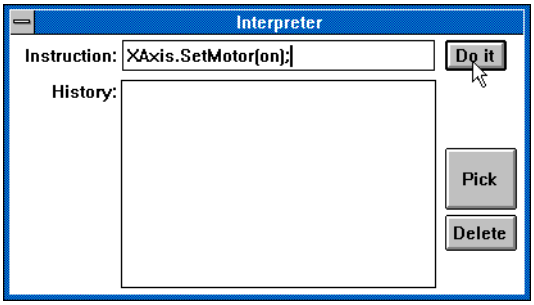
From the SAW menu of select Run\Interpreter. A dialog should appear. This interpreter can be used to submit individual commands to the control system. The command interpreter looks like the following picture:



Click inside the command line and type the command:

```
XAxis.SetMotor(On);
```

After typing in the command click the “do it” button. The dialog box should appear as shown below:



If you configured the XAxis to be a servo motor, the motor should now be displaying stable servo behavior by resisting any applied torques. If the motor is a stepper motor, the amplifier should be enabled and the stepper motor holding position.

Now type the following command:

```
XAxis.MoveBy(20000);
```

After typing in the command click the “do it” button. The X axis motor should move to a point 20000 counts away, about 10 turns for a typical 500 cycle/rev encoder. To bring the motor back do the command:

```
XAxis.MoveBy(-20000)
```

Command Structure

Motion Server commands follow an “object oriented” structure which has become popular with languages such as Object Pascal, C++, and Visual Basic. The basic structure of a command is:

```
<receiver>.<method>(parameters);
```

The <receiver> part of the command is filled in with the name of an “object” in the language system. In the example the object name was XAxis

and referred to the first motor controlled by the system. Objects include items such as a particular axis, a group of axis, a control plate, or a control element such as an editor. In general objects contain some sort of state information as well as operators that can change the state. These are all bundled together into a single unit. Motion Server permits developers to create their own objects. Details about this can be found in the section on User Defined Types.

The <method> part of the command is filled in with the name of one of the operations or "methods of doing something" that the object has been "trained" in performing. In the example the method was MoveBy. The method is separated from the receiver by a period. The period may remind you of the separator between a Pascal record variable (or struct in C) and a field of that record. The parallel is quite direct. Objects are supersets of records allowing operations as well as static fields to be related to that object. MoveBy has the behavior of moving the motor by a certain displacement. The displacement is indicated as a parameter to the MoveBy method. In this example the parameter was 20000.

Different methods take various numbers of parameters. For example the following command would move both the X and Y axis in a coordinated straight line move:

```
XYAxis.SetMotor(On);
XYAxis.MoveBy(20000,30000);
```

Here the receiver is the XYAxis, an "axis group" of type T2Axis which represents the coordinated X and Y axis together. MoveBy is the method however it has a different number of parameters than the MoveBy for a single axis. For a two dimensional axis group MoveBy requires two parameters. For a 3 dimensional axis group you would provide 3 parameters etc. The same method name can be used by different objects, each providing the appropriate response to that behavior and taking as parameters the appropriate amount of information. For example several different objects respond to the method "Clear". A text object responds by erasing the text in its window. A T2Axis responds by erasing any curve information which may have been recorded.

Single Axis Modes of Motion

Single axis are the most versatile movers in Motion Server. A single axis can respond to the following trapezoidal profiing methods:

`MoveTo`.....move to absolute coordinate
`MoveBy`.....moves relative to current commanded position
`BeginMoveTo`..moves to absolute coord but doesn't wait
`BeginMoveBy`..moves to relative coord but doesn't wait
`Jog`.....moves at a constant speed indefinitely
`SetAccel`.....set the acceleration rate
`SetDecel`.....sets the deceleration rate
`SetSpeed`.....sets the slew speed during moves
`Stop`.....causes the axis to decelerate to 0 speed and stop - execution waits for stop to complete
`BeginStop`.....causes the axis to start to decelerate to 0 speed and stop - execution immediate continues
`Abort`.....abandons motion and stops immediately

Details of these methods can be found in the on-line help by entering the help system, selecting search, and typing in the name. Let's review each motion mode one at a time.

MoveTo performs a move to an absolute coordinate. Do the command:

```
XAxis.MoveTo(0);
```

The X axis motor should move to the position 0. From the interpreter you can check the result by doing the command:

```
Prompter.Writeln(XAxis.CommandedPosition);
```

The following message box should have appeared on your screen:

The prompter is another object which knows how to display information in a modal manner (i.e. press "ok" before anything else happens). "Writeln" is a method understood by many objects to display information as readable ASCII characters. CommandedPosition is an object function which returns the position the receiver axis is currently commanded to position to. The

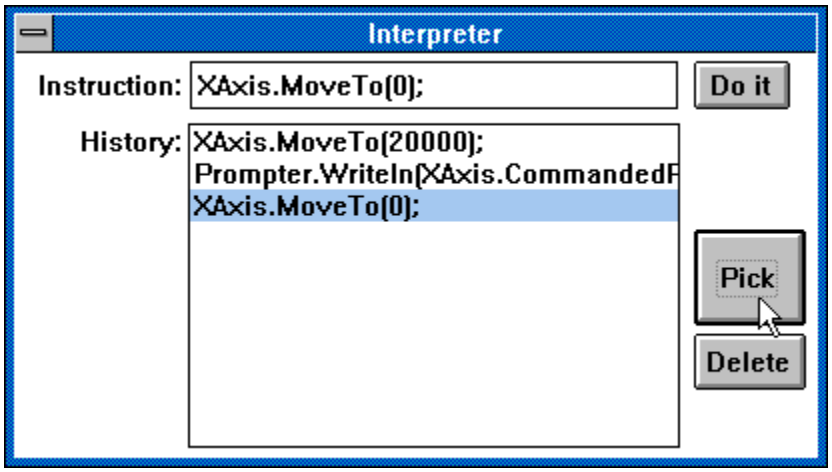
display should have indicated a value of 0 for the XAxis.CommandedPosition. You may also see the actual position of the motor by typing the command:

```
Prompter.Writeln(XAxis.ActualPosition);
```

This should be a value close to 0 but may not quite be 0 due to stiction, compensation, and other effects that determine the tracking quality of a servo. In general "To" is indicative of absolute motion, and "By" is indicative of relative motion. Now do the command:

```
XAxis.MoveTo(20000);
```

The motor should move about 10 turns. Check the position with the prompter again to see that it moved to the position 20000. Now move the motor back to 0 by "picking" the desired command from the history list and then clicking the "do it" button.



MoveBy, as has been seen, performs relative motion. You can perform the same set of exercises with MoveBy. With a starting point of 0 the results should be the same.

SetAccel and SetDecel are used to change how rapidly the motor comes up to speed and how quickly it slows to a stop, respectively. These methods take parameters in units of counts per second squared. Generally values of 100,000 are gentle, values of 1,000,000 sharper. Do the command:

```
XAxis.SetAccel(1000000);
```

Repeat the MoveBy test. You should notice a quick acceleration with the same previous deceleration. Stepper motors can "slip" or "lose steps" if they are asked to produce excessive accelerations. You will need to understand the limits of your own system based on manufacturer information about your drive electronics and motors as well as the demands placed on the system by the loads you are moving.

Set the acceleration back by doing the command:

```
XAxis.SetAccel(100000);
```

An important issue in motion control applications has to do with “synchronization”. Motion systems are often positioning “effectors” of some kind, such as a drill head. Typically you want to move to some position, drill a hole, move to a new position, drill another hole etc. You certainly do not want to drill the hole before you get to the destination and you do not want to move while you are drilling. It is important for the activities of drilling and the activities of moving to be synchronized in such a way as to properly wait for each other. MoveTo and MoveBy are methods which wait for the move to finish before going to the next command. The program is suspended until the motion is complete. For an application like the drill application waiting is extremely important. In other applications, however, just the opposite may be important. In order to save time it may be necessary for other activities to be preparing while the drill is moving to a new location. For cases where it is important to continue doing activities while the motors are in motion there are commands which start with the word “Begin”, i.e. BeginMoveTo, BeginMoveBy, BeginStop.

One obvious command that is relevant to an axis in motion is the command Stop. Begin another move using BeginMoveBy and then do the command:

```
XAxis.Stop;
```

The motor should stop. Stop slows the motor down at the specified deceleration rate. Abort halts the motor dead in its tracks. Again, begin a move and do the command:

```
XAxis.Abort;
```

The motor should suddenly stop. The difference between Stop and Abort is the deceleration of the motor. Abort should generally be used in emergency situations where movement may be dangerous. The jolt to the machine is usually not good for the mechanics, particularly when aborting from a high speed. As well, aborting from a high speed may result in such a large following error that the servos may go beyond the specified ErrorLimit and shutdown. If you plan on aborting from high speeds make sure the ErrorLimit is large enough to prevent the servos from shutting themselves down.

There are some situations where you need to stop motion but continue to do operations while the motor is slowing down to complete the stop. This can be achieved with the method BeginStop which immediately returns after telling the motor to start decelerating.

Jog is used to run a motor at a specified speed indefinitely. (Only do the following test if your mechanism has no movement limitations). Do the command:

```
XAxis.Jog(1000);
```

The motor should begin jogging at a speed of 1000 counts/second. Now do the command:

```
XAxis.Jog(2000);
```

The motor should speed up by a factor of 2. While in the middle of a move (in this case an indefinite move) it is possible to request a new move.

Dynamic Profiling

One of the strengths of Motion Servers' motion system is the ability to change a move while in the midst of the move. It is possible to change the acceleration, deceleration, speed, and even destination on the fly (for a single axis) during any part of the accel, slew, decel velocity profile. It's possible to change a jog to a move or a move to a jog while in the midst of doing either one. Let's study some examples of this.

Set the X axis speed to 1000 with the command:

```
XAxis.SetSpeed(1000);
```

Now do the command:

```
XAxis.BeginMoveBy(20000);
```

Immediately follow the command with:

```
XAxis.SetSpeed(20000);
```

The motor should speed up to a much higher speed and promptly finish the move which was started at a slower speed. Now set the decel to be a very small value, i.e. do the command:

```
XAxis.SetDecel(10000);
```

Start another move with the command:

```
XAxis.BeginMoveBy(20000);
```

The motor should quickly speed up and then start almost immediately slowing down in a gradual manner. Because the decel is so slow it has to start slowing down early in order to come to a stop by the time it gets to the destination. Now do the command:

```
XAxis.SetDecel(100000);
```

The motor should accelerate up to its slew speed and then decelerate at the higher decel rate to complete the move. Reset the motor position by using the command:

```
XAxis.SetActualPosition(0);
```

It's possible for a single axis move to have the destination changed on the fly. Begin a move with the command:

```
XAxis.BeginMoveBy(20000);
```

Before the command has a chance to finish do the command again, i.e. give another

```
XAxis.BeginMoveBy(20000);
```

Wait for the move to finish and inspect the actual position with the command:

```
Prompter.Writeln(XAxis.ActualPosition);
```

You see that the position is farther than 20000. The first move was superseded by the second move which "spliced" a move by 20000 counts onto the current position of the motor, somewhere on its way to the original target of 20000.

You may be thinking that there are some problematic situations with dynamic profiling. It seems possible to ask for unreasonable things. This is indeed the case. For example perform the previous test, however instead of having the second move be a move by 20000 use a parameter of 0. What we are saying is "while in motion, move to where you currently are", that is to say stop instantaneously.

Start the motor with a first move command of:

```
XAxis.BeginMoveBy(20000);
```

Immediately follow the command with:

```
XAxis.MoveBy(0);
```

The result should be a prompter box indicating that an “escape” has occurred and the motor slowing at the specified decel rate. The escape code indicates a “Motion Overrun”. This is the case. It is impossible to satisfy the command to stop immediately while maintaining the specified decel rate. The behavior of the system when given impossible commands is to gracefully stop after “raising an exception”. Douloi Pascal supports structured exception handling, a powerful tool for responding to these sorts of problems. Additional details of the use of exception handling will be discussed in the language section. Using the language system it is possible to “trap” this exception and respond in an appropriate manner so as to achieve the desired application behavior.

Multiple Axis Modes of Motion

Motion Server supports vector coordination of groups of axis for group sizes from 2 through 6. Groups are coordinated so as to all start at the same time, stop at the same time, and transition from accel to slew and slew to decel at the same time. This results in “straight line joint space” vector motion in whatever dimension of space the machine occupies. Note that if the machine has rotary joints that straight lines in “joint space” do not create straight lines in Cartesian space. Accomplishing straight cartessian space lines on a mechanism with rotating joint, such as a cylindrical or scara configuration robot, requires kinematic equations. Motion Server is very well suited for performing high-performance kinematics. Consult Douloi Automation for additional information.

The methods for multiaxis machines include the commands for single axis machines with additional parameters as needed for the additional axis included. The following methods are provided for multiaxis groups:

`CommandedPosition...` returns distance along vector path
`MoveTo.............` move to absolute coordinate
`MoveBy.............` moves relative to current commanded position
`MoveToVector.......` moves to absolute vector coordinate
`MoveByVector.......` moves by relative vector coordinate
`BeginMoveTo........` moves to absolute coord but doesn’t wait
`BeginMoveBy........` moves to relative coord but doesn’t wait
`BeginMoveToVector...` moves to absolute vector coordinate no wait
`BeginMoveByVector...` moves by relative vector with no wait
`SetAccel...........` set the acceleration rate
`SetDecel...........` sets the deceleration rate
`SetSpeed...........` sets the slew speed during moves

`Stop`.....causes the axis to decelerate to 0 and stop, execution wait for stop to finish
`BeginStop`.....causes the axis to decelerate to 0 and stop, execution immediately continues
`Abort`.....abandons motion and stops immediately
`LinkTo`.....associates a data storage area to the for holding curve info
`AppendMoveTo`.....adds an absolute component to a curved path
`AppendMoveBy`.....adds a relative component to a curved path
`AppendArc`.....add an arc segment to a curved path
`MoveAlongCurve`.....begins motion along continuous curved path

When referring to acceleration and speeds for a multiaxis group the accelerations and speeds refer to the vector path of motion, not of any particular axis. Note that Motion Server is not aware of any physical gear reductions in the mechanism that makes one axis have a different counts/inch ratio than another axis. This may result in physical velocity discrepancies between two directions of motion although the path will be correct. In general it is helpful to the motion controller for the physical scale of the machine, i.e. counts/inch, to be the same in all directions for uniform behavior. If this is not possible it to accomplish there is a technique which can be employed to make the mechanism appear to be so. For additional information contact Douloi Automation.

Multiaxis groups do not permit changing the destination of an in-progress move on-the-fly however accel, decel and speed may be changed on-the-fly.

It is also possible to specify continuous motion along a curve. This capability will be discussed in a later section.

Chapter 4

4) Servo Application Workbench Tutorial

Introduction to SAW

The interpreter used in the previous section has provided an opportunity to perform motion using individual commands. Servo Application Workbench provides a way to package commands and direct application behavior. This section describes how to learn to use SAW.

Developing an application with Servo Application Workbench is similar to using a drawing program and similar to procedural programming. The visual arrangement and constitution of a program is described with drawing tools to include objects such as buttons, text items, editors, and “subassemblies” which themselves can contain such parts. Behaviors and relationships between these objects are then created using programming techniques.

SAW uses an “event” model of operation. Different activities, such as clicking the mouse button, create “events” that the program responds to. Different objects may be programmed to respond to different events to create program behavior.

The most straightforward way to understand SAW is to use it. The tutorial following in the next section will lead you through the use of different SAW capabilities in lessons with occasional “projects” that combine principles from the lessons in a practical result.

Douloi's version of Object Pascal is used in the tutorial. If you are familiar with most any programming language you should be able to navigate the tutorial aided by the specific commands provided. The chapter discussing the Douloi Pascal Language provides more specific language details.

Lesson 1- Running a Minimum SAW Application

Objective

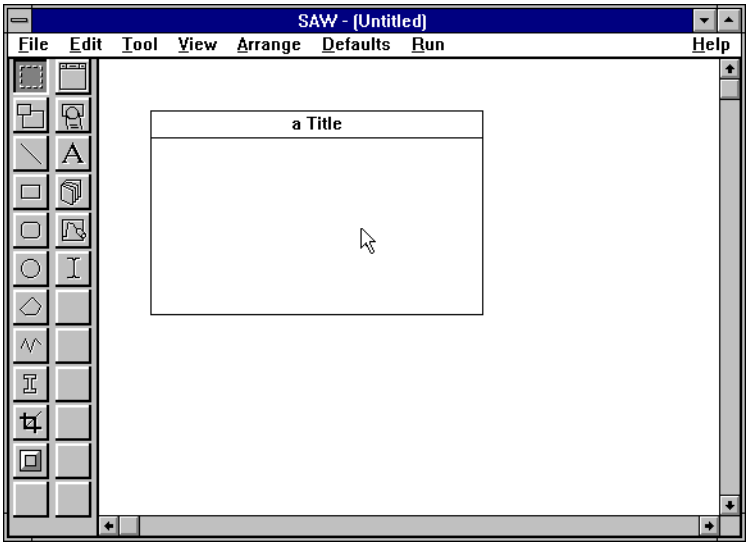
The purpose of this lesson is to understand how little is required to make an application with the Servo Application Workbench and to gain experience in starting applications.

Start SAW

Double click on the SAW icon. An "About Box" is shown introducing SAW.

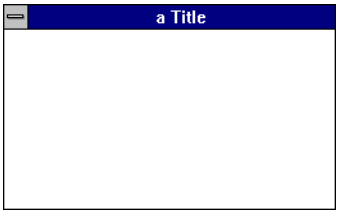
Click on the “Ok” button after the SAW title is shown.

SAW's work area should then be similar to what is shown below:



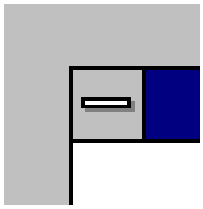
Run the Default Application

From the SAW main menu choose Run, and then choose Start App. There should now be a new window that contains as its caption “a Title" such as shown below.

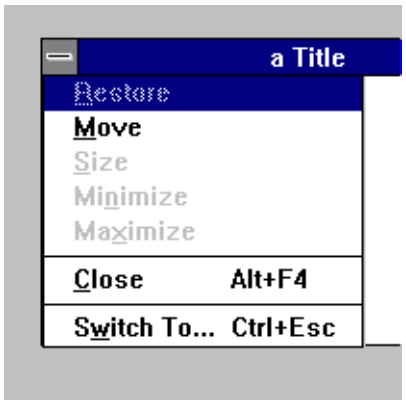


This is the “default” application provided as an application foundation. This window should be the same size and shape as the drawing first seen in the drawing area of SAW. Drag the title bar of this window and note that it behaves like other Windows applications.

Close the window clicking on the "system menu" of the window (the little upper-left corner button on many Windows applications).



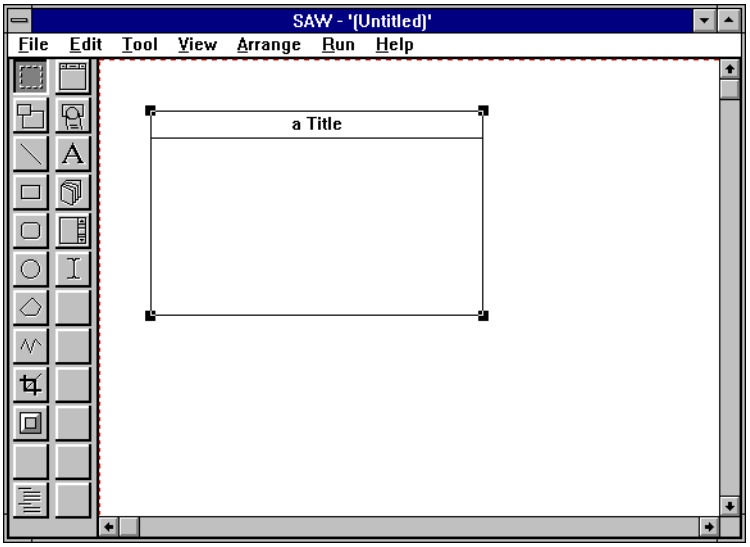
This produces the following pop-up menu:



Choose "Close" . The window should disappear. Alternatiely, you can double click on the system menu to close the application.

Modify the Default Application

Click on the picture of the Window which is in the SAW drawing area (not the resultant Window application that was just being studied). Four "handles" should appear on the outline indicating that the window is selected for alteration.



Move the cursor over one of the handles. The cursor shape should change to a four-way shape indicating its over a handle. Drag the handle to a new location to change the shape of the window.

From the main menu choose Run, then choose Start App. You should see the running application window again but with a different size reflecting the alteration you made.

Summary

SAW is able to immediately provide a default application which has fundamental behaviors such as dragging and displaying a system menu. This foundation application is what motion applications are based on.

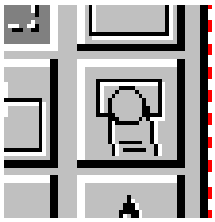
Lesson 2 - Creating a Button

Objective

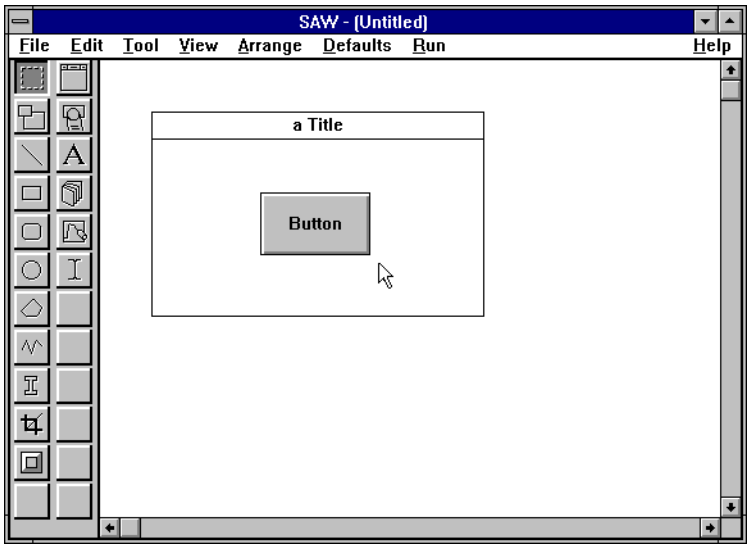
Although we've created an application it hasn't done anything yet. Buttons are the simplest way of asking for program behavior. This lesson should familiarize you with the general behaviors of buttons.

Create a Button

Select from the "tool bar" on the left of side of SAW the tool which is in the right-most column, second from the top, which has the rectangle being pushed by a finger. This is the "button" tool.



Move the cursor to the upper left interior of the default Window in the SAW drawing area and drag to the lower right area. When you release the mouse button you should see a Windows button in the drawing area of SAW, such as shown below.



From the main menu select Run/Start App. You should now see the default application running with the recently created button. Pushing the button shows the characteristic button movement but no action is performed because none has been specified. Close the window and return to SAW.

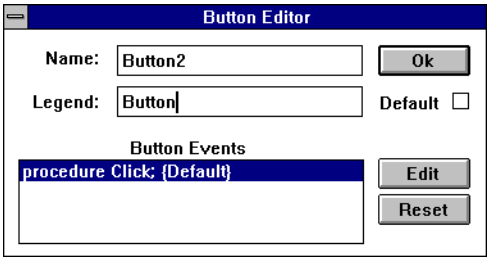
Modify a Button Appearance

In the SAW drawing area, select the button by clicking inside the button shape.

Relocate the button (within the boundaries of the default window frame) by dragging the button to a new location.

Resize the button by dragging one of the four corner handles to a new location.

Double click on the button in the SAW drawing area. A button editor should appear similar to the one shown below.



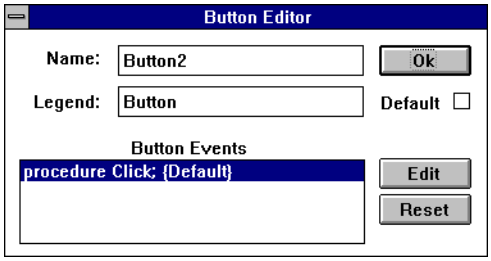
Select the “Legend” field, currently containing the word “Button” and change that word to some other word, such as “Test”. Close the button editor by clicking "Ok".

From the main menu choose Run, then choose StartApp. Note that the legend on the button has changed.

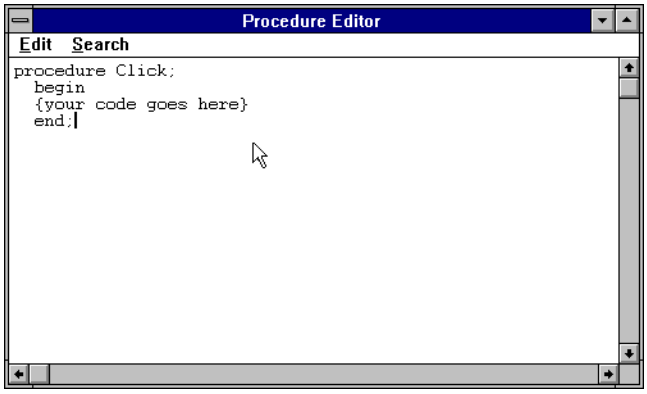
Close the window.

Modify a Button Behavior

Double click on the button in the SAW drawing area. Note that on the lower section of the button editor there is a list of Event Procedures. SAW supports the “event” user interface model used by Windows. Different objects can experience different “events” and produce a related behavior. For buttons the most frequently experienced event is a mouse click. Buttons are provided with a procedure named “Click” which defines what should happen if the user clicks the mouse on the button.



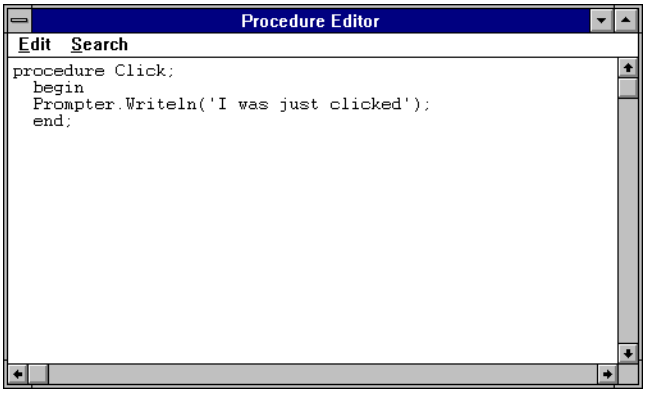
Select the click procedure in the list box and then select the Edit button adjacent to the list box. An editor should appear similar to the one shown below.



Position the cursor on the line which says {Your code goes here} and replace that line with the line:

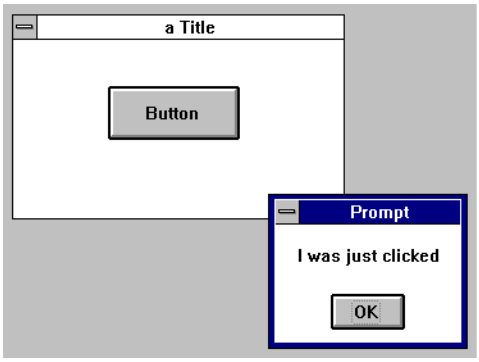
```
Prompter.writeln('I was just clicked');
```

Note that the word after Prompter and the period is "write line", with "line" abbreviated to "LN", (not 1N). Also note that the quotes around "I was just clicked" should be single quotes, (i.e. un-shift of double quote, not un-shift of tilda), and that both quotes are the same (despite whatever “help” the manual publishing system has provided by making one appear to be a back-quote).



Close the editor by double click on the upper left icon. Close the button editor by selecting “Ok”.

Start the application. You should see a window with a button in it as before. Now click on the button. You should see a prompter window appear with the text “I was just clicked”.



Click on the “OK” button below “I was just clicked” to acknowledge the prompter.

Summary

Buttons are created by selecting the Button Tool, and dragging a button shape in the SAW drawing area on the application window. By double clicking on the button picture a button editor is provided to alter button attributes such as the legend and the behavior of the button when clicked.

Lesson 3 - Creating Text

Objective

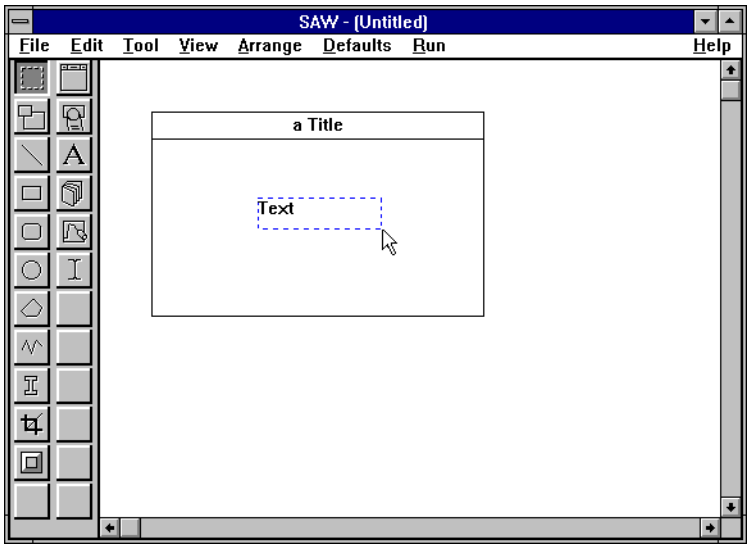
This lesson illustrates the use of text in an application both passively for annotation and actively as a mechanism for displaying information.

Create a Text Item

Select from the “tool bar” on the left of side of SAW the tool which is in the right-most column, third from the top, which has the capital A. This is the “text” tool.

Move the cursor to the upper left interior of the default Window in the SAW drawing area and drag to the lower right area. When you release the mouse button you should see a dotted rectangle that contains the word “Text” as illustrated below.

From the main menu select Run\Start App. You should now see the default application running with the word “Text” on it. Close the window and return to SAW.



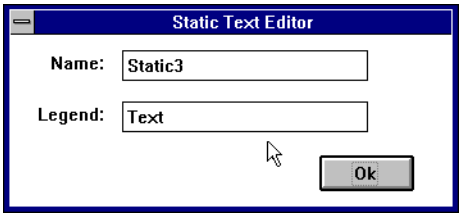
Change Text Interactively

In the SAW drawing area, select the text item by clicking inside the dotted rectangle shape.

Relocate the text (within the boundaries of the default window frame) by dragging the text item to a new location.

Resize the text window by dragging one of the four corner handles to a new location. The size of the text is currently fixed. The length of the box determines how long a section of text can be accommodated.

Double click on the text box in the SAW drawing area. A text editor should appear similar to what is shown below.

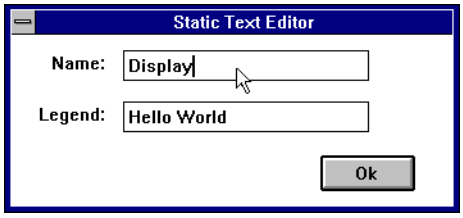


Select the lower field, currently containing the word “Text” and change that word to some other word, such as “Hello World”. Select Run\Start App. The appearance should have changed to what is shown below.

Close the application and go back to SAW.

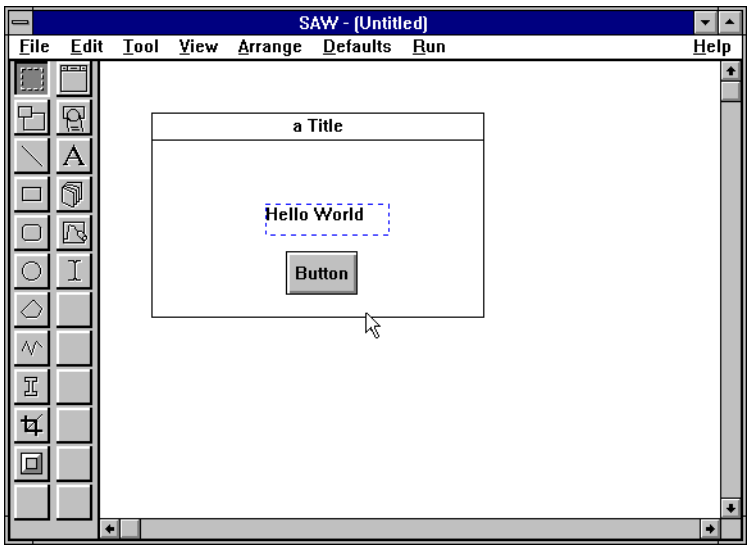
Change Text with a Program

Double click on the text box again. The upper field is called “Name”. Objects being placed into the application, such as buttons and text items, have names which are used to refer to these objects. Edit the name of the text item to be “Display” such as shown below.

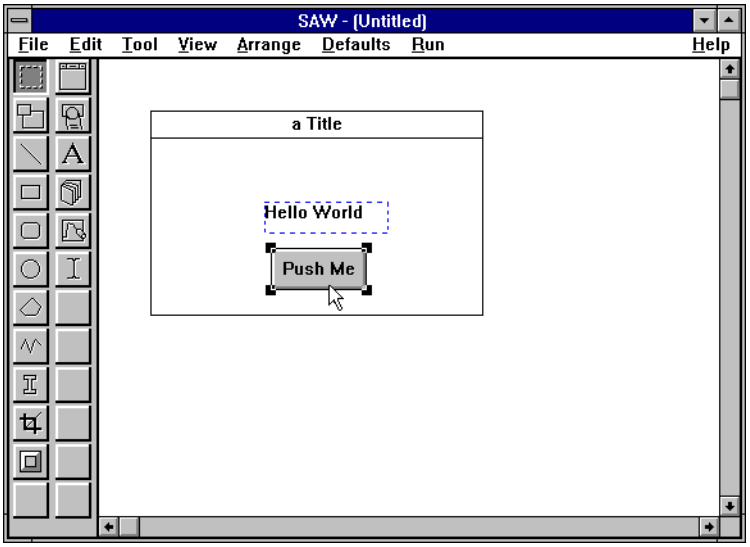


Close the Text editor.

Create a new button and place it below the text item as shown in below.



Double click on the button and change the button legend to “Push Me”.

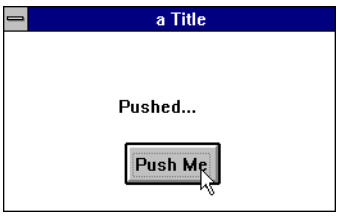


Select the Click procedure in the event list and select “Edit”. Modify the click procedure to include:

```
procedure click;
      begin
      Display.Clear;
      Display.Writeln('Pushed....');
      end;
```

Run the application.

Push the “Push Me” button. You should see new text in the Display text item.



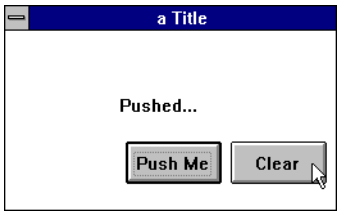
Close the application and return to the SAW drawing area.

Select the “Push Me” button. While holding down the control key on the keyboard drag the “Push Me” button to a new location. Note that you have created a copy of the button by holding down the control key. Copies contain the same shape, attributes, and procedures as the original however the name has been changed. Edit the legend of this button copy to be

"Clear" and edit the click procedure to read:

```
procedure click;
      begin
      Display.Clear;
      end;
```

Run the application again. Now the "Push" button fills the text item and the "Clear" button clears it.



Questions and Answers

Question: Does the button name matter?

Usually button names are not used although they do need to be unique. The names provided for you by SAW should be unique names. It is possible to refer to the click procedure of a button, however, by referring to the name of the button followed by click. For example if the clear button in the above example was named "ClearButton" we could have invoked the click procedure for that button in the "Push Me" button by referring to ClearButton.Click.

Summary

Text items can be created during development to annotate an application. Text items can also serve as display mechanisms to show values being produced by application programs. In a manner similar to buttons, spatial attributes of text can be set by dragging the Text item in the drawing area. Other attributes an be changed by double clicking on the text time and editing the attribute.

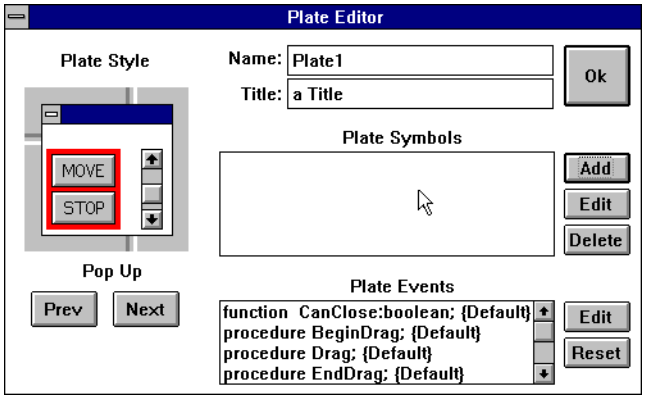
Lesson 4 - Using Plates

Objective

Several different metaphors are used by the Servo Application Workbench. This lesson introduces the plate metaphor.

Change a Plate Title

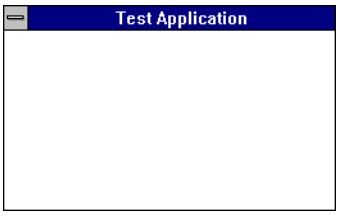
Double click on the default window. An editor should appear similar to figure below.



This is called a Plate Editor. The default window is called, in SAW terms, a plate rather than just a window. Plates appear as windows but contain additional information, much of which is available to alter with this plate editor. The idea behind the word “plate” is the action of attaching items to plates. Already text items and buttons have been attached to this plate. As well it is possible to attach non-spatial items to a plate including data structures such as variables and arrays, and behaviors such as event procedures and user procedures and functions. These different things become part of the plate and travel along with it.

Click on the “Title” field and change that field to the title “Test Application”

Close the Plate Editor by clicking "Ok" and run the application (Run/Start App). Note that the caption is now "Test Application" instead of "a Title".

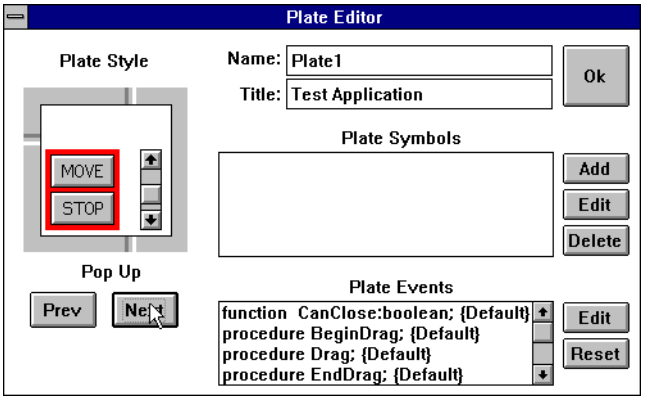


Close the application by double clicking on the upper left system menu icon on the Test Application window.

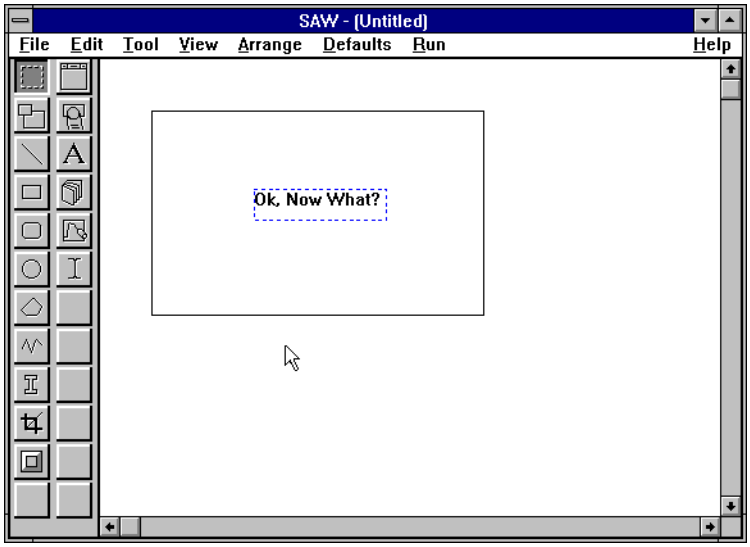
Change a Plate Appearance

Double click on the default window to invoke the Plate Editor again.

Plates can display with different Window frame styles. On the left side of the plate editor is a small picture of the current style. By clicking on “Next” or “Previous” alternative styles can be chosen. Advance through the selection of styles until a thin black line border without a caption is chosen. Close the plate editor by clicking "Ok".



Add a text item to the center of the plate which reads “OK. Now What?”.

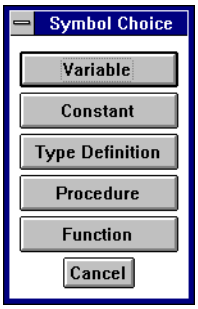


Run the application. The plate appears, without the caption bar and most noticeably without the system control icon which allows you to close the application!

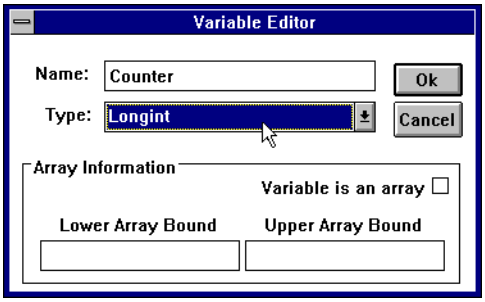
From the SAW main menu select Run\Stop App to close the application even when the application itself is lacking a system menu.

Create a Plate Variable

Double click on the default plate to invoke the plate editor. In the middle of the plate editor is a list for Plate Symbols. These are different procedures, variables, and constants that may be attached to the plate. Select Add. A dialog box appears showing the different sorts of things that can be attached to a plate.



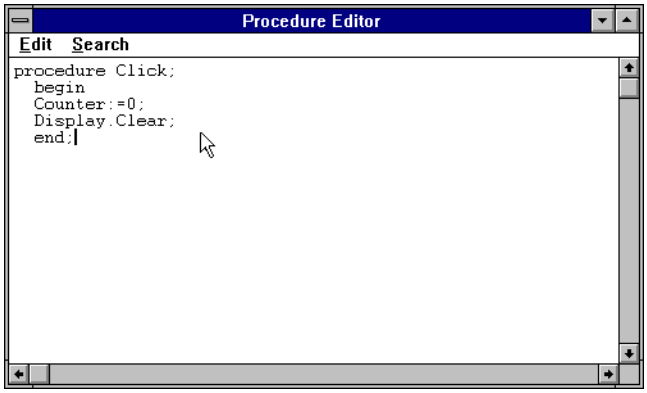
Select the default choice, "Variable". A variable editor appears that allow you to specify a name for the variable, a type, and array information if the variable is to represent an array. Type into the Name field the word “Counter” and select from the Type combination box the type “Longint”. Do not check “Is An Array”.



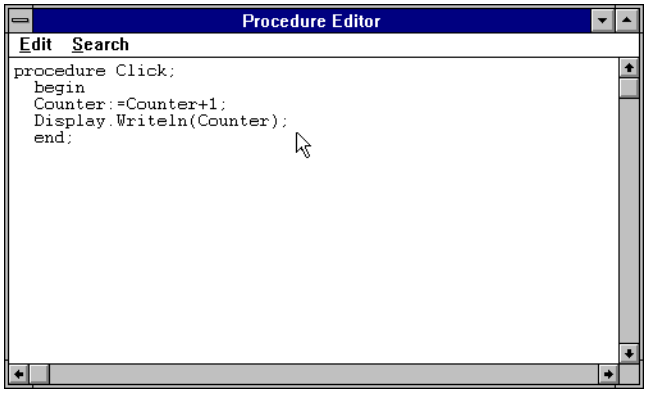
When these items have been indicated select “OK” to leave the variable editor, and “Ok” to leave the plate editor so that you are back to the SAW drawing area.

Change a Plate Variable

Change the name of the Text item in the application to the name "Display". Add to the plate a button with the legend “Reset”. Make the click procedure for reset contain:

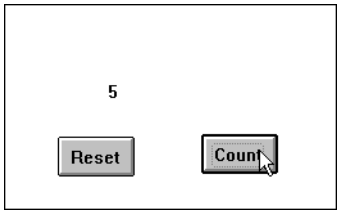


Add to the plate a button with the legend “Count”. Make the click procedure for count contain:



Run the application. Push Reset. The Display should clear. Push Count. The Display should show 1. Push Count again and the display should show 2.

The figure below shows the application after several button presses.

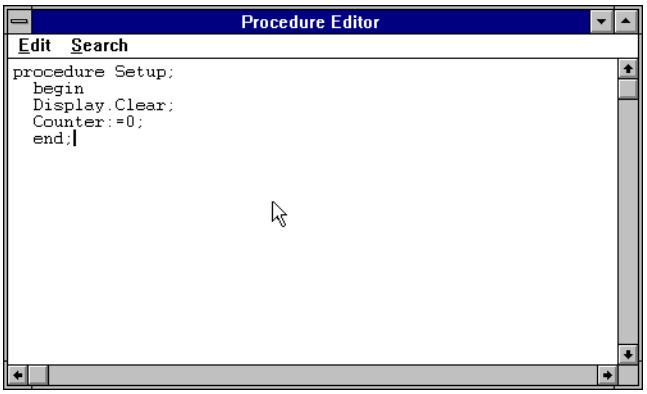


Close the application to go back to SAW.

Change a Plate Event Procedure

Like buttons, plates have different events that they respond to through event procedures. Double click on the plate to invoke the plate editor. The event procedure list for plates, near the bottom, includes names such as BeginDrag, Drag, EndDrag, and Setup. BeginDrag occurs when the mouse button is pushed down over a plate. Drag occurs when the mouse moves with the button down over a plate, and EndDrag occurs when the mouse buttons is released. Examples of how to “drag and drop”, including dragging a 2 axis mechanism with the mouse, are discussed in the Drag and Drop Application Sketch.

Select the Setup event procedure and choose Edit. Setup is an event procedure which invokes when the window first appears. Edit the setup procedure to be the reset button behavior:



Close this window by double clicking on the window's sytem control icon, and then close the plate editor by clicking "Ok".

Select the Reset Button and choose Edit/Cut. Cutting the selected object removes it from the plate. We don’t need the reset button now to initialize the plate because the initialization behavior has been put into setup.

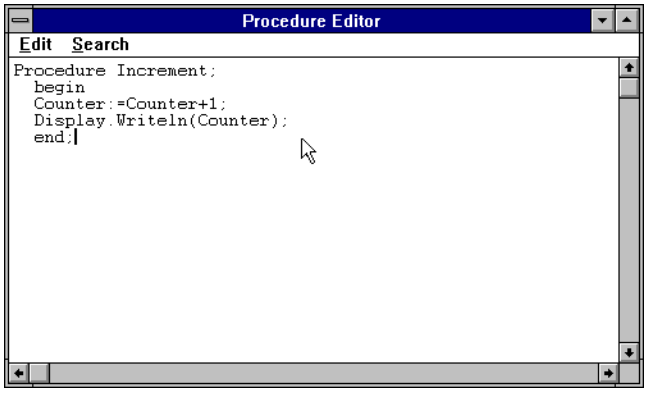
Now run the application. The initial appearance of Display should be blank. Pushing “Count” should display 1 and ascend with subsequent clicks.

Close the application.

Create a Plate User Procedure

Plate procedures provide subroutine capability to SAW. Double click on the default plate to invoke the Plate Editor. Click on "Add" near the Plate Symbol list. Choose "Procedure" from the list of different user symbols. An editor should appear with a procedure template.

Edit the procedure to look like the following:



Close the editor by choosing close from its system menu or by double clicking the system menu.

Edit the click procedure for the "Count" button to read:

```
procedure click;
  begin
  Increment;
  end;
```

Close this editor and close the Plate Editor by clicking "Ok". Run the application. The behavior should be the same.

Summary

Plates provide a framework for declaring application related procedures and variables. Plates respond to events and can have event response procedures which invoke when events occur.

Lesson 5 - Using Bump Graphics

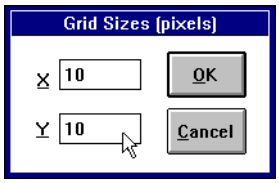
Objective

Bump graphics provide applications with a more physical appearance that helps the user feel more comfortable with “instrument” type interfaces. Bumps are a convenient means of creating these appearances. This lesson introduces you to the use of bumps.

Turn on the Grid

The grid in SAW causes the cursor to "snap" to a discrete array of points helping retain alignment of buttons and graphics that are placed on the screen. It is most convenient to develop with the grid on. From the SAW main menu select View/Grids... to see the following dialog:

A grid size of 10 by 10 is convenient for many types of applications. After typing in 10 for X and 10 for Y select OK. Select the default plate and drag the upper left corner as well as the lower



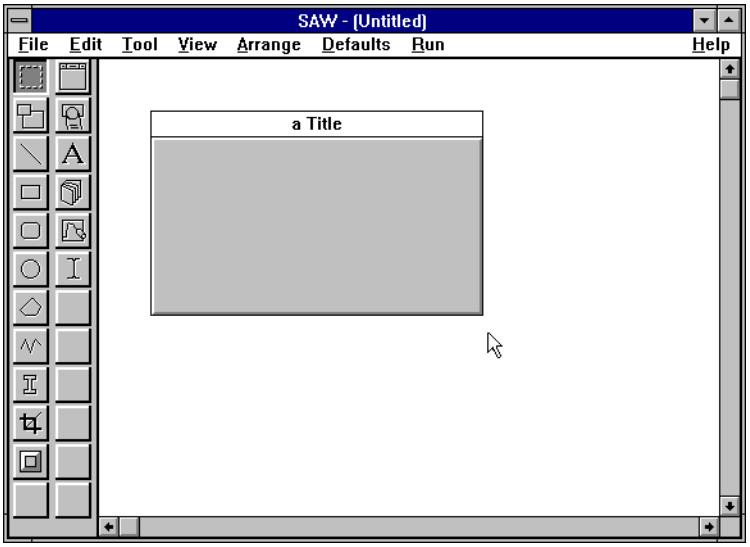
right corner so as to "snap" these points onto the grid.

Create a Bump

The left column of tools is primarily used to provide cosmetic graphics to an application. Select the “Bump Tool”, which is depicted as a chamfered surface near the bottom of the left column of tools.

Drag a rectangle over the surface of the default plate. When you release the mouse a “bump” has appeared on the screen in default colors and depth. Place the upper left point on the left margin, under the caption bar, and the lower right point at the lower right point of the default plate.

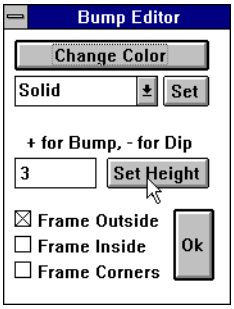
Select the bump by clicking inside its area. Drag one of the bump handles to a new location to resize the bump.



Modify a Bump

Double click on the bump. A “Bump Editor” appears allowing you to alter the properties of the bump.

Indicate a different depth for the bump by changing the “depth” value to a larger number. Click on “Set” after changing the value. Click on "Ok" to close the Bump Editor.

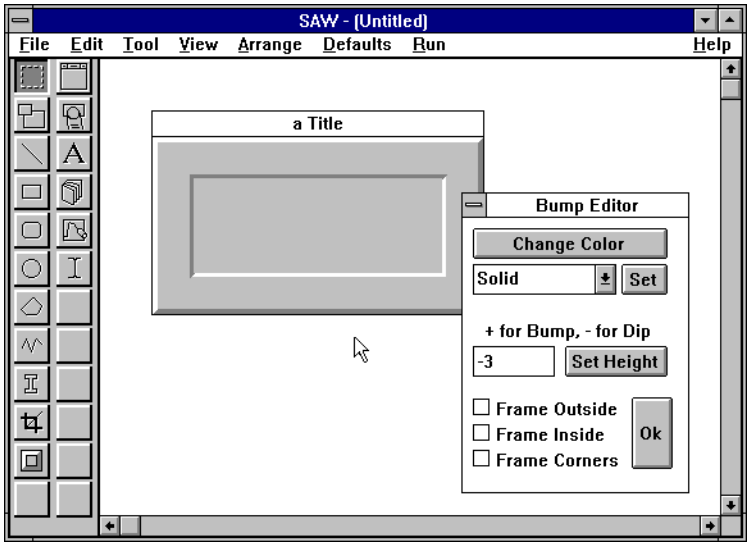


Place a Dip on Top of a Bump

Select the bump tool again and drag another bump on the surface of the first bump. Now you have a bump on a bump.

Double click on the new bump to invoke the “Bump Editor”. Make this bump have a negative depth, i.e. -3, and click the "Frame Outside" checkbox to remove the unnecessary outline. Click “Ok” to have the new depth take effect. Note that the bump now appears to go down instead of up, i.e. a dip instead of a bump.

Dips generally are nicer without being outlined. If dips or bumps are particularly large, drawing in the corners helps define the shape. Experiment with these different outline options to see the effect.



Reorder Graphics

When graphics are on top of one another the order of display is important. To study this issue, create a new application by selecting “File\New” and confirm you want to start over.

Create a button in the middle of the default dialog.

Create a bump that completely covers the button. Note that the button is now hidden.

Select the bump (if its not still selected) and select the menu option “Arrange/Send To Back”. Graphics are recorded in a list. The list is drawn from the back to the front. By sending the selected graphic to the back you are indicating that you want it drawn earlier, i.e. “in back of” the rest of the

graphics in the image. Select graphics and “Arrange/Bring To Front” and “Arrange/Send To Back” to become familiar with this ordering principle.

There can be times, when multiple graphics are on top of one another or are the same size, that selecting a particular graphic is difficult. "Sending to the back" a graphic which insists on being selected is one technique which can help you select the desired graphic. Another technique which can be used is to temporarily move out of the way a graphic which seems to insist on getting selected so as to make sufficient clearance to pick a particular graphic you would like to work with. After modifying the desired graphic you can easily move the offending graphic back into place.

Summary

By using the “Bump Tool” cosmetic graphic effects can be added to a plate to enhance the texture or “physical appearance” of the plate to produce effects such as chamfered edges and indentations. These features can help group and organize an application as well as make the application appear more comfortable and familiar to the user.

Lesson 6 - Calculator Project

Objective

Enough principles have been learned to make your first application. This lesson will lead you through the construction of a simple 4 function calculator. This calculator illustrates the relationships between the different features learned so far.

Create Calculator Faceplate

Turn on the grid with 10 pixels for each X location and 10 pixels for each Y location.

Resize the default plate to be about half the size of the SAW drawing area.

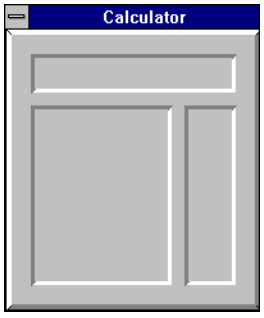
Change the name of the default plate to "Calculator".

Add a bump to the surface of the default plate and make the bump 5 pixels high. Frame the outside of the bump and frame the corners.

Add three dips to the surface of the default plate with a depth of 3, i.e. -3. Make a wide narrow dip near the top for the calculator display, a more square dip near the bottom to group the calculator number keys, and a vertical rectangular dip on the lower right to group the operation keys.

Save the project development so far by selecting “File/Save As” and type the name “CalcTest”.

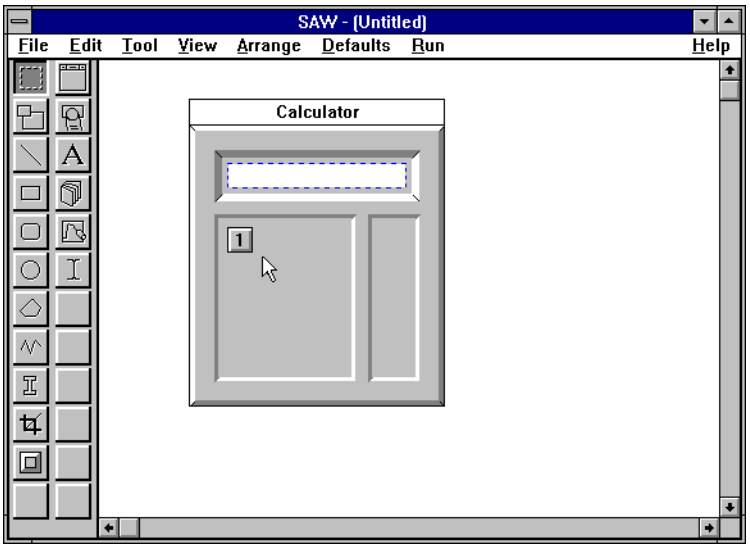
Run the application to test what has been done so far. You should have a result something similar to what is shown below.



Create Calculator Number Keys

Add a Text Item inside the dip near the top of the calculator. Rename this text item to be called “Display”. When placing the text you may discover that the "dip" is not the right size or in the right place. After positioning the text, drag corners of the dips, possibly moving other things out of the way, so as to have the dip symmetrically around the display. It may be good to make the dip deeper, as is often the case in physical calculators.

Add a small button inside the lower left dip. Place on this button the legend “1”. In general buttons will be the size of the dragged rectangle, however a button will enlarge itself so as to hold the legend. If the button "grew" it most likely is because of the size of the default legend "button". Change the legend first to "1" (a much shorter legend) and then resize the button by dragging the button's corners. Your work should appear similar to the following picture.



Edit the click procedure for this button to be:

```
procedure Click;
  begin
  PushNumber(1);
  end;
```

PushNumber is a routine (yet to be written) that handles the action of a calculator key being pressed.

Control-Drag the button to create a copy and place it adjacent to the first. Change the legend of this button to read “2”. Edit the click procedure for this button to be:

```
procedure Click;
  begin
  PushNumber(2);
  end;
```

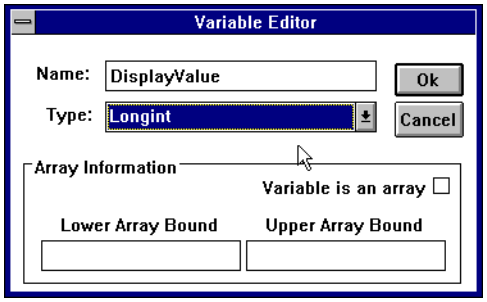
In a similar manner create the other 8 keys for the calculator keypad. Having the number "1" key in the upper left conforms to the "telephone" keypad. Normally calculators have the number "1" in the lower left. Change from the telephone arrangement to the calculator arrangement by dragging buttons to the correct locations. Note that you are allowed to drop buttons into the empty area beside the calculator and pick them up later after you've freed up the spot where they will be placed.

Save your work.

Create Procedure PushNumber

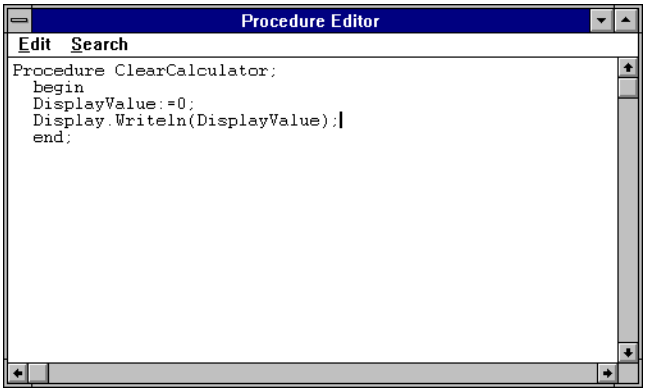
PushNumber handles the case of one of the number keys being pushed. Double click on the default plate to invoke the plate editor. A good place to click plates covered with bumps is the title bar area. Clicking on the body of the plates, generally produces the bump editor for the overlaying bump graphic representing the calculator surface, which is not the item you want at this point.

Click “Add” and choose “Variable”. Edit the name of the variable to be DisplayValue and give it a Longint type. The variable editor should appear as shown below:

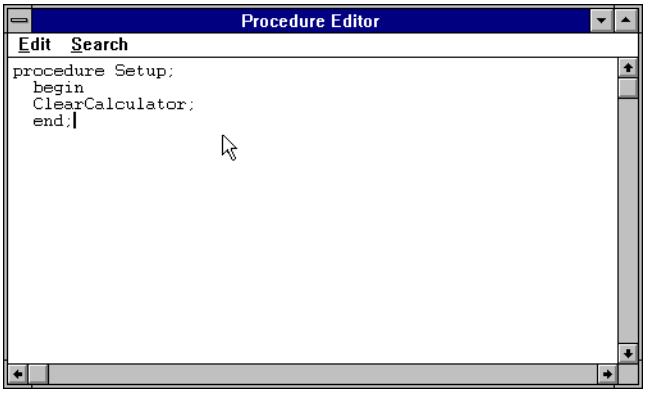


DisplayValue will remember the value of the calculator display. Click on “Ok” to confirm this new variable.

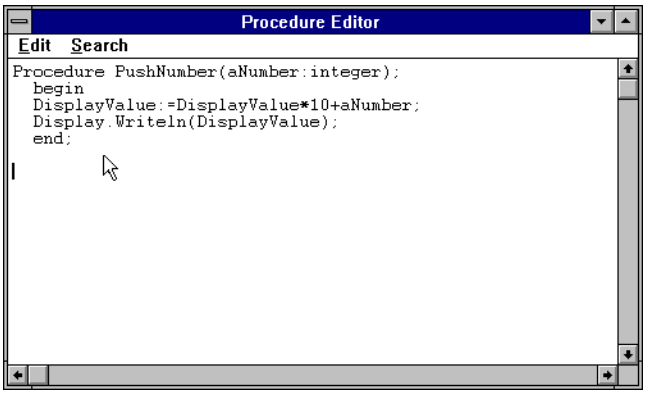
Click “Add” and choose “Procedure”. Edit this new user procedure to be:



Edit the Setup procedure for the plate to be:



Click “Add” and choose “Procedure”. Edit this new user procedure to be:

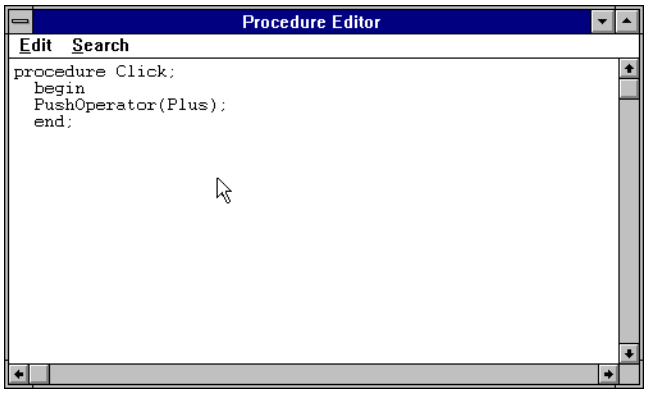


Each time a number key is pushed this will cause the calculator display to shift to the left and accumulate the contribution of that particular key value.

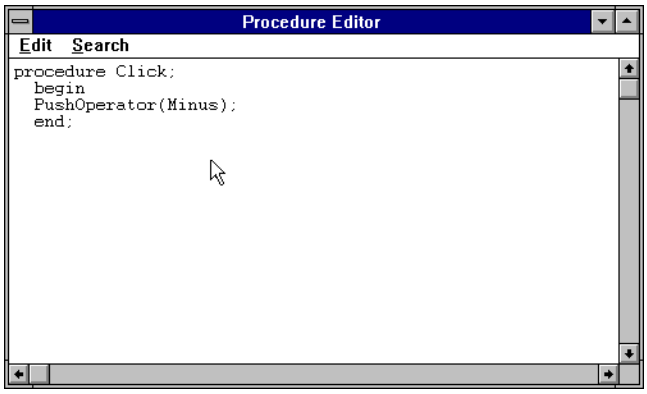
Save your work and run the application. Clicking numbers should cause the calculator display to change.

Create Calculator Operation Keys

Control-Drag one of the number keys into the vertical dip area where the calculator operations will be placed. Edit the legend of the key to read "+". Change the click procedure for this button to be:

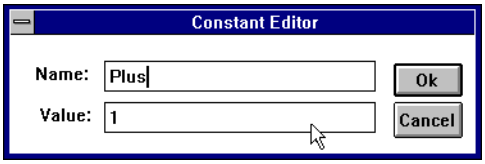


Control-Drag a copy of this button to the space below. Change the legend to the "-" sign and change the click procedure to be:



In a similar manner create multiply ("x") and divide ("/") keys with the PushOperator parameters of "multiply" and "divide".

Invoke the Plate Editor and click on “Add”. Choose “Constant” from the set of choices. Constants relate names to values which do not change. Type the name “Plus” and set the value to 1 so that the Constant Editor looks like the one shown below.



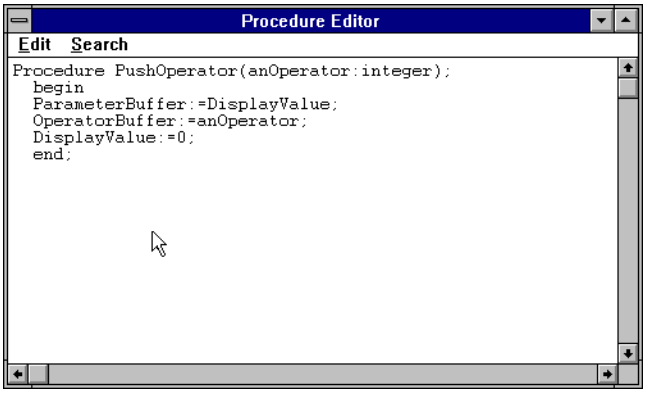
Click “Ok” to confirm this new symbol.

In a similar manner create the constants Minus, Multiply, and Divide with values of 2, 3, and 4 respectively.

Add a new Variable named ParameterBuffer and make it a longint variable.

Add an integer variable named Operator.

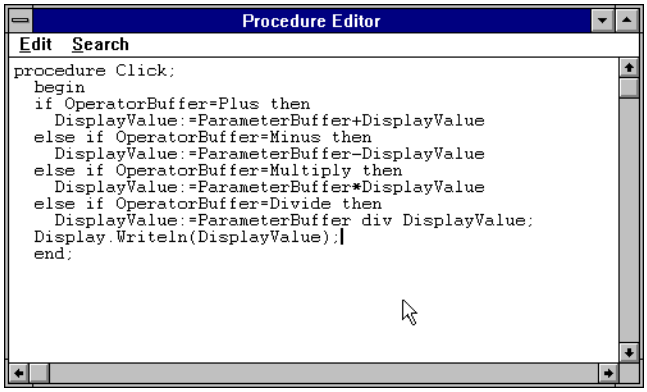
Add a new procedure with the following body:



Save your work.

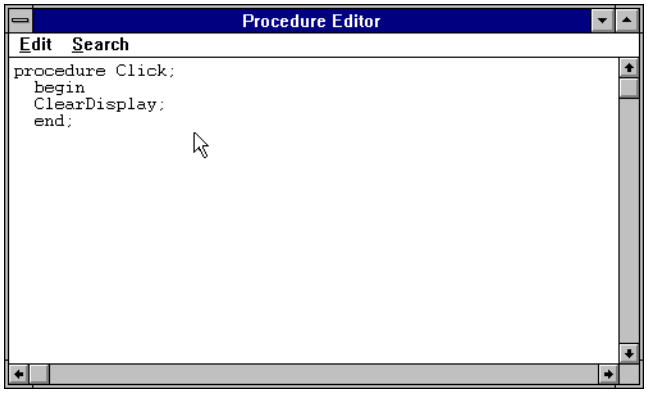
Create Equals Key

Control-Drag a copy of the Plus key below the list of operation keys. Make the legend "=" and edit the click procedure to read:



Create Clear Key

Control-Drag a copy of the Plus key below the equals key. Make the legend "C" and edit the click procedure to read:



Test the Calculator

Run the application. One possible appearance is shown below.

Click on several number keys. The number should accumulate in the calculator display. Click on an operator. Click on some additional number keys, and click on "=". The answer should appear in the calculator display. "C" should clear the display back to "0".

Close the application.

Summary

Using only the concepts learned so far a four function calculator has been built which has a more physical "feel" than many commercial Windows calculators. By sharing code sections through user procedures most click procedures were only one or two lines long containing calls with button-specific parameters.

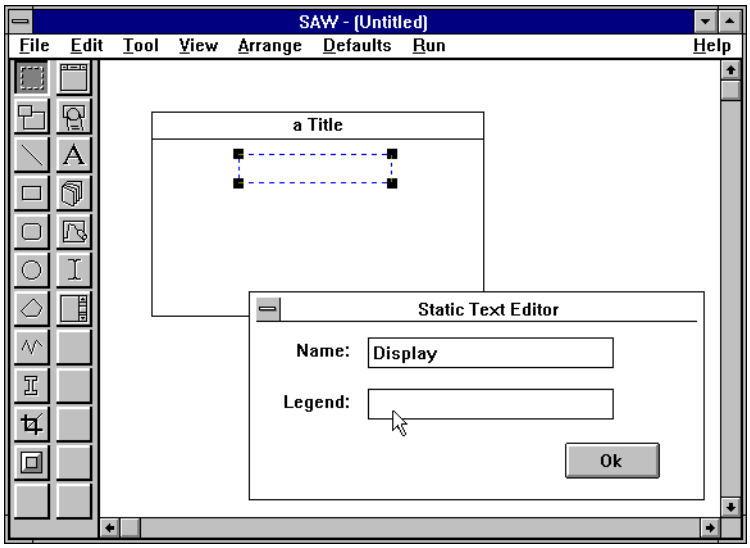
Lesson 7 - Using Plate Drag Methods

Objective

Plates support several different methods related to mouse activity including BeginDrag, Drag, and EndDrag. These methods are illustrated in this lesson.

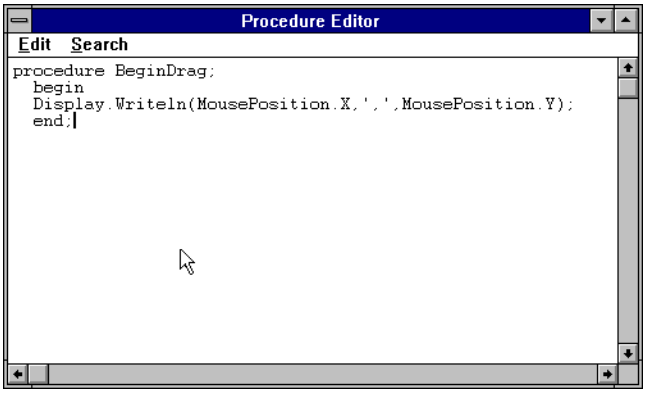
Create a Display

Use the "text" tool to create a text item named Display. Place the text in the center of the window near the top. The result should look similar to what is shown below.



Edit the BeginDrag Method

Plates respond to the mouse being clicked down on their surface by invoking their BeginDrag method. To edit the BeginDrag method double click on the default plate, select the BeginDrag method from the list of plate procedures displayed in the plate editor, and select Edit. Change the BeginDrag method to have the following text.



Run the application. Click the mouse on the plate. The coordinates of the mouse should be shown in the Display text object. Move the mouse to a different area and click again. The coordinates dislpayed should be different. The BeginDrag procedure invokes whenever the mouse button goes down (normally the beginning of a "drag" operation i.e. the name BeginDrag). The behavior in this case of BeginDrag is to write the coordinates of the mouse into the Display. MousePosition is a T2Vector that contains the coordinates of the mouse as defined in the current coordinate frame of the plate.

Edit the Drag Method

Now edit the Drag method in the plate event procedure list and give Drag the same statement as BeginDrag. Run the application again. Click the mouse on the plate and move the mouse while continung to hold down the mouse button. Note that the display coordinates change in response to the mouse position. The Drag method invokes at about 18 Hz whenever the mouse moves with the mouse button down.

Edit the EndDrag Method

Now edit the EndDrag method to contain the following statements:

```
procedure EndDrag;
      begin
      Display.Writeln('Button Up');
      end;
```

Run the application. Now when the mouse button is released the display changes from showing the coordinates of the mouse to the message "Button Up". The EndDrag method invokes when the mouse button is released. This gives you an opportunity to act on the indicated change in mouse position. You can establish a specific coordinate frame with SetCoordinateFrame.

Summary

By using the BeginDrag, Drag, and EndDrag methods it is possible to perform operations based on mouse movement. BeginDrag invokes when the mouse button goes down, Drag invokes while the mouse button moves, and EndDrag invokes when the mouse button releases.

Lesson 8 - Using Bitmap Graphics

Objective

The appearance of an application can be enhanced with several different graphical techniques. One technique, "bump" graphics, has been discussed in a previous lesson. This lesson introduces the use of bitmaps which can be made with programs such as the Paint program that comes with Windows. Bitmaps allow detailed images to be incorporated into an application.

Select the Bitmap Filename

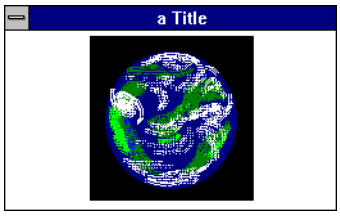
Select the File\Choose Bitmap menu item. Select the provided bitmap EARTH.BMP found in the Servo subdirectory. This is the bitmap that will be used for subsequent bitmap operations.

Crop Bitmap

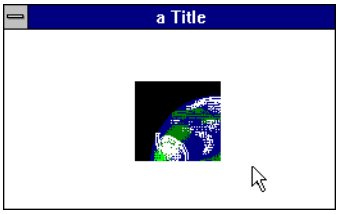
Select the "cropping" tool, the tool with the 90 degree opposing right angles. This is used to drag on the surface of a plate a region which the bitmap will "show through". The bitmap will be constrained or "cropped" to the inside of this rectangle. After selecting the tool drag a box across the default plate approximately the size of the default plate.

Position Bitmap

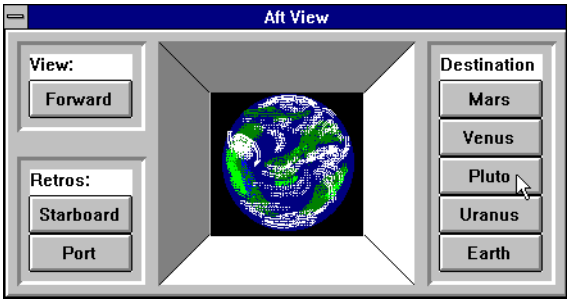
The cursor should now change to a fat cross. Drag the cross over the center of the plate and note that the bitmap image (in this case a small bitmap of earth) follows the fat cursor. Release the mouse button and the bitmap becomes "fixed" to the last position. If you are not happy with the bitmap location, select it by touching it with the "Choosing" tool, select Edit\Cut, and try again. Run the application. An example result is shown below.



Close the running application, select and cut the bitmap and place the bitmap again into the application with the cropping tool. However, this time, crop a much smaller rectangle, about half the size of the bitmap. The bitmap only is shown through the more restricted size of the cropping. The result of this smaller cropping rectangle is shown below.



Bitmaps may be used with other graphics techniques. For example, bitmaps can be used in conjunction with bump graphics to produce panels such as is shown below (application not fully functional!).



Remember that it may be necessary to order the grahics appropriately so that the you see what you would like to. Drag methods work "through" graphics

Summary

Bitmap graphics allow the addition of detailed images to Windows applications. These images can serve as a context for controls (i.e. a picture of a machine on which buttons effect corresponding aspects of a machine) and can be used with other graphic techniques such as bumps. Bitmaps may be created with the Windows Paint program or any other program that provides *.BMP format Windows bitmap files.

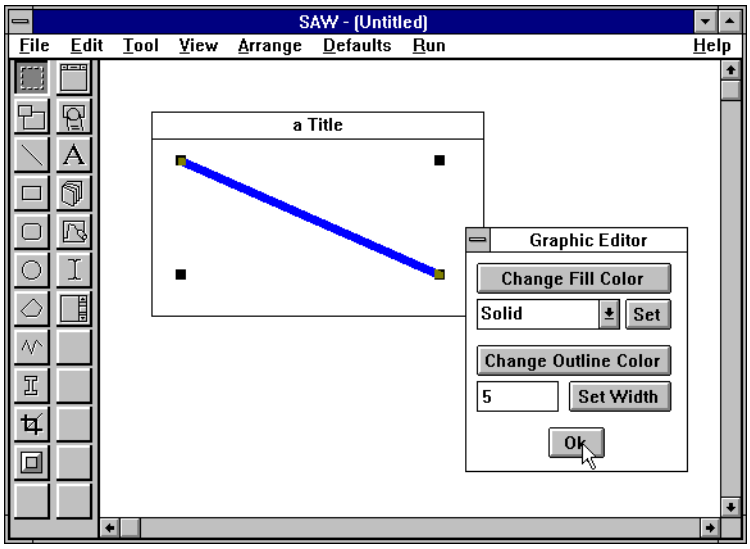
Lesson 9 - Using Geometric Graphics

Objective

Geometric graphics of various shapes are available to enhance an application's appearance. These shapes are more simply constructed than bitmaps having tools for construction as part of SAW. The creation of lines, squares, squares with rounded corners, and ellipses will be described.

Drawing a Line

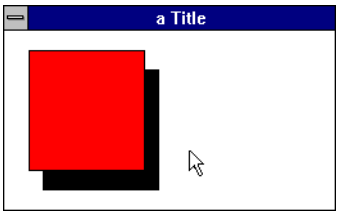
Select the line drawing tool which has the diagonal line and is positioned on the left side of the tool bar. Create a line by dragging across the surface of the default plate. Double click on the line and the graphic editor should appear allowing you to change the outline color and width of the line. Most graphics have insides as well as outsides and can have different colors and attributes for each. Lines just have outsides. The session at this point might look like the figure below.



Lines are considered "rectangular" graphics and can be resized with the four handles typical of a rectangle. Note that the selection of a line is based on the area of the described rectangle. Accordingly horizontal or vertical lines are not selectable by touching since they have no area. However they can be selected by being enclosed by the chooser tool.

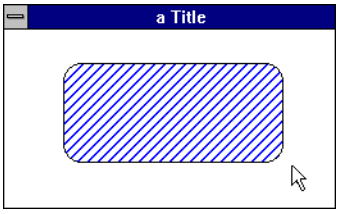
Drawing a Rectangle

The next tool down on the toolbar is the tool for drawing rectangles and has a rectangle on its surface. Select this tool and drag a diagonal to indicate the size of the rectangle. Double clicking on the rectangle produces the graphic editor allowing you to change the color of the inside, the pattern of the inside, the width of the describing outside line and the outline color. By control-dragging the selected rectangle you can make a copy the size size. By making the copy black and "sending to the back" you can create a graphic shadow effect such as is shown below.



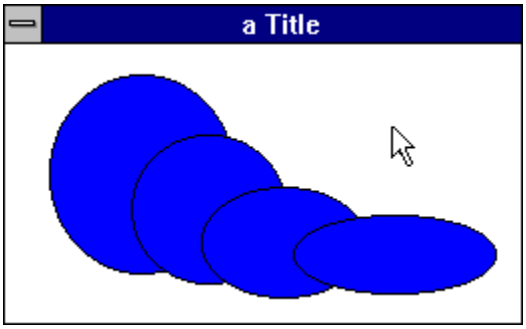
Drawing a Rounded Corner Rectangle

By using the tool below the rectangle tool you can draw rectangles with rounded corners. The figure below has a diagonal pattern,



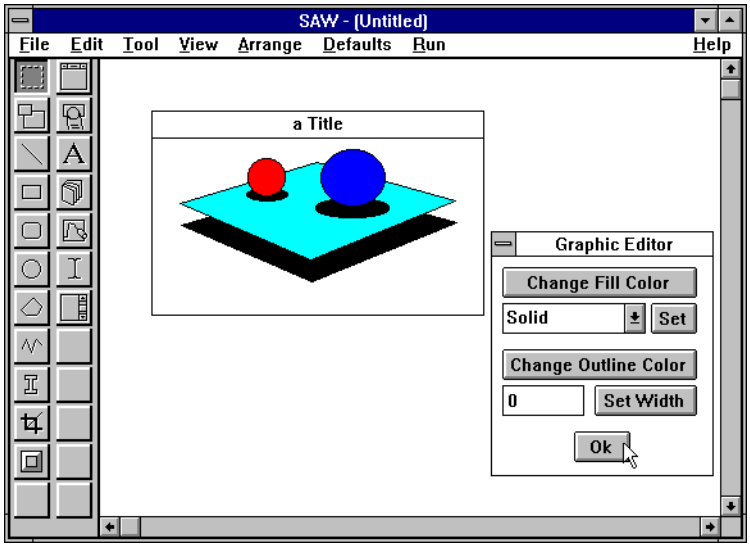
Creating an Ellipse

An ellipse can be made with the next tool down. If the sides of the dragged rectangle are the same, the ellipse will be a circle. The following figure shows several ellipses drawn across the surface of the plate.



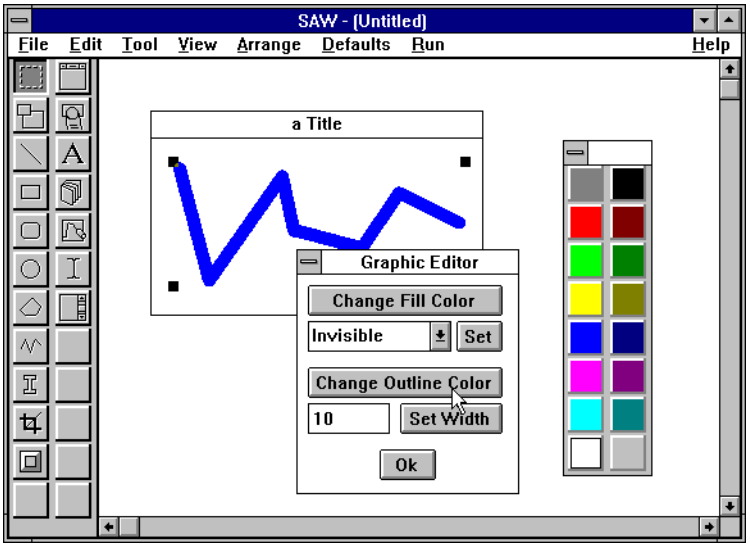
Drawing Closed Polygons

The next tool draws closed polygons. Select the tool and drag a line. After releasing the mouse button go to another point and touch the plate. An additional line is added to the polygon. Continue clicking points to add vertexes. On the last vertex double click. This concludes the operation by adding the double click vertex and connecting that vertex to the original point. It takes a bit of practice to remember to double click before you get all the way back to the original point. Creating a polygon with a shadow and two shadowed spheres is shown below.



Drawing Open Polygons

The last geometric graphic tool is the open polygon tool located just below the closed polygon tool. This tool works in much the same way as the closed polygon however the final line connecting the last vertex to the first vertex is not drawn. Below is a session where the width of the polygon has been increased to 10 and the color bar has popped up in response to a request to change the color of the polygon.



Summary

Geometric graphics can add color, shape, and context for controls and displays in an application. Using the geometric drawing tools it is possible to add lines, rectangles with and without rounded corners, ellipses, and closed and open polygons. The color and pattern of these objects can be changed by double clicking on the graphic object to produce the graphic editor which changes the attributes of the graphic.

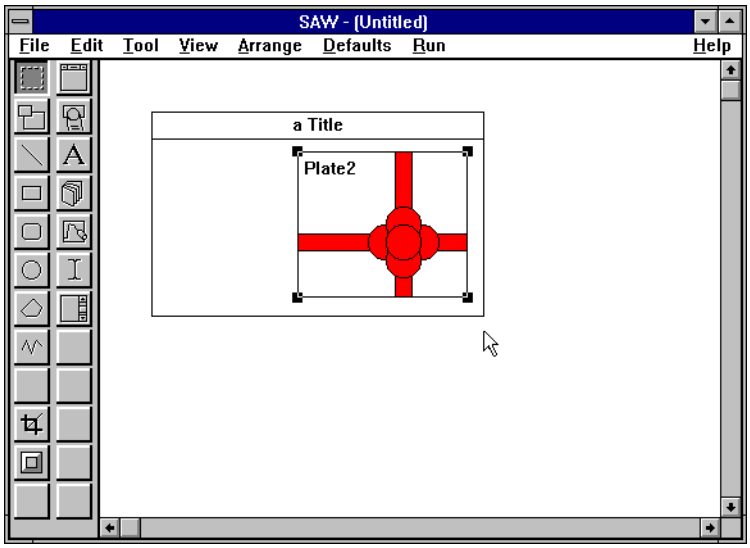
Lesson 10 - Using Attached Subplates

Objective

Hierarchy is a powerful organizational tool. SAW is able to create hierarchies of plates allowing the construction of modular, simpler, more capable applications. Subplates may be created as components that are part of a plate and relate in different ways including attached, popup, and merged. This lesson discusses the subplate relationship of being attached.

Creating a subplate

Select the plate tool which is on the right side of the tool bar at the top. Drag a square over a part of the default plate. When you release the mouse you should see a "gift wrapped " package such as is shown below.



This new subplate is drawn as a gift wrapped package with red ribbons and a red bow. Also displayed in the upper left corner of the package is the package name. As is characteristic of most objects in SAW you edit subplates by double clicking on the subplate object.

Double clicking causes the package to "open up". The parent plate on which this subplate is mounted is not displayed, and the gift wrapping is gone to show what is inside the package. Since this is a default plate, there is simply an empty plate waiting for objects to be attached to it. Double click on this plate and you will see the familiar plate editor such as was used to edit the default plate earlier.

To go back to the top level, close the plate editor by selecting "Ok", and double click anywhere outside of the plate. This causes the view to ascend back to the original default plate showing this subplate as a giftwrapped package. Double click on the subplate gift wrap again and double click on the subplate itself so as to return to the plate editor.

On the left side of the plate editor is the style selection. Click on "next" multiple times to see all of the options. Subplates have more options than the main plate. Underneath the graphical picture of each plate style is a descriptive word. The word changes from "Popup " to "Attached " to "Merged ". These are the three different ways a subplate can relate to its parent. This lesson is concerned with the attached relationship. Attached subplates are created the same time their parents are created and travel around with the parents conveying a sense of being "attached " to the parent plate.

Note that the plate editor for the subplate operates on a different plate than the main default plate. When you create a variable in the subplate, that variable is distinct from variables in the main plate. You can even use the same variable name. They remain distinct, each plate having its own set of variables. Note that the subplate, because it is a child of the main plate, is able to "see" all of the variables and objects in the main plate directly as well as its own. The main plate routines can access variables in the subplate if the subplate name is included as part of the variable "path". This is illustrated in the following exercise.

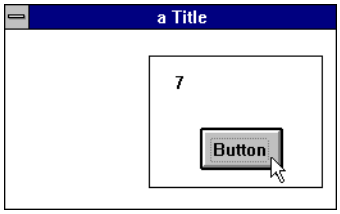
Create a variable in the subplate named "Counter " and make it an integer. Create a text object in the subplate and name it "Display ". Create a button that contains the click procedure:

```
procedure click;
      begin
      counter:=counter+1;
      Display.Writeln(Counter);
      end;
```

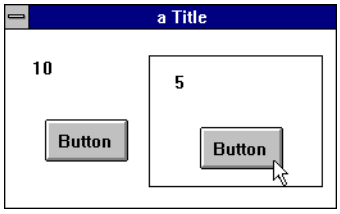
Edit the setup procedure for the plate to assign Counter to 0.

```
procedure Setup;
      begin
      counter:=0;
      end;
```

Select the subplate frame style that has the thin border. Save your work and run the application. You should see the default plate with the sub plate containing the display and button such as shown below.



Now add exactly the same elements to the top plate. Add an integer variable named Counter, and a button that increments the value and writes the value into a text object named display. You can save time making the button by descending into the subplate, selecting the button, selecting Edit\Copy, ascending back to the parent, and selecting Edit\Paste. Then position the copy of the button on the left. Edit the setup procedure for the main default plate (which is different than the one for the subplate) and assign the counter value to 0. Now run the application. Each button should operate its own seperate count display such as shown below.



Even though the variables have the same names they remain distinct. Each variable first associates with the plate "nearest" it. Subplates have access to all of the variables of the parents (and grandparents etc) without any additional help. If they have variables of their own with the same name their own variables take precedent over their parents variables. If you particularly want the parents variable you can indicate so by giving the full "path" to the parent variable. For example edit the main default plate and give it the name main (as distinct from the default name PlateXXX). Now edit the subplates button procedure to display Main.Counter instead of just Counter. Now when you run the example the subplate display reflects the value in the main display rather than its own counter variable.

This ability to use the same symbolic names and not have conflicts is a very useful feature of SAW and high level languages in general that support "scoping ". Scoping permits subassemblies to be constructed seperately from their place of use and to work correctly regardless of the environment they are placed into. In general it is good practice to keep variables "close" to their use and minimize the amount of "variable paths " which are used to travel outside an objects scope.

Scoping is particularly useful when working with Catalog components. Catalogs permit dropping into applications prefabricated subplates that contain functions such as joysticks or storage scopes. Being able to use these components without concern for name conflicts is very helpful.

Summary

Attached subplates allow an application to be arranged in a heiarchical manner with isolation between the symbols used in the parent plate and symbols used in the subplate. This isolation prevents name collisions and problems that would otherwise occur without a symbol scoping mechanism.

Lesson 11 - Using Pop-Up Subplates

Objective

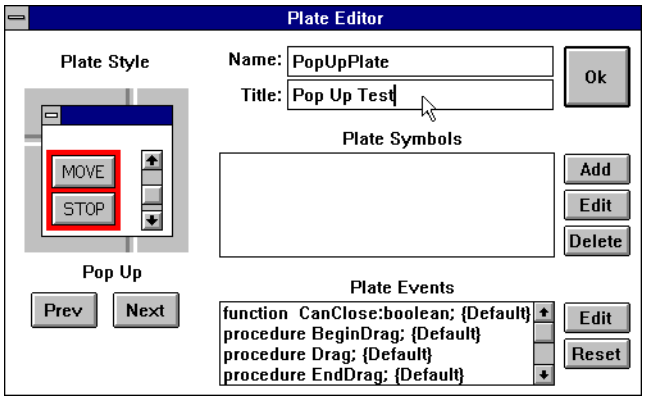
Although an application may require many controls to properly operate, it is very confusing to see all of an applications controls at one time. Pop-up subplates allow the construction of "dialog boxes" that allow the operator to focus on one particular aspect of an application's operation and then dismiss the controls after they have been used. This section describes how to create pop-up subplates.

Creating a pop-up subplate

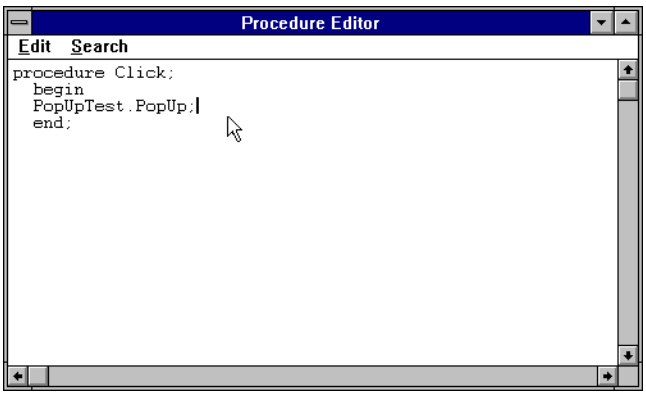
In a manner just like the creation of an attached subplate, select the plate tool which is on the right side of the tool bar at the top. Drag a square in the drawing area of SAW. The subplate does not need to reside on the surface of the main default plate, but can be off to the side or partially overlapping. When you release the mouse you should see a "gift wrapped" package.

Double click on the subplate package to open it up. Double click on the plate shown to invoke the plate editor.

The default Window style is shown on the left as "Pop Up". There are several frame styles for a pop-up subplate which can be seen by choosing "Next" and "Previous". Change the name of the plate to be "PopUpPlate " (no spaces), and change the title of the plate to be "Pop Up Test". The Plate editor should appear like the one below.



After selecting the plate style select "Ok". Ascend back to the main plate by double clicking outside of the subplate. The subplate should become gift wrapped. Add to the main plate a button. Edit that button's click procedure to be:

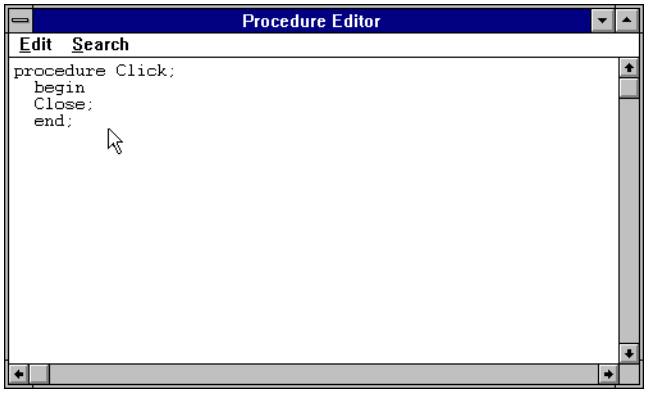


Run the application. Select the button shown on the main plate. The subplate should pop up. Double clicking on the subplates system menu should cause the subplate to disappear.

Plates which are configured to be pop-up plates recognize the method PopUp. When they pop up, these plates perform their setup procedures. Attached plates perform their setup plate procedures when the plate they are attached to first appears. Pop-up plates perform their setup procedures when they pop up. Note that the sequence of setup routines is not specified. Attached plate setup procedures may invoke before the setup procedure of their parent plate.

3) Closing Under Program Control

Pop-up plates respond to the Close method to disappear. This is necessary if you choose a frame style for the pop-up plate that does not have a system menu control. For example, edit the subplate style to be a pop-up plate with a thick border frame but no title bar or system menu control. Place on the plate a button with the legend "Ok" and the button procedure shown below.



Close is a method of the subplate. The default receiver for the Close method is the plate the button is attached to, in this case the PopUpPlate. Run the application. Click the main plate button. You should see the thick framed subplate and the "Ok" button. Select the button to cause the plate to close.

Popup plates can descend multiple levels however it is usually not good user interface design practice to have deep structures. Users tend to get lost in the structure, unable to get back to a place they once were at but can no longer find. "Flat" structures tend to work better in helping users find familiar pop up panels.

Pop up plates have all of the capabilities and privileges as the main default plate. They can have their own locally specified persistent variables, their own procedures, their own controls, and their own subassemblies.

Summary

Pop-up plates provide a simple way to create dialog type user interface components, and allows the application developer to conceal seldom used controls until requested by the user. By using subplates in a pop-up manner, an application can be made simpler to use and less overwhelming because only the relevant controls for the current operation need be seen at one time.

Lesson 12 - Using Merged Subplates

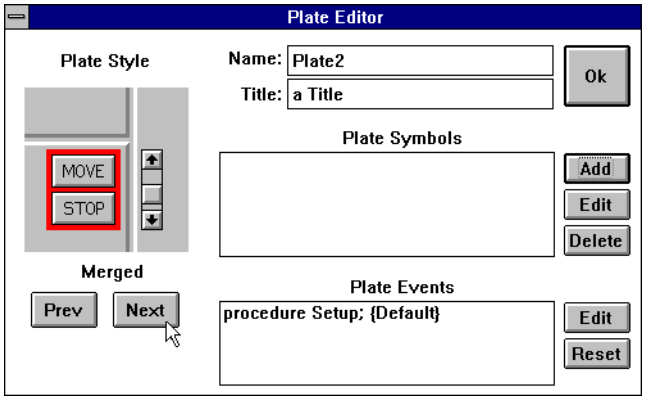
Objective

The sub-plates described so far are very independent having their own appearances, drag methods and regions. Merged plates, on the other hand, are primarily organizational packages. Components in a merged plate attach to the parent plate almost as if they had been defined on the parent plate. The only difference is that the component names are prefixed by the merged plate's name to help prevent name collisions. Merged plates are very analogous to "include files " in a text-only programming environment. The objective of this lesson is to show what a merged plate does and when you would want to use one.

Creating a Merged Subplate

In a manner just like the creation of other subplates, select the plate tool which is on the right side of the tool bar at the top. Drag a square in the drawing area of SAW within the area of the default plate. When you release the mouse you should see a "gift wrapped" package.

Double click on the subplate package to open it up. Double click on the plate shown to invoke the plate editor. Use the "Next" button near the plate style to advance through the style options unti you find a plate style named "Merged ". The appearance of the style shown is that of a set of controls on top of a graphic surface with no other apparent window features. The Plate editor should appear like the one below.

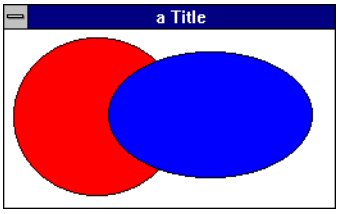


Note that when you select the merged style, the Drag procedures are not listed in the Plate Event Procedure list. Merged plates do not support

seperate drag methods because there is no "physical" plate there. Merged plates, in their unobtrusive manner, allow the drag procedures of the plates they are children of to operate "through" them. Merged Plates do retain the setup procedure which is useful for initialization. After setting the plate style to be merged press the "Ok" button to leave the plate editor.

Adding Graphics to Plates

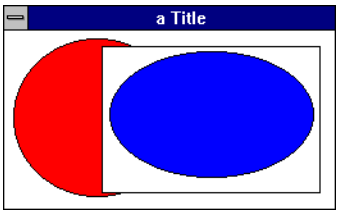
To illustrate the relationship between a merged subplate and the assembly it attaches to add a graphic to the subplate, for example, a blue ellipse. Now ascend to the parent and add a graphic which overlaps the graphic area of the subplate, for example a red ellipse. Run the application. You should see overlapping graphics such as is shown below.



The elements on the merged plate behave as if they had been directly applied to the parent assembly. Yet they remain in a group with their own internal relationships maintained.

Comparing with Attached Plate

To most easily see the difference between a merged plate and an attached plate close the application, go back to the subplate editor, and "backup" two styles with the "Prev" key to specify an attached plate with a thin black border. Run the application and you should get an appearance similar to what is shown below.



Note that the circle on the main plate is partially obscured by the subplate. Attached subplates are responsible for the entire appearance of the screen that they cover. Accordingly, if an attached subplate has a white background, that background imposes itself on the assembly obscuring what lies underneath. Merged plates, on the other hand, do not take responsibility for a section of the screen but simply add their contents to what is already in the parent allowing graphics to blend together.

Summary

Merged plates should be used when adding primarily graphic subassemblies to an application, or when adding subassemblies which do not specifically have appearance, such as a collection of procedures for operating a third party IO board. Use attached plates when you want to delegate a section of the application's appearance to another part of the program. Use a pop-up plate when you want a section of the application to come and go in the course of execution.

Chapter 5

Application Sketches

Purpose

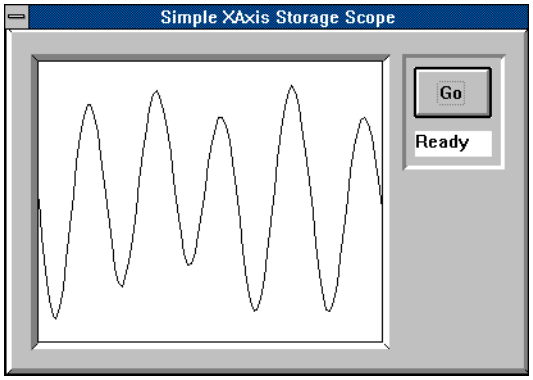
Application Sketches provide examples of certain functions can be performed. Functions such as collecting and displaying real-time data, dragging and dropping figures, and graphically presenting information in various windows are illustrated with short, simple examples that can show a particular technique or approach to the problem given the resources of Servo Application Workbench.

In the same way that a hand drawn "sketch" can convey a concept without providing a great deal of detail, application sketches convey principles of application construction. Explore the Application Sketches through the descriptions written here as well as "dismantling" the sketches provided with SAW and located in the SERVO directory.

Simple X Axis Storage Scope

Description

This SIMPSCOP.SAW example displays the position of the X Axis verses time. Pressing the "Go" button causes the data collection to start and operate for 1 second. The status line indicates the particular point in the storage scope process of Ready, Collect, and Plot. Pressing "Go" again causes a new plot to be shown.



How It Works

This example is based on two plates, one for the instrument and one for the storage scope screen, one button, one text object for the status line, and three "bumps". The plate contains the following data items:

```
const BufferSize=100;
```

The storage scope needs a buffer to hold the position information. This constant will be used to symbolically refer to the size of the buffer. By using a constant rather than a number a change to this constant can be made at a future time (for example to enlarge the collection size) and all of the uses of that value automatically effected

```
const SampleDelay=10;
```

The storage scope waits between each collection for the duration of the SampleDelay. Making this number smaller would cause the scope to collect finer resolution info but not collect as long a duration.

```
Buffer:array[1..BufferSize] of Longint;
```

This buffer is used to store the XAxis positions. Note that the upper bound is BufferSize, the constant, rather than an explicit number.

```
Procedure Plate1.Setup;
   begin
   Status.Writeln('Ready');
   end;
```

The setup procedure for the plate is used to display a "Ready" message in the text object which has been named Status.

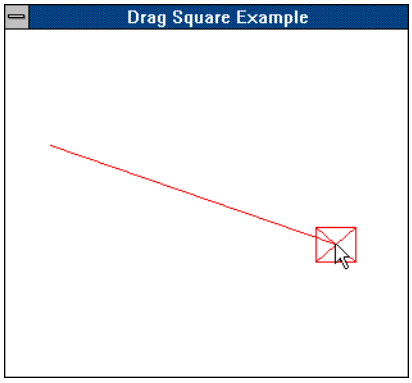
```
Procedure Plate1.Button3.Click;

   var scanner:integer;

   begin
   Status.Writeln('Collect');
   for scanner:= 1 to BufferSize do
      begin
      delay(SampleDelay);
      Buffer[scanner]:=XAxis.ActualPosition;
      end;
   Status.Writeln('Plot');
   Plotter.Clear;
   Plotter.Fit(Buffer);
   Plotter.Plot(Buffer);
   Status.Writeln('Ready');
   end;
```

All of the work occurs in the button's click procedure. First the Status writes the word "Collect" indicating that the storage scope is now collecting data. Scanner is a locally declared variable used to loop through the collection process BufferSize times. This is the advantage of using a symbol for the constant BufferSize. Both the size of the array (in the declaration) and the number of times the iterations should occur, are represented by the same number. If the size of the array is changed as well the number of iterations will automatically change to keep the two together and avoid a programming defect. Each time through the loop the program waits for SampleDelay milliseconds and then assigns the next element in the array to be the actual position of the x axis. After all of the data has been collected the status is updated to indicate that the next process is plotting. The plotter (the name of the storage scope screen) is told to clear, fit the data array, and plot the data array. The status is then told to indicate "Ready".

Dragging and Dropping a Square

Description

This DRAG_SQR.SAW example illustrates how to use the BeginDrag, Drag, and EndDrag plate event procedures to move a graphic around the window. When not being handled, the rectangle is displayed in a blue color. When dragged with the mouse, the rectangle becomes red in color and a red line is drawn from the original location of the square to the current location. When the mouse is released (dropped) the square assumes the new position and goes back to a blue color.



How It Works

This example application is composed of a single plate. The plate contains the following items:

```
const HalfWidth=10;
```

The constant HalfWidth is used to indicate how large the square drawn will be. This is both added to and subtracted from the center position of the square in each direction to result in coordinates for the square.

```
SquarePosition:T2Vector;
```

SquarePosition is a 2 dimensional vector used to remember where the square currently is.

```
OriginalPosition:T2Vector;
```

OriginalPosition is a 2 dimensional vector used to remember where the square was when the drag was started and serves as the other end of the line indicating drag displacement.

```
Procedure Plate1.UpdateSquare;
   begin
   Clear;
   DrawSquare(SquarePosition.X-HalfWidth,
      SquarePosition.Y-HalfWidth,
      SquarePosition.X+HalfWidth,
      SquarePosition.Y+HalfWidth);
   DrawLine(SquarePosition.X-HalfWidth,
      SquarePosition.Y-HalfWidth,
      SquarePosition.X+HalfWidth,
      SquarePosition.Y+HalfWidth);
   DrawLine(SquarePosition.X-HalfWidth,
      SquarePosition.Y+HalfWidth,
      SquarePosition.X+HalfWidth,
      SquarePosition.Y-HalfWidth);
   end;
```

This UpdateSquare procedure is responsible for drawing the square. The first method is Clear. Because this is a procedure defined on a plate, Clear applies to the associated plate. DrawSquare, similarly, is sent to the associated plate to produce the outline of a square. DrawLine is used to produce the diagonal lines inside the square. Update causes the appearance to take effect.

```
Procedure Plate1.UpdateDragLine;
   begin
   DrawLine(OriginalPosition.X,OriginalPosition.Y,
      SquarePosition.X,SquarePosition.Y);
   end;
```

The procedure UpdateDragLine draws a line from the OriginalPosition to the SquarePosition and will be used to indicate drag displacement.

```
Procedure Plate1.BeginDrag;
   begin
   SetLineColor(Red);
   OriginalPosition:=SquarePosition;
   SquarePosition:=MousePosition;
   UpdateSquare;
   UpdateDragLine;
   Update;
   end;
```

BeginDrag sets the line color for drawing to Red, remembers the OriginalPosition of the square, sets the square's new position to be the mouse position, and updates the square and the line representing the drag displacement. Note that assignment between vectors is supported, i.e. all of the components of MousePosition are assigned to the respective components of SquarePosition, for example. In general this is true for record , object, and array types.

```
Procedure Plate1.Drag;
   begin
   SquarePosition:=MousePosition;
   UpdateSquare;
   UpdateDragLine;
   Update;
   end;
```

Drag assigns the SquarePosition to the current MousePosition and updates the image.

```
Procedure Plate1.EndDrag;
   begin
   SetLineColor(Blue);
   SquarePosition:=MousePosition;
   UpdateSquare;
   Update;
   end;
```

EndDrag puts the color back to blue, updates the square position one last time and redraws the square.

```
Procedure Plate1.Setup;
   begin
   SetCoordinateFrame(-100,-100,100,100);
   SquarePosition.Init(0,0);
   SetLineColor(Blue);
   UpdateSquare;
   Update;
   end;
```

The plate Setup procedure establishes a coordinate frame for the plate. These coordinates are used for drawing lines and reporting mouse position. The SquarePosition is initially set to be the center of the window and the initial color is Blue. The square is drawn with the UpdateSquare routine.

Questions and Answers

Question: Why didn't the plate name have to proceed the use of plate functions, such as DrawLine?

Answer: The standard format for communicating to objects is to indicate the name of the object (the receiver), the operation to perform (the method) and any parameters that might be used by the method. Because these procedures are part of the plate itself, the plate serves as the default receiver for the methods. If you had wanted to write on a different plate, for example a child plate with respect to this plate, you would have had to use that plate's name explicitly to indicate you were not referring to the default plate.

Technique Applications

Dragging and dropping objects is a very convenient way to relate to a machine when using a computer. Imagine that the EndDrag method included the statement:

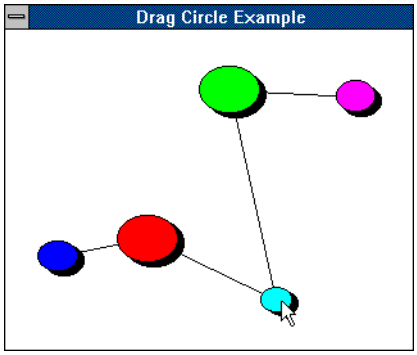
```
XYAxis.MoveTo(MousePosition).
```

The coordinate space for the window is set during the Setup routine and can be set to indicate the count-sized space of the machine. The Window would then represent the entire range of a 2 axis machine's operation. As well as the image being dragged and dropped, a servo controlled mechanism can be dragged and dropped. This provides a powerful "endpoint specification" technique for two dimensional mouse joystick movement. In a similar manner, the SquarePosition could be constantly updated by a scheduled task to have the current actual position of the machine shown on the display as a picture moving in two dimensions. This allows the construction of visual model of the machine's activity with the connection going both ways.

Mouse Indicated Selection

Description

This DRAGCIRC.SAW example illustrates how to select one item from among several for dragging. The 5 circles in the figure can be individually dragged to a new location by clicking within their radius, dragging to a new location, and releasing the mouse.



How It Works

This example application is composed of a single plate. The first constant is NumberOfCircles:

```
const NumberOfCircles=5;
```

The constant NumberOfCircles is used to size arrays which contain circle information and as an upper bound for iterating over the circles in the application.

There are three arrays in the application:

```
var CircleLocation:array[1..NumberOfCircles]
   of T2Vector;
var RadiusArray[1..NumberOfCircles] of integer;
var ColorArray[1..NumberOfCircles] of integer;
```

These arrays contain the attributes for the circles numbered 1 through 5. Each circle can have its own location, radius, and color. These arrays are initialized in the setup procedure of the plate which states the following:

```
procedure Setup;
  begin
  SetCoordinateFrame(-100,-100,100,100);

  CircleLocation[1].Init(-80,-80);
  CircleLocation[2].init(-30,-30);
  CircleLocation[3].init(0,0);
  CircleLocation[4].init(30,30);
  CircleLocation[5].init(80,80);

  ColorArray[1]:=blue;
  ColorArray[2]:=red;
  ColorArray[3]:=cyan;
  ColorArray[4]:=green;
  ColorArray[5]:=magenta;

  RadiusArray[1]:=10;
  RadiusArray[2]:=15;
  RadiusArray[3]:=8;
  RadiusArray[4]:=15;
  RadiusArray[5]:=10;

  UpdateDisplay;
  end;
```

The Setup procedure sets initial values for each of the 5 locations, colors for each of the 5 circles, and radius values. The last call is UpdateDisplay, which draws the circles. UpdateDisplay has the following description

```
Procedure UpdateDisplay;

  var scanner:integer;

  begin
  Clear;
  for scanner:=1 to NumberOfCircles-1 do
    DrawLine(
   CircleLocation[scanner].x,CircleLocation[scanner].y,
  CircleLocation[scanner+1].x,CircleLocation[scanner+1].y);
  for scanner:=1 to NumberOfCircles do
    DrawCircleFigure(scanner);
  Update;
  end;
```

Scanner is declared to be a local variable. This will be used to iterate over the five circles. The first instruction clears the plate so as to have a clean drawing surface. Four lines are drawn connecting the center points of the 5

circles together. Five circles are then drawn by the routine DrawCircleFigure. This routine has the following description:

```
Procedure DrawCircleFigure(index:integer);

  const ShadowOffset=3;
  var Radius:integer;
  var Location:T2Vector;

  begin
  Radius:=RadiusArray[index];
  Location:=CircleLocation[index];
  SetBodyColor(black);

  {draw shadow}
  DrawEllipse(
    Location.X-Radius+ShadowOffset,
    Location.Y-Radius-ShadowOffset,
    Location.X+Radius+ShadowOffset,
    Location.Y+Radius-ShadowOffset);
  SetBodyColor(ColorArray[index]);
  DrawEllipse(
    Location.X-Radius,
    Location.Y-Radius,
    Location.X+Radius,
    Location.Y+Radius);
  end;
```

The circle being worked on is indicated by the index. The radius and location are loaded from the corresponding arrays. The shadow is then drawn by offsetting the location of the circle and using a black color. The circle itself is then drawn using the color of the circle found in ColorArray.

The BeginDrag method is used to determine which of the 5 circles is being selected for movement. The BeginDrag procedure has the following description:

```
procedure BeginDrag;

  var DeltaVector:T2Vector;
  var Length:integer;
  var MinimumDistance:integer;
  var scanner:integer;

  begin
  for scanner:=1 to NumberOfCircles do
    begin
    DeltaVector:=
      MousePosition-CircleLocation[scanner];
    Length:=DeltaVector.Length;
    if scanner=1 then
```

```
      begin
      MinimumDistance:=Length;
      MoveIndex:=1;
      end
    else if Length < MinimumDistance then
      begin
      MinimumDistance:=Length;
      MoveIndex:=scanner;
      end;
    end;
  CircleLocation[MoveIndex]:=MousePosition;
  UpdateDisplay;
  end;
```

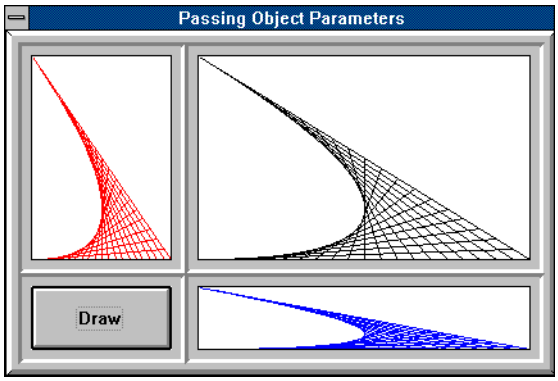
For each of the cricles the DeltaVector is calculated, which is the vector pointing from the circle to where the mouse clicked. This shortest length and corresponding circle index is then remembered to identify the nearest circle. The location of the nearest circle is updated to be the mouse position and the display is updated to show this change. The currently selected circle is remembered in the MoveIndex. The Drag procedure uses this to continually update the circle location as the mouse is dragged:

```
procedure Drag;
  begin
  CircleLocation[MoveIndex]:=MousePosition;
  UpdateDisplay;
  end;
```

Passing Object Parameters

Description

PASS_OBJ.SAW illustrates how parameters to functions can be objects. In this example three different plates are passed as parameters to a function which performs a drawing operation. Each plate has its own color, and each can independently zoom in on the image that has been placed into it.



How It Works

This example is composed of four plates, 5 bumps, and one button. The plate contains the following procedure:

```
Procedure Plate1.PlaceDrawingIn(
   var aPlotter:TPlate);

   var scanner:integer;

   begin
   aPlotter.Clear;
   aPlotter.SetCoordinateFrame(0,0,200,200);
   for scanner:=1 to 20 do
      aPlotter.DrawLine(scanner*10,0,
         200-(scanner*10),scanner*10);
   aPlotter.Update;
   end;
```

This procedure clears the plotter passed as a parameter, sets the coordinate frame for that plotter, and then places into the plotter 20 lines in a sweeping pattern. After the lines have been drawn the plotter is asked to update its appearance. Note that an object, such as a plotter, can be passed as a parameter. The variable aPlotter serves as a reference to the plotter in the caller. This is called pass-by-reference.

The following setup procedure establishes which plotters are which colors:

```
Procedure Plate1.Setup;
   begin
   Plotter1.SetLineColor(red);
   Plotter2.SetLineColor(blue);
   Plotter3.SetLineColor(black);
   end;
```

The drawings are made by the button which has the following click procedure:

```
Procedure Plate1.Button5.Click;
   begin
   PlaceDrawingIn(Plotter1);
   PlaceDrawingIn(Plotter2);
   PlaceDrawingIn(Plotter3);
   end;
```

Questions and Answers

Question: What's the difference between "call-by-reference" and "call-by-value"?

Answer: Pascal parameters to procedures and functions are normally call by value. What this means is that the procedure or function operates on a copy of the variable provided by the call. If the procedure or function assigns a new value to the variable that change is not reflected in the variable found in the callers "scope". Parameters can be thought of as temporary variables. By placing the word " var " in front of a procedure or function parameter you are indicating that the parameter is to be "called-by-reference". This means that you are not working with a copy, but changing and effecting the object that was provided by the call. In general objects that you intend to modify should be passed as var call-by-reference parameters since your changes would not take effect with a call-by-value parameter. In this case it doesn't make any difference since TPlotters are almost completely indirect anyway.

Chapter 6

6) The Douloi Pascal Language

Introduction to the Language System

Purpose

Motion Server through SAW uses a procedural language similar to Object Based Pascal to express behavior and relationships in a motion control application. This language system follows many of the principles of Pascal allowing developers with pascal background to feel at home with a familiar tool. The purpose of this section is to describe the capabilities and use of the language. Application of the language is best understood by working through the tutorial indicated in the last section. This section may be helpful, when encountering an unfamiliar construct, for understanding that construct through a more detailed explanation than is provided in the tutorial.

Throughout this section there will be italic paragraphs that provide a detailed explanation of related principles to a topic being covered. These paragraphs are provided to answer questions advanced developers may have but are entirely optional. If you find the initial italic paragraphs helpful, read future ones, and if not skip future ones.

Language Overview

Motion Server through SAW may be programmed through the provided Object Based Pascal language referred to as Douloi Pascal. The instructions you write are compiled into 32 bit 386 object code for quick execution. During this compilation step certain programming errors might be detected. SAW attempts to provide you with an opportunity to correct those errors by presenting the offending instructions in a suitable editor for you.

Douloi Pascal is strongly typed which means that everything you say must be clearly explained. Strongly typed languages generally do not "help" you by automatically creating variables or accepting as correct different types of parameters for a function call but rather insist on issues being clear up front.

Question: Why is the language described as "Object Based" rather than "Object Oriented"?

Object oriented languages are making important inroads as industrial tools, primarily through C++ . The definition of an object oriented language is not completely standardized however the general consensus includes attributes such as polymorphism, encapsulation, and inheritance.

Douloi Pascal does contain the concept of an object , an aggregate of state information with related operations to alter that state. Douloi Pascal uses the familiar variable.method syntax rooted in record field access which has become standard for languages such as Visual Basic , C++ and Turbo Pascal . However Douloi Pascal does not meet the "standard" criteria .

Although there is object structure this structure is publicly accessible and not "private" limiting the degree of encapsulation which may be claimed for the language.

Generally, polymorphism requires dynamic allocation so that a single indirect data item could refer to different objects. Douloi Pascal does not support dynamic allocation so there is no polymorphism possible.

Although many of the system objects are internally constructed with inheritance relationships there is no inheritance mechanism available at the application level, i.e. you cannot create two object types and have one be a descendent of the other.

The literature's common response to languages which display some object oriented behaviors but not these three in particular is to refer to those languages as Object Based rather than Object Oriented.

Question: Why was Object Based Pascal Chosen?

Douloi Automation believes that strong typing and as much compile-time error detection as possible is needed, particularly for motion control applications. It is not appropriate to discover programming errors at run time in procedures with names like AbortMotion . Procedures such as these need to work since machine and operator safety can be influenced by program correctness. Object principles effect the architecture of the language very much. Douloi Automation believes the leading language system that provides very strong type checking, Object characteristics , and industrial familiarity is Object Pascal.

Question: What are the primary differences between Object Pascal (such as Borland's Turbo Pascal for Windows) and Douloi Pascal?

Douloi Pascal is designed with fundamental trade-offs primarily related to its use with real time motion applications. The following list of differences reflects these trade-offs.

Douloi Pascal generates 32 bit 386 object code for fast program execution speed. Most compilers need to generate code compatible with 8086 and 80286 machines and accordingly generate 16 bit code.

Douloi Pascal does not provide dynamic data management . All data structures are defined as either global or local (stack based) at compile time.

Douloi Pascal does not support inheritance at the application level

Douloi Pascal does not support local procedures (i.e. procedures declared within the bodies of other procedures. If you are a C or C++ programmer you won't miss this since those languages don't support local procedures either).

Douloi Pascal provides built-in support for multitasking. Multitasking is important for motion control application development.

Douloi Pascal has limited code and data sizes. Data is limited to approximately 70K maximum data memory (16KBytes default) and Code to approximately 256K bytes. Although this may seem incredibly small bear in mind the following points:

(1) Borland's original release of Turbo Pascal generated programs with only one code segment and one data segment (approximately 60K bytes each) but was still an extremely useful tool.

(2) By definition a "real time" application must be quick. There simply is not enough time to execute a large amount of code in a real time application.

(3) Because all of the user interface for SAW is done through Windows in standard mode this data and code space is only needed for what is critically real time. Windows memory , up to 16 megabytes, is still available for windows controls , bitmaps, graphics etc. Much of the functionality of a SAW based application is through behaviors of these objects separate from the real time code which consumes none of this space.

Douloi Pascal allows object procedures and functions to be elaborated directly inside the object type definition.

Douloi Pascal does not support sets directly, although ordinal constants can be used to simulate set behavior .

Douloi Pascal uses "null terminated" strings since this format is most standard for DLL compatibility. The default string length is 32 characters (not 255) but strings of various lengths can be specified.

Formats and Conventions

Douloi Pascal, like pascal in general, is a “free format” language. You are not obliged to complete a single command on one line. It does not matter where the command begins, i.e. you do not have to start in column 1. In order to make code as readable as possible, however, it is a good idea to follow a format guideline. There are many different conventions religiously defended by as many corresponding users. The chapter titled "Program Formatting" contains a particular format used by Douloi Automation and encountered in the demo programs . Other conventions are described in books on Pascal programming. Which convention you use does not matter however using some convention consistently does matter. Use whatever convention makes the most sense to you.

Because commands can start on one line and end on another you need to be explicit about where a command actually ends. In Pascal the end of a command or “statement” is indicated with a semicolon. The semicolon serves to both clarify possibly ambiguous cases and to “check up on you” to insure that what you believe to be a complete statement is what the compiler also believes to be a complete statement.

Douloi Pascal supports and invites the use of symbolic names . These symbols are used to represent your problem and the corresponding solution. The clarity of a program and its ability to be correctly written the first time, corrected, and maintained is dramatically effected by the choice of symbolic names. The proper description of a problem is half of the distance to having a solution and much of the description occurs in the naming process of the operations and information in the program. If you are unable to meaningfully name a variable this may be a strong indication you don’t fully understand the problem. There is no run-time penalty for lengthy variable names . There is no merit to long names for the sake of simply being long however long names are sometimes required in order to be clear. The only mainstream method of keeping tight correspondence between documentation about a program and the program itself is to

have the program be self-documenting, i.e. writing the program with such expressive symbol naming that to read the program is to read a description of the program.

Douloi Pascal allows symbols to be up to 60 characters long. Symbols are generally composed of letters, numbers, and the underscore character but should not contain other characters (i.e. no spaces). There is no distinction made between capital and small letters. One symbol naming school of thought is to have multiple word symbols separated by the underscore as if it was a space, for example:

```
drill_head_home_position
```

Another school of thought is to use capitalization to emphasize where a new word begins, for example:

```
DrillHeadHomePosition
```

The second approach is used for predefined objects and methods in Douloi Pascal so your program may appear more consistent using the second method. Either choice is fine.

Another industrially encountered convention used by Douloi Pascal is a capital "T" before symbols that are type definitions, i.e. the predefined 4 dimensional vector type is called "T4Vector". The leading capital T is a clue that the symbol has something to do with type definitions. Again this is optional but you may choose to conform to the standard for the sake of consistency.

Douloi Pascal supports curly bracket comments. Anything inside curly brackets is disregarded and may be used to provide notes to describe the nearby program. Comments are not nestable, i.e. the one thing a comment cannot contain is another comment. The following program fragment indicates how a comment might be used:

```
procedure RaiseDrillHead;
  {This procedure is used to move the head clear of
   the table and should be called before any table
   movement is performed}
  begin
  ZAxis.MoveTo(DrillHeadClearancePosition);
  end;
```

Variables

Purpose

The language system's purpose is to help you organize a solution to your problem, and a basic need in getting organized is having a place to put things. Variables provide the places where information about your application is put. Douloi Pascal provides simple variables, aggregate variable mechanisms and arrays to help provide appropriate data structures to store information.

Fundamental Types

Variables come in different shapes and sizes (known as "Types") to accommodate the different shapes and sizes of information that you need to put somewhere. The fundamental types provided by Douloi Pascal include:

Boolean

Booleans are able to remember yes/no information. Boolean variables are helpful for decision related information and often have names such as

```
PartHasBeenRemoved
ProcessHasFinished
PointIsSelected
```

Integer

Integers are scalar numbers that have a range of -32767 to 32768. These are typically used for "small" numbers and use 16 bits of storage. Integers are often used for iteration and quantities such as:

```
NumberOfPartsRemaining
DataBufferIndex
CurrentMotorTorque
```

Longint

Longint, or Long Integers, are 32 bit quantities and may have a range of about -4 billion to +4 billion. Longints are often used for position information expressed in counts (which can cover this range). Longints might have names such as:

```
RegistrationPosition
InputCapturePosition
DistanceToGo
```

String

Strings are variables which can contain words. Douloi Pascal supports variable length strings with the default being 31 characters although you can specify different sizes when you describe the variable. Examples of string variables names might include:

```
PartName
ErrorMessageText
LastRecordedCommand
```

Single Precision IEEE Floating Point Reals

Single precision reals are 32 bit variables which contain about 7 or 8 signficant digits of resolution over a range of 10 ^ -37 to 10 ^ 38. Floating point operations in single precision reals are available even if your machine does not have a math coprocessor. The floating point routines operate about one sixth the speed of a 32 bit longint multiply so you do incur a speed penalty for using floating point. If you need more resolution or more speed the best approach is to have a math coprocessor which is supported as a "calculator" for use by application programs.

Double Precision IEEE Floating Point Reals

Double precision reals are 64 bit variables which contain about 15 or 16 significant digits of resolution over a range of 10 ^ -307 to 10 ^ 308. Infix notation floating point operations in double precision reals are available even if your machine does not have a math coprocessor. However the math coprocessor is much faster if you have one. Double precision reals are provided primarily for the benefit of DLL connectivity since some DLLs require parameters of this type. For real-time calculations, however, it is much better to use the math coprocessor or to user single precision or integer formats for your calculations.

Usage

Variables are most easily used when given meaningful names and meaningful structure. You express your desire to have a variable through a variable declaration. Many of the variables you'll use will be "plate" variables created with the "plate editor" in Servo Application Workbench. The plate editor allows you to add, delete, and change variables with a dialog box and select the type you would like the variable to have from a list. You may also declare variables in procedures and functions for use locally within those procedures and functions using the "var" statement. This statement has the form:

```
var <variable name> : <variable type> ;
```

After the word "var" is a space followed by the name you would like to use for the variable. After the name (no space required) is a colon, and after the colon (no space required) is the variable type followed by a semicolon (which means "end of statement"). Example variable declarations include:

```
var MoveDistance:longint;
var FirstTimeThroughProcedure:boolean;
var TorqueValue:integer;
var UserMessage:string;
```

Variable names must not have any spaces (although you may use the underscore character) and in general should only be composed of letter and numbers. Variable names may include numbers but may not begin with numbers.

Aggregate Types

You may use a "desk organizer" in your desk drawer. A desk organizer contains various compartments for holding different types of things commonly found in a drawer such as paper clips, pencils etc. The desk organizer is a single, integrated item that has internal structure of various compartments, each compartment providing a suitable place for a particular aspect of its storage purpose. In a similar manner most high level programming languages allow the construction of such informational "desk organizers", data structures that contain "compartments" where information can be stored while retaining a relationship with the whole. Douloi Pascal supports aggregate structures through Record and Object type declarations.

The creation of new Record and Object structures will be deferred until the Advanced Language section however the use of predefined record and object types will be reviewed here. Predefined aggregate types include:

```
T2Vector..........2 dimensional vector
T3Vector..........3 dimensional vector
T4Vector..........4 dimensional vector
T5Vector..........5 dimensional vector
T6Vector..........6 dimensional vector

T1Axis............single axis of motion
T2Axis............2 coordinated axis of motion
T3Axis............3 coordinated axis of motion
T4Axis............4 coordinated axis of motion
T5Axis............5 coordinated axis of motion
T6Axis............6 coordinated axis of motion

TButton...........Windows Button Control (SAW only)
TEdit.............Windows Edit Control (SAW only)
TFile.............DOS File (SAW only)
THPGLFile.........HPGL File Interpreter (SAW Only)
TListBox..........List/Combo box style control (Saw Only)
TPlate............Assembly "Base" for applications (SAW only)
TPrompter.........Modal dialog for operator (SAW only)
TText.............Output text/display (SAW only)
```

The first group are different types of vectors and are collectively referred to as TNVector. In actual usage the “N” in TNVector is replaced with a number between 2 and 6 representing the dimension of the vector. These vectors contain vector “component” compartments which are longint types. The names of the compartments are X,Y,Z,U,V,W if you have all six. The language definition for a T3Vector, for example, is:

```
Type T3Vector=object
X:longint;
Y:longint;
Z:longint;
   {object procedures}
   end;
```

A “compartment” or “field” of an object or record is indicated with a period after the variable name followed by the field name. The period indicates that the name following is specifically related to that variable. Field names can indicate component storage areas in an aggregate variable such as a record or object. For example, to assign

the Y component of a variable named HomePosition of type T3Vector you could use the command:

```
HomePosition.Y:=500;
```

In pascal the assignment operator is a colon followed by an equals sign indicating that the expression on the left is to receive information from the right side.

"Records" contain only storage areas. "Objects" contain storage areas just like records and as well type specific procedures and functions which can operate on that information.

The second group of objects listed are axis groups ranging from T1Axis, a type to represent a single axis of motion, up through T6Axis, a type to represent a 6 axis coordinated group. These axis group types are collectively referred to as TNAxis types where the "N" in TNAxis is replaced by a number between 1 and 6 in actual usage. You might create a variable to represent a 2 axis positioning table with the command:

```
var PositioningTable:T2Axis;
```

Now you have a variable named PositioningTable which can be given commands. The first command necessary is an initialization command to associate that variable with actual axis in the system, for example:

```
PositioningTable.Init(XAxis,YAxis);
```

XAxis and YAxis are predefined variables of type T1Axis which correspond to the physical X and Y axis in the servo system. Now the PositioningTable can be given a command to move, for example:

```
PositioningTable.SetServo(On);
PositioningTable.MoveBy(20000,30000);
```

Detailed information about the predefined variables and operations supported by TNVectors and TNAxis can be found in the help system.

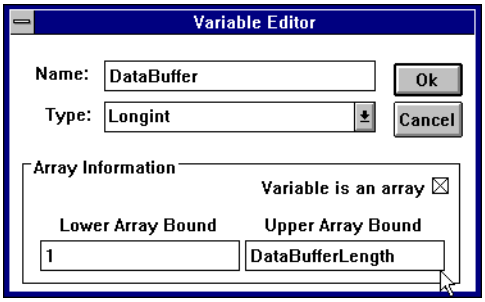
The remaining predefined types are for use specifically with Servo Application Workbench. Some of these types are declared graphically while others are declared with the “var” keyword. In general these other types provide Windows components that can be manipulated by the programs you write to display information and create program behaviors.

Arrays

Up until this point variables have been named symbolically and their access has been explicit on the part of the programmer. Arrays are a tool that give the program itself the ability to specify what storage area will be used to save information.

Arrays are linear arrangements of variables of a particular type. When referring to information in an array you refer to the name of the array itself and to which particular item in the array you are concerned with, for example the first one, or the third one. The location of the storage area of interest can be expressed numerically giving the program the ability to indicate a desired location.

Arrays may be specified in SAW as plate variables through a dialog box such as shown below:



Any variable has the capacity to be part of an array. When defining a variable you can check the box indicating the variable represents an array and indicate the array bounds. Douloi Pascal allows the lower bounds and upper bounds of an array to be set. An example of an array being declared is shown below. Note that the bounds may be symbolic constants.

The C language, as I understand it, compels the lower bound to be 0. The

reason for having the lower bound be 0 is one less adjustment of the index before accessing the array in the assembly language code which is generated. The Douloi Pascal Compiler, however, offsets the array's basis when referencing the array eliminating this overhead. Accordingly there is no run time penalty for using a non-0 lower bound.

Arrays of longints are frequently used as "storage scope buffers" to collect position or servo information at the controller sample rate to be later plotted showing step response results or other information.

To create a variable array start with the VAR keyword and follow it with the name of the array. Following the name place a colon, as you would for a normal variable declaration. Following the colon place the keyword "array" followed by a left square bracket, the lower bound either as a number of constant, two periods, the upper bound as a number of constant, a right square bracket, the word "of" and then the type of the array's components followed by a semicolon. An example of an array declaration might be:

```
var DataBuffer:array[1..DataBufferSize] of longint;
```

Assignment

Purpose

Having a place to put everything is a good start for getting organized, however no progress is made until those storage areas get used. Assignment is one of the main ways information gets placed into a storage area. This section explains assignment for simple types and the implications of assignment for advanced types.

Assignment of Simple Types

In Douloi Pascal the assignment operator is a colon followed by an equals sign. For example the variable DrillDepth, a longint, might be assigned with the statement:

```
DrillDepth:=500;
```

The semicolon at the end of the statement indicates that the programmer believes the statement to be complete.

Assignments to single or doubles types allows the use of a period to indicate the decimal point, i.e.

```
FractionalUserVariable:=6.5;
```

Strings may be assigned by using single quotes, i.e.

```
MachineStatus:='Ready';
```

Upper and lower case letters in string constants are kept distinct. Note that although the hardcopy of the manuals (or even this page) may show a “back quote” for the leading quote this is a result of the printer that produced the documentation. The intended symbol is always the conventional single quote, most often found under the double quote to the right of the semicolon and the left of the enter key.

Variables may be assigned from another variable, i.e.

```
CurrentPosition:=LastPosition;
```

Variables of the same type may always be assigned. Variables of different types are converted automatically if the assignment falls into the following cases:

A longint can be assigned from an integer. The integer always fits. An integer may be assigned from a longint however some information may be lost. It is the your responsibility to make sure you are not assigning a value beyond the integer range. Singles and doubles may be assigned to integers only after using the "ROUND" operation, i.e.

```
MyInteger:=Round(MyRealSingle);
```

Integers and longints may be assigned to singles and doubles however it is the programmer's responsibility to insure they will properly fit into the range of the chosen floating point type. Doubles can hold most anything, however singles will lose information if a very large longint is assigned.

Assignment of Aggregate Types

Aggregate types are often assigned on a component by component basis. For example a variable named HomePositionVector of type T3Vector might be assigned with the statements:

```
HomePositionVector.X:=500;
HomePositionVector.Y:=600;
HomePositionVector.Z:=700;
```

Because this is tedious most object aggregates include procedures that consolidate assigning all of the object component values into a single statement, usually named Init. The same 3 dimensional vector from the previous example may be assigned in a single statement:

```
HomePositionVector.Init(500,600,700);
```

Question: Is Init a constructor?

Answer: Generally no, Init is a conventional procedure. Because Objects in Douloi Pascal are not dynamic and do not support inheritance there is no virtual method table. Procedure links are established at compile time. Accordingly Douloi Pascal does not use constructors or destructors for dynamic allocation or VMT needs. However there are some SAW Objects, such as TPrompter and TFile, which when initialized in a SAW program result in the creation of a dynamic object in the Windows environment.

When used in assignment statements all of the components of the source object or record (on the right side) are copied into the corresponding components of the destination (on the left side). For example if the following symbols were 6 dimensional vectors of type T6Vector this statement would copy 6 longints:

```
TaughtPositionVector:=CurrentPositionVector;
```

When records or objects are copied in this manner there is never any type conversion. Records and objects in an assignment statement must be of the same type.

Constants

Purpose

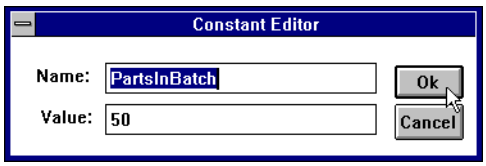
Many different tools are needed in the process of getting the problem and solution organized. This section describes the role constants play in program readability and maintenance.

Description

Constants provide a way to give a symbolic name to a piece of information that you do not intend to change in the course of a program's execution. For example if you are writing software for a machine that produces 50 parts in every batch of parts it manufacturers you might create the constant:

```
const PartsInBatch = 50;
```

Constants can also be created as plate symbols using the Constant Editor such as shown below.



The equals sign is used instead of the assignment operator to convey that these two things, PartsInBatch and 50, are equivalent. There is no "copy" activity that occurs in the program at run time. Using a constant is just as efficient as using the explicit number in terms of the resulting code however the program is much easier to understand. The following statement may appear in the application program:

```
if NumberOfPartsMade = PartsInBatch then......
```

It's quite clear that the decision has something to do with the batch being finished. If the number 50 was there instead that would not be as clear.

The place where the use of constants makes a dramatic difference is when you are told that there's been a change and the number of parts in a batch is now 75. The number of parts in a batch may be used many times and in many places throughout the program. If you has used the explicit number 50 and were told that the number changed from 50 to 75 you would need to search through your program finding all of the 50s, making sure that the particular 50 related to the number of parts in a batch (as distinct from some other quantity which might happen to also have the value of 50) and make the change. Subtle problems can occur if you missed one of the 50s somewhere. There's also a tendency for 49s and 51s to appear, particularly in loop definitions, and these are easily missed. However if you have a constant named PartsInBatch all you need to change is the constant declaration:

```
const PartsInBatch=75;
```

All of the places that were concerned with the number of parts in a batch are now automatically updated. The places where you might have had 49s or 51s now become PartsInBatch+1 and PartsInBatch-1. It may appear that this is less efficient because of the additional operations taking place to calculate that value however this is not a problem. The compiler is aware that the expression PartsInBatch+1 is composed of values all known at compile time so the compiler itself performs the calculation once creating the 51s and 49s (or 76s and 74s if the PartsInBatch is 75). Using constants allows you to centralize a machine attribute and derive other values from it to manage application information. If a numerical value is used in more than one place for the same meaning use a constant to insure these two values "stay in synch" in the event of a future change.

An important place for constants is in array bound declarations. Arrays are often used to store information about the servo system during data collection. It is very important that an array is not "over-filled". Accordingly whatever procedure is filling the array should use constants to indicate the range of filling which are the same constants used to define the array size. This insures that as the array size is altered that the filling operation is automatically altered as well to insure consistency. Note that symbolic names may be typed into the lower bound and upper bound fields in the variable declaration dialog box allowing this technique to be used with SAW.

Operators

Purpose

Beyond merely shuffling items from one place to another, organization is accomplished by combining and operating on items to produce new items. This section describes common infix operators used in Douloi Pascal.

Operators for simple types

Douloi Pascal supports different infix operators. Infix means that expressions are written in a "conventional math" sense with the operator in between the two parameters it relates to. For example to add two variables and assign them to a third is done with:

```
Sum:=Operand1+Operand2;
```

Spaces are not required between the operator and the operands. A frequently encountered need is to increment a variable by one. In pascal this is usually done with the command:

```
NumberOfPartsCompleted:=NumberOfPartsCompleted+1;
```

Question: Isn't that a lot of overhead for a simple increment? Does Douloi Pascal provide an INC instruction to tell the compiler just to generate an assembly level INC?

Answer: The compiler in Douloi Pascal recognizes that the destination of this operation and the source are the same, and that the amount being added is one. In this case the compiler will directly generate a 386 INC instruction of a size suitable for the integer or longint type.

Infix operations include

+ *Addition operator*

- *Subtraction operator*

*	*Multiplication operator*
div	*integer division*
/	*floating point division, i.e. fractional*
mod	*remainder operator for integer division*
and	*logical and operator*
or	*logical or operator*
xor	*logical exclusive or operator*

Operations may be grouped with parenthesis to explicitly control precedence, i.e.

```
Answer:=(2+(4*6));
```

would be a different answer from:

```
Answer:=((2+4)*6);
```

Douloi Pascal automatically holds multiplication and division at a higher precedence than addition and subtraction however it's best to be explicit.

Douloi Pascal supports compile time evaluation of certain constant expressions. The previous examples would have resulted in only a run-time assignment of the answer into the respective variables.

Operators for aggregate types

A unique attribute of Douloi Pascal is support for infix operations on aggregate types. Aggregates (i.e. objects and records) respond to infix operators by performing the operator on a component by component basis for each component in the structure.

Motion control applications often involve calculations having to do with space. Vectors, and specifically the TNVector types, are provided to represent n dimensional space. Having infix operators between objects means that vector addition and subtraction are directly supported. For example consider:

```
NewPositionVector:=OldPositionVector+DeltaPositionVector;
```

The variables NewPositionVector, OldPositionVector, and DeltaPositionVector can be declared as vectors of any dimension supported. This statement performs vector addition to create a new n dimensional location. Note that all of the variables need to be of the same dimension, i.e. infix operators when used with aggregates only work if the aggregates are exactly the same type.

Note that infix operators on some aggregates may be meaningless, for example adding together the XAxis and YAxis produces a meaningless T1Axis. Also note that vector multiplication is not supported in the "dot" or "cross product" sense, however it is supported in the component-by-component sense.

Procedures and Functions

Purpose

Organizational activities speed up when you are able to put things into bags and manage the enclosed group as a single item. Procedures and Functions provide "bags" which can contain statements organized to accomplish a particular objective. Once organized this behavior can then be requested from many different points in a program to provide a specialized capability. Procedures are like subroutines in basic and "void" functions in C. Functions are procedures which return an answer.

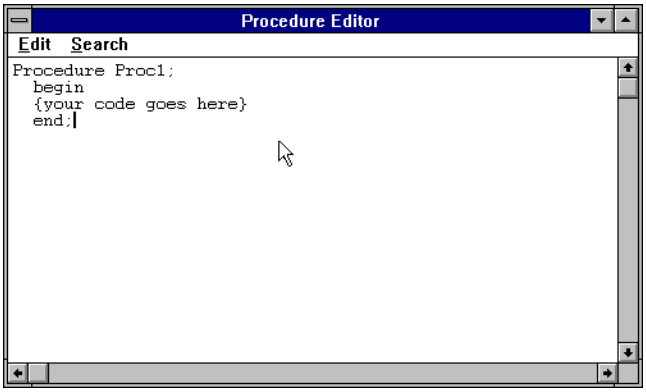
Procedures

The syntax of a procedure without parameters is:

```
procedure <procedureName>;


   <optional local variables>

   begin
   <statements>;
   end;
```

The first word is "procedure", a keyword indicating that a procedure is being defined. Following is a space and the name of the procedure followed by a semicolon. At this point you may choose to include "local variables". Local variables are temporary storage areas which remain intact only for the duration of this particular execution of the subroutine. Following any local variables is "begin" followed by statements followed by "end".

If you are using SAW you are provided with a procedure template when you create new plate procedures. An example template looks like:



The following procedure moves the XY axis in a square:

```
Procedure MakeSquare;
   begin
   XYAxis.MoveBy(1000,0);
   XYAxis.MoveBy(0,1000);
   XYAxis.MoveBy(-1000,0);
   XYAxis.MoveBy(0,-1000);
   end;


Procedure MakeManySquares;
   begin
   XYAxis.MoveTo(0,0);
   MakeSquare;
   XYAxis.MoveTo(2000,2000);
   MakeSquare;
   end;
```

Now whenever you need to make a square you can simply say MakeSquare rather than the individual statements. MakeSquare has accomplished several things. It has “bagged” the four movement instructions so that now the behavior can be accomplished with one statement instead of four. It has also “raised the level of abstraction”. Now instead of describing the problem in terms of movements, the problem can be described in terms of “squares”, a higher abstraction description of a problem. Procedure names are excellent places to convey meaning about program behavior. Take the time to make descriptive and meaningful procedure names.

What if you want to make squares of different sizes? Procedures are able to take parameters. The syntax of a procedure with parameters is:

```
Procedure <ProcedureName> ({var}<ParamName>
:Type{;another...});
```

To include parameters follow the procedure name with a left parenthesis. After the parenthesis comes a description of the parameter. If the first word in the description is "var" the parameter will be accessed using call-by-reference. If "var" is not the first word the parameter will be call-by-value. The distinction between call-by-reference and call-by-value is important and will be discussed in the following section. The name of the parameter follows, a colon after the name and a type after the colon similar to a conventional variable declaration. You may have multiple parameters however you must separate them with semicolons. The right parenthesis completes the description of the parameter list and a semicolon follows just as it had in the original procedure definition. Usually pascal languages allow you to specify multiple parameters of an identical type with commas separating consecutive names. Douloi Pascal requires that you explicitly provide the type of each parameter, even if it is the same as its neighbor.

The previous example can be rewritten to produce squares of various sizes:

```
Procedure MakeSquare(aSideLength:longint);
   begin
   XYAxis.MoveBy(aSideLength,0);
   XYAxis.MoveBy(0,aSideLength);
   XYAxis.MoveBy(-aSideLength,0);
   XYAxis.MoveBy(0,-aSideLength);
   end;


Procedure MakeManySquares;
   begin
   XYAxis.MoveTo(0,0);
   MakeSquare(1000);
   MakeSquare(2000);
   MakeSquare(3000);
   end;
```

The parameter aSideLength is provided in the calling routine with explicit numbers and produces 3 squares of different sizes. The parameter could also have been provided as an expression or function result.

Functions

Functions are similar to procedures with the additional feature of returning an answer. The syntax of a function is similar to a procedure however the keyword is “function” instead of “procedure”, and following the declaration is a colon, type name, and semicolon instead of just the semicolon, i.e.:

```
function <name>{(<parameters>)}:<type>;
```

The result of a calculation can be returned by the function and used by the caller. To return a value perform an assignment statement to the function name inside the body of the function. To use the answer include the function name anywhere you might use an expression.

An example function might square a parameter:

```
function SquareOf(aNumber:longint):longint;
   begin
   SquareOf:=aNumber*aNumber;
   end;


procedure ReportSquares;
   begin
   Prompter.Writeln(‘Square of 4 is ‘,SquareOf(4));
   Prompter.Writeln(‘Square of 8 is ‘,SquareOf(8));
   end;
```

If while inside a function body the function name is used as a source expression, i.e. on the right side of an assignment statement, Douloi Pascal simply interprets the function name as a temporary variable that responds with the value last assigned to it. Douloi Pascal will not perform recursion as would normally be the case with other pascal implementations. Douloi Automation recommends against using the function name in this way since it develops non-standard habits for you. Use a local variable instead.

Do not forget to assign the function value to be something! In general pascal compilers do not check to insure that the value of a function was ever assigned. Douloi Pascal does not support return types of arrays, strings, or aggregate types however you can access and modify such structures from a procedure or function by passing them as call-by-reference, or "var" parameters.

Call-By-Value and Call-By-Reference

Call-by-value and call-by-reference are two different parameter passing models supported by Douloi Pascal. Which model is appropriate to use depends on what you are doing.

Call-by-value is the default parameter passing model. In this case, copies of the parameters are provided to the calling routine which regards the parameter names as temporary variables. The routine can freely read or write to these parameter variables and cause no effect on the variables that were used in the calling routine to fill in the parameters. The only way for an answer to get back to the caller when using call by value is through a function return result. This approach is generally regarded as the safest way to go because there is less of a “connection” between the activities of the function and the information in the caller.

Call-by-reference is a technique where a “pointer” to the information in the caller is given to the function. The “var” keyword indicates that the parameter is being passed as a referenced piece of information in the caller. All of the “referencing and dereferencing” is handled automatically by the compiler. When you change the value of a parameter that is being assessed through call-by-reference, the value of the parameter in the caller changes. It’s as if the single variable in the caller has two names, its original name and the parameter name in the called function. Call-by-reference allows information to travel from the function back to the caller in a way besides a function return result. For example a procedure can have a var parameter and through assignment to that parameter change the value of a variable in the calling routine. In some cases, when you want to return more than one item of information in one call, call-by-reference is the best way to go. Call-by-reference also incurs less stack space than call-by-value for structures which are large. Douloi Pascal creates temporary stack space for all call-by-value parameters

regardless of size. Some language systems default to call-by-reference. Developers who are familiar with this technique may choose to use the VAR keyword to better simulate a parameter passing method they are familiar with.

The general principle for choosing between the two is: If you expect the value of the variable in the calling routine to be changed by the function, use call-be-reference by indication with a “var” keyword.

Question: What happens when a constant expression or function is used to provide a VAR, call-by-reference, parameter? How can an explicit number or function be "pointed to"?

Answer: If an explicit parameter or function is found for a call-by-reference call the compiler creates, behind the scenes, a temporary variables. The compiler then arranges for that constant or function expresssion to be placed into the temporary variable, and then uses a pointer to that temporary variable for the VAR reference pointer. Note that this takes more time to execute than simply passing a value to the routine. For small variables, ie integers, booleans, longints, and single types, it is faster to simply pass the variable by value rather than by reference.

Control Structures

Purpose

In order to organize a solution to a problem you need a way to express a strategy. Control structures allow you to create strategies to solve your application problem.

Control Structure Principles

Douloi Pascal supports several of the familiar Pascal style control structures. Control structures are used to conditionally execute certain instructions or to accomplish repeated execution of certain instructions (iteration) with criteria to determine when the iteration is completed.

Most control structures use the notion of a “conditional expression”. Conditional expressions, when evaluated, will produce either a “true/false” or “yes/no” type of answer. If this condition is true the program behaves one way. If the condition is false it behaves another. The following are examples of conditional expressions:

```
XAxis.ActualPosition > 2000
Input1 = On
NumberOfPartsMade >= NumberOfPartsRequired
```

Conditional expressions often involve the following comparison operators:

```
<   less than
>   greater than
<=  less than or equal to
>=  greater than or equal to
<>  not equal to
=   equal to
```

Douloi Pascal supports the basic logical operators AND, OR, and NOT as well as compound conditional expressions.

```
RequiredPartsCompleted and NoNewBuildsRequested
not XAxis.MoveHasCompleted
UserAbort or SystemProblem
```

The following statement is also legal in a conditional expression:

```
(XAxis.ActualPosition > 2000) and
(YAxis.ActualPosition < 1000)
```

This would need to be broken up into separate statements. A discussion of control structures now follows.

"If" Construct

"If" constructs are used to conditionally execute instructions. The format of an "If" construct is:

```
IF <some conditional expression> then
   <statement>;
```

An example of an "If" construct would be:

```
If XAxis.ActualPosition > 2000 then
   XBeyondLimit:=true;
```

If you have just one statement that needs to be performed you can simply place that statement after the "then". However if you need to execute several statements you need to enclose the entire group inside a begin..end to create a single statement "block". For example:

```
if XAxis.ActualPosition > 2000 then
   begin
   XAxisBeyondLimit:=true;
   SignalAlarm;
   AbortTask(Addr(XMovementTask));
   end;
```

You may be familiar with languages which have an "Endif" to an "If" instruction. Pascal instead uses the idea of performing "one" statement after the IF, and that one statement can be a single compound statement which contains inside many individual statements.

"If" instructions may be nested to allow construction of multiple requirements:

```
if XAxis.ActualPosition > 2000 then
   if YAxis.ActualPosition > 2000 then
      begin
      BothAxisBeyondLimits:=true;
      SignalAlarm;
      AbortTask(Addr(MovementTask));
      end;
```

"If-Else" Construct

"If-Else" constructs are used to perform as exclusive alternatives either one group of instructions or an alternate group of instructions. The syntax of an "If-Else" construct is:

```
If <conditional expression> then
   <statement>
else
   <statement>;
```

If the condition is true the statement following the "then" is performed. If the condition is not true the statement following the "else" is performed. For example:

```
If XAxis.ActualPosition > 2000 then
   XAxisBeyondLimit:=true
else
   XAxisBeyondLimit:=false;
```

In a manner similar to the IF construct if more than one statement needs to execute enclose that group of statements in a begin..end compound statement.

The thing that's hard to remember about if-else constructs is that a semicolon does not follow the statement preceding the "else". The entire construct, from the compilers point of view, is one single statement. Semicolons do not come until the end of the statement. If an unnecessary semicolon is provided before the else the compiler will let you know so you can remove it.

If-Else statements can be nested to produce one-action-from-many type selections, i.e.:

```
if Today=Monday then
   Prompter.Writeln('What happened to the week-
end?')
else if Today=Tuesday then
   Prompter.Writeln('Where is the coffee?')
else if Today=Wednesday then
   Prompter.Writeln('Half way there....')
else if Today=Thursday then
   Prompter.Writeln('We are almost there...')
else if Today=Friday then
   Prompter.Writeln('TGIF!')
else
   Prompter.Writeln('Love that weekend');
```

The Last else "catches" all the cases that did not qualify in the proceeding tests. Note again, that the semicolon does not occur until after the last statement to conclude the entire group.

Douloi Pascal does not provide a "Case" statement. Case statements can be simulated with the illustrated If-Else type constructs.

"For" Loop

The "For" loop is an iterative construct that is most useful when you know how many times you want to do some group of statements. The syntax of the "For" loop is:

```
For <local variable> := <expression> to <expres-
sion> do
   <statement>;
```

The local variable after the "For" keyword will contain successive values during the execution of the loops. The first value it will contain is the value in the first numerical expression after the assignment. After each execution of the following statement this variable will be incremented by one. The iteration continues while the local variables value is less than or equal to the second numeric expression. For example:

```
For DestinationIndex:=1 to 5 do
   XAxis.MoveTo(DestinationIndex*1000);
```

This will cause the XAxis to move to the coordinates 1000,2000,3000,4000 and 5000 in quick succession.

Douloi Pascal does not support DownTo. Iterative loops that have a decrementing loop variable may be created with a “While” or “Repeat” construct.

In Douloi Pascal the criteria for performing the statement is the loop variable value being less than or equal to the upper range numeric value. If a loop is constructed where this is never the case the associated statement will never execute. This detail varies among pascal implementations. Some implementation execute the body of a for loop at least once, regardless of the loop criteria. Douloi Pascal does not necessarily execute the loop body. For example:

```
for scanner:=1 to -1 do
   Prompter.Writeln('This will never write');
```

Setting up explicit numeric bounds such as this would probably never occur. However the following frequently occurs:

```
for scanner:=1 to NumberOfPointsInArray do
   begin
   PlotPointAtVector(DataBuffer[scanner]);
   if scanner > 1 then
      Plotter.DrawLine(
         DataBuffer[scanner-1].X,
         DataBuffer[scanner-1].Y,
         DataBuffer[scanner].X,
         DataBuffer[scanner].Y);
   end;
```

In this case data in an array is being plotted with discrete points as well as lines between data values. Iterating over collections frequently occurs and having to handle an empty collection as a special case is annoying, i.e. if the body of the for loop always executed at least one time you would be obliged to write the previous example as:

```
if NumberOfPointsInArray > 0 then
   begin
   for scanner:=1 to NumberOfPointsInArray do
      begin
      PlotStarAtVector(DataBuffer[scanner]);
      if scanner > 1 then
         Plotter.DrawLine(
         DataBuffer[scanner-1].X,
         DataBuffer[scanner-1].Y,
         DataBuffer[scanner].X,
         DataBuffer[scanner].Y);
      end;
   end;
```

To simplify management of empty collections Douloi Pascal does not require executing the for statement at least once.

The loop variable must be a local variable, i.e. defined inside the procedure where this construct is being used. By being local the variable is "safe" from being changed by some other activity that would confuse the loop operations. The variable must also be an integer or longint since it needs to "increment" from one loop to the next.

Although it is possible to assign a loop variable inside the "for" statement this is considered very bad style since it breaks the abstraction of the code body looping a specific number of times. If you feel you need to assign a value to the loop variable what you really need to do is use one of the following constructs which are specifically designed for more conditional termination.

"While" Loop

A "While" loop is used to perform a group of instructions while a certain condition remains true. The syntax of a "while" statement is:

```
while <expression> do
   statement;
```

The expression is evaluated and if found to be true the statement is executed. This continues until the statement is found to be false. For example:

```
While not MovementFile.EndOfFile do
   begin
   MovementFile.Readln(XCoord,YCoord);
   XYAxis.MoveTo(XCoord,YCoord);
   end;
```

This would check to insure there is information remaining in the MovementFile (a TFile SAW object for accessing the DOS file system). If information is still in the file that information is retrieved and the system moves to that location until the information is all used. This will iterate various numbers of times depending on how many coordinate pairs are in the file. Note also that if there is no information in the file this loop never executes at all, i.e. it handles the empty case.

While loops are best for iteration where the number of times desired may be unknown and perhaps none at all.

There are some cases where you would deliberately want to create an "infinite loop". This is most simply done in Douloi Pascal by saying:

```
while true do
   begin
   <instructions>
   end;
```

True is always true.

There are some synchronizing tasks where the objective is to do nothing at all until a certain event occurs and then to proceed with an operation. For example you might want to turn on a glue gun after the x axis has passed the 10000 count mark:

```
XAxis.MoveTo(0);
XAxis.BeginMoveTo(20000);
while XAxis.ActualPosition < 10000 do
   yield;
TurnOnGlueGun;
```

Note the "yield" instruction inside the otherwise "empty" while statement. For some single-tasking systems, doing nothing is an option but in a multitasking system doing nothing is quite selfish. If a task is simply waiting it is important to yield to other system activities. Yield is a multitasking management procedure that directs a task

to give up execution because the application developer knows that nothing interesting will happen until the next sample. Execution will return to this procedure in the next millisecond so you really are not yielding control very much. However it is extremely important that you are willing to let go. Douloi Pascal supports high frequency multitasking however it is cooperative in nature. A task must not attempt to control the CPU for longer than it's share of a 1 millisecond sample. Forgetting the yield instruction would cause the system to lock. Principles of multitasking will be discussed later at greater length however the point to be made here is that in Douloi Pascal there are usually no empty loops.

"Repeat" Loop

The "repeat" loop has a similar purpose to the while loop, of handling iteration for an indeterminate number of times, however it is organized so as to ask the question at the end rather than the beginning, and it asks the question in the opposite sense. The syntax of a "repeat" loop is:

```
repeat
   <statements>
until <expression>
```

This is the only construct in Douloi Pascal which is guaranteed to execute its body at least one time. Execution "passes by" the repeat keyword and immediately performs the instructions in the repeat body. After doing these instructions a condition is evaluated. If the condition is true execution falls through the until keyword and continues past the block. If the condition is false execution returns to the location indicated by "Repeat" and the loop is repeated. For example:

```
repeat
   XAxis.MoveBy(100);
until XAxis.ActualPosition >= 20000
```

This causes the XAxis to "hop" a small step repeatedly until the absolute position is greater than 20000.

Repeat constructs are unusual in that they are the only construct which permits a group of statements, between the repeat and until keywords, without an enclosing begin..end construct (although

you're welcome to put the begin..end in there if you like). The structure of the construct explicitly delimits the range of instructions so the compiler is able to distinguish the range without any additional help.

For the same reasons as explained for while constructs it is never appropriate to have an empty repeat construct, i.e. the following would be the minimum repeat body:

```
repeat
   yield;
until XAxis.ActualPosition < 20000; {or some other
condition}
```

"Try..Recover"

"Try..Recover" provides structured exception handling. Exception handling has been recently approved by the ANSI committee for C++ and is regarded as a powerful new capability that will simplify program design in the future. Douloi Pascal provides an exception handling mechanism through Try..Recover. Although more involved than other control structures Try..Recover can simplify application design. It is important to understand Try..Recover because many object methods use this structure and may expect applications calling these methods to respond appropriately.

An "exception" or "escape" is a system response to some problem that occurs. For example getting a word from the user when a number was explicitly requested generates an exception. Discovering that a file contains no more information when more information was expected generates an exception. Attempting to control an axis which is currently being controlled by something else generates an exception. In a sense exceptions are problems which actively seek out their solutions. Using Try..Recover involves making arrangements so that exceptions will find their solutions.

The syntax of "Try..Recover" is:

```
Try
   <try statement>
Recover
   <recover statement>
```

In a manner similar to the "if..else" construct you don't want to have a semicolon before the Recover keyword.

During execution the Try keyword is encountered and the program constructs a kind of "safety net" with one end "pegged" to the "Try" keyword and the other end of the net "pegged" to the recover keyword. The statement between the try and recover is then spanned by this safety net. Execution of the try statement begins and is most likely a compound statement with begin...end bracketing a group of statements. If at any time during this group of statements an exception is generated the execution of those statements stops at that point and execution "drops out" onto the safety net. Like the circus performing trapeze artist who misses the bar, lands on the safety net and then rolls to the end of the net to get off, program execution is captured by the net and "rolls" to the recover block end of the net and continues execution with the instructions in the recover block. If no problem was encountered during the try statements there is no need for recovery and the statements in the recover block are ignored.

The following example would illustrate how a try..recover block might be used:

```
Try
   SpeedEditor.Read(aSpeed)
Recover
   Prompter.Writeln('Speed value not a number');
```

In this example the Read method is expecting to get from the editor information compatible with the type of the Read parameter. If the user types in a name instead of a number an exception is generated by the Read method and execution moves to the recover block where an error is displayed.

If there are several statements and different types of exceptions could occur how are the exceptions distinguished? Exception types are identified by unique numbers. When responding to an exception in a recover block it is good practice to look at the number of the exception to gain some understanding as to what problem occurred and what a suitable response might be. The exception number or "escape code" is found with the global function EscapeCode. Typical recover blocks have a set of tests that check if it is this problem or that problem. If the problem is beyond the capabilities of the recover block the block ends by issuing an exception itself in the hopes that a proce-

dure or function that called it may know how to handle the problem. Exceptions "climb up" the procedure invocation ladder looking for a recover block that is willing to handle their problem. This recover block exception is created with the statements Escape(EscapeCode). Escape is a procedure which causes an exception to be generated. The value of the escape is EscapeCode, the currently unhandled error. If no one knows how to handle the problem the exception eventually makes its way back to SAW which has a default recover block that displays the escape code on the screen.

An example of a more advanced try recover block might be the following:

```
try
   begin
   DestinationEditor.Read(aDestination);
   XAxis.BeginMoveTo(aDestination);
   end
recover
   begin
   if EscapeCode=ReadEscapeCode then
      Prompter.Writeln('Please provide a longint')
   else if EscapeCode=MotionOverrunEscapeCode then
      Prompter.Writeln('Impossible move requested')
   else
      Escape(EscapeCode);
   end;
```

In this example there are two statements in the try block. The first statement, the editor read, has the capacity to escape if the user provides a type which is not suitable. The second statement, the x axis movement, has the capacity to escape if while in motion a destination is given to the axis which it cannot perform. The recover block checks for each of these conditions using the symbolic names for escape codes. These names can be found in the on-line help by searching for the word "escape". If the escape was neither of these two cases the recover block escapes with the value of EscapeCode so that the routine which called it can possibly respond. Note that the chain of recourse for an escape is "up the call ladder". An escape will attempt to get an answer from the immediate routine, then the caller, then the caller of that etc. until either it finds an answer or finds the top level default recover block which displays the escape code on the screen. Note that this "ascent" up the call chain is enabled by using Escape(EscapeCode) in the recover block. If you do not issue an

escape for unhandled escape codes you are basically ignoring the problem of an unhandled escape and "sweeping the escape under the carpet". This is not a good practice because an important piece of run-time information relating to something failing is not being acknowledged. Always end recover blocks with a "none-of-the-above" response of Escape(EscapeCode).

Up till now we haven't strongly motivated the reason to have Try..Recover. It seems that there are several simpler alternatives for accomplishing what we are doing with Try..Recover such as using an unconditional jump to an error handler or passing status parameters. The next example makes an important distinction between try..recover and these other techniques. Imagine the following procedures:

```
procedure PerformMotion(Destination:longint);
   begin
   XAxis.BeginMoveTo(Destination);
   end;


procedure PerformManyMoves
   var scanner:longint;
   begin
   for scanner:=1 to 10 do
      PerformMotion(scanner*1000);
   end;


try
   PerformManyMoves
recover
   begin
   if EscapeCode=MotionOverrunEscapeCode then
      Prompter.Writeln('Motion Overrun Occurred')
   else
      Escape(EscapeCode);
   end;
```

The first procedure abbreviates an X axis motion. The second procedure performs many of these motions. The try statement invokes PerformManyMotions however it is possible for the motor to be actively moving to a destination from some previous activity. If the BeginMoveTo which eventually gets called in PerformMotion cannot successfully splice a new destination onto the current motion a MotionOverrunEscapeCode will be generated. PerformManyMoves contains no error management code at all. Never the less, the MotionOverrunEscapeCode will travel "through"

this intermediate procedure and get caught in the recover block of the routine which called PerformManyMoves. This emphasizes the "safety net" model of try..recover. Escapes can traverse through different call levels to find a solution to a problem.

The ability of Try..Recovers to be nested indirectly through procedure calls is a very powerful capability. This allows subordinate routines to attempt recovery operations themselves, handling the day-to-day problems as underling managers. However if major problems come along that they cannot handle they have a way to express that they cannot cope with the problem and request help from their manager.

Exception handling allows a high level procedure to express responsibility for handling a particular class of problem and permits intermediate-manager procedures from having to be concerned at all. If a low level exception is generated execution goes right past the intermediate managers directly to the one that expressed responsibility where it gets handled. This greatly simplifies error management.

Question: Try Recover just looks like a complicated way to handle a "goto". Why don't you just "goto" the recover block if you have a problem?

Answer: For the simplest case this might work however you would encounter at least the following two problems: (1) Imagine that the procedures called by the try statement are used by different routines. How would the procedure know where to go with a "Goto"? You would have to have distinct procedures for every procedure use thwarting the generality procedures bring to programming or pass pointers in some manner, (2) If an exception occurs "several levels down" in procedure calls and you simply performed an unconditional jump, the intermediate stack frames that were created would be left on the stack and program execution would be corrupt by a confused stack. Try..Recover manages the stack frames automatically.

Question: What's the advantage of exception handling over simply an error result? Why not just provide procedures with an additional var parameter that indicates if the operation was successful?

Answer: That method of solutions is a good one to consider for systems which do not have exception handling mechanisms. The disadvantage of the error result approach is that you have to check or at least record the error result for every call made to the routine or you may lose an error indication. This is tedious and programmers are tempted to disregard the information.

This requires that error results get "shipped" back from called procedures explicitly, i.e. the only way to get an error result back from a low level procedure is to pass it back through intermediate procedures as a variable or to record it globally and have all of the procedures in between check the error and leave prematurely if an error exists. This is very tedious to thoroughly implement. Most likely an error will occur and the absence of a check will cause a "manager" procedure to incorrectly continue as if everything is fine resulting in future problems.

One language design question with respect to exception handling which comes up is "Should exceptions be resumable?" i.e. should there be some way, after responding to an escape code in a recover block to resume execution at the point where the escape occurred? Douloi Pascal does not include resumable exceptions. Motion systems in particular tend to have a physical state as well as an informational or program state. If something goes wrong it is usually necessary not to go back to where things failed but rather to go back before that point to some prior physical state to setup another attempt. Rather than reproduce this setup in the recover block it is usually better to take a fresh "full cycle" attempt through an enclosing repeat loop or higher level repeat loop. For example:

```
repeat
   try
      begin
      XAxis.BeginMoveTo(aDestination);
      TroubleEncountered:=false;
      end
   recover
      begin
      TroubleEncountered:=true;
      if EscapeCode=MotionOverrunEscapeCode then
         XAxis.WaitForMoveToFinish {motion stop}
      else
         Escape(EscapeCode);
      end;
until not TroubleEncountered;
```

A discussion of multitasking capabilities will follow later in the document however one point needs to be made now. Once a task is started, even by another task, it is accountable to no one and operates on its own. Escapes generated within a spawned task do not travel back to recover blocks in the task which spawned the escaping task. They will instead travel up to the default handler. Tasks need to take

responsibility for their own errors.

Error handling can be shared by writing a single handler routine which is used by many different tasks, i.e.:

```
function
EscapeProperlyHandled(anEscape:integer):boolean;
   begin
   EscapeProperlyHandled:=true;
   if anEscape=MotionOverrunEscapeCode then
      Prompter.Writeln('Motion Overrun')
   else if
anEscape=EditorExpectingLongintEscapeCode then
      Prompter.Writeln('Expecting longint')
   else
      EscapeProperlyHandled:=false;
   end;


try
   <do task operations>
recover
   if not EscapeProperlyHandled(EscapeCode) then
      Escape(EscapeCode);
```

The routine EscapeProperlyHandled can be used by many different tasks in top level recover blocks to avoid having to copy the recover behavior into each one.

User Defined Types

Purpose

Getting organized can be aided by having containers that are the right size and shape for items you need to have stored. Douloi Pascal supports user defined types for records and objects that permit creating special storage containers for information. This section describes the creation of user defined types.

User Defined Record Types

If you find a need for a special data type that has not been provided as a predefined type you can create your own. When creating user defined type definitions you are moving out more on your own than with general types or predefined types. The following code fragments would be placed into a "Type definition" user symbol from the SAW plate editor. After the leading type definition you would then include variable and array declarations explicitly, i.e. the interactive variable dialog box is not aware of special types you create and accordingly cannot provide you with the type name when creating a variable.

Type definitions can be made for records and objects. Records will be discussed first since they are simpler. The syntax of a record type definition structure is:

```
Type <type name>=record
   <FieldName1>:<FieldType1>;
   <FieldName2>:<Fieldtype2>;
   <FieldName3>:<FieldType3>;
   ...
   end;
```

The type name is the name you will use in future variable declarations to indicate a variable with this structure. The field names are the "compartment" labels that you will use to indicate that particular part of the variable. The fields have types described by their respective type names. For example imagine that a drilling machine was designed to drill holes in various locations to various depths.

A helpful data structure for such a machine might be:

```
Type THole=record
   Location:T2Vector;
   Depth:longint;
   end;
```

Holes have a location and a depth. This is reflected in the structure of this variable. The name THole is used to conform to the convention that type definitions begin with a capital T to help direct the reader in understanding the purpose of the symbol.

A function that might use such a type could be:

```
procedure DrillHole(aHole:THole);
   begin
   XYAxis.MoveToVector(aHole.Location);
   ZAxis.MoveTo(aHole.Depth);
   ZAxis.MoveTo(0);
   end;
```

You can then create an array of holes through an array declaration such as:

```
const NumberOfHoles=20;
var HoleArray:array[1..NumberOfHoles] of THole;
```

You could then have the drill perform all of the holes (assuming they were all initialized to appropriate values) with the routine:

```
procedure DrillAllHoles;
   var HoleScanner:integer;
   begin
   for HoleScanner:=1 to NumberOfHoles do
      DrillHole(HoleArray[scanner]);
   end;
```

The array is composed of THoles. Any particular item in the array is a THole and accordingly can be passed as a parameter to a function expecting to have a THole.

User defined types help elevate the abstraction of a program by allowing the program to manipulate items that are more like items in the actual problem, i.e. a drill is concerned with holes. It is very

convenient to be able to directly discuss holes rather than being constantly concerned with lower level details of holes such as x and y locations and drill depths.

User Defined Object Types

Objects are similar to records and have similar type declarations. The difference between an object and a record is that as well as fields representing information, objects can have fields which represent behaviors, i.e. functions and procedures that are specifically understood by that object. These procedures and functions can access the information in the object as well as other procedures and functions. Procedures and functions may be written directly in the body of the type declaration. The type definition for an object is:

```
Type <ObjectName>=object
   <DataItem1>:<DataItem1Type>;
   <DataItem2>:<DataItem2Type>;

   <Procedure1Name>;
      <procedure body>
   <Procedure2Name>;
      <procedure body>
   <Function1Name:<Function1Type>;
      <Function Body>
   ...
   end;
```

Object programming kindles the imagination because it invites thinking of objects in much more active terms than the physical object might ever portray. For example we can "turn around" the previous hole drilling example by making "active" hole objects instead of having inactive hole records. Consider the following object typed definition for a hole:

```
Type THole=object
   Location:T2Vector;
   Depth:longint;
   procedure Init(XPos:longint; YPos:longint;
HoleDepth:longint);
     begin
     Location.Init(XPos,YPos);
     Depth:=HoleDepth;
     end;
```

```
   procedure Drill;
      begin
      XYAxis.MoveToVector(Location);
      ZAxis.MoveTo(Depth);
      ZAxis.MoveTo(0);
      end;
   end;
```

As well as specifying the previous data items for a hole there is now a drill procedure allowing holes to "drill themselves". (This kind of reasoning comes up frequently in object programming where objects take on active responsibilities).

We could test out this object with the following code:

```
procedure test;
   var aHole:THole;
   begin
   aHole.init(1000,2000,500);
   aHole.Drill;
   end;
```

Drilling through an array of holes now becomes:

```
const NumberOfHoles=20;
var HoleArray:array[1..NumberOfHoles] of THole;
procedure DrillAllHoles;
   var scanner:integer;
   begin
   {some process establishes hole data}
   for scanner:=1 to NumberOfHoles do
      HoleArray[scanner].Drill;
   end;
```

The convenience of objects includes the ability to specify a "standard protocol" for objects being used. For example all of the different TNVectors respond to the Init procedure to conveniently fill in values. The parameter lists for each Init procedure are different because the dimensions of the vectors are different. A TNAxis axis group also has an Init constructor. If you need to know how to initialize an object there is a good chance that the way is to use Init. You may need to look up the parameter list in the on-line help documentation but you'll know where to look.

Using the Math Coprocessor

Purpose

Douloi Pascal supports infix notation for 32 bit single precision and 64 bit double precision numbers that can be calculated on systems without a math coprocessor. However, when an application requires higher trigonometric functions or fast floating point performance the best approach for a real time system is to use a math coprocessor. PC's based on the 486DX have a math coprocessor built in. 386 machines will require the external plug-in math coprocessor to benefit from the following capabilities.

Calculator Model

The math coprocessor is provided to application programs as an RPN (Reverse Polish Notation) calculator that directly maps to the structure of the device. If you've used an HP calculator this approach to calculating is very familiar. The calculator contains a numeric "stack" onto which you "push" parameters. After a sufficient number of parameters have been provided an operation can be performed which often replaces the parameters with a result. This result most often remains on the stack to become a parameter for the next operation. Most RPN calculations are performed "from the inside out" when compared to a conventional algebraic equation.

Douloi Pascal provides routines that push and pop Douloi Pascal numeric types. Once these types are on the "stack" operations can then be performed benefitting from the full 80 bit real format used by the math coprocessor. The math coprocessor is a shared resource of the computer. The most considerate technique for using the math processor requires saving its current state to a buffer, initializaing it (to have a fresh start), performing your calculations, saving the answer, and restoring the coprocessor to its previous condition, all in one sample. This allows calculations being performed by other programs to complete correctly. The "template" for this process is shown as follows:

```
procedure CoprocessorExampleUse;
   var Buffer:TMathCoprocessorBuffer;
   begin
   FSave(Buffer);
   FInit;
   {...perform calculations...}
   FRestore(Buffer);
   end;
```

Calculation Procedures and Functions

The following list briefly describes procedures and functions used with the math coprocessor. Additional information can be found in the on-line help. In general the procedures map directly to math coprocessor instructions. The notes use a "Forth" style comment notation indicating the "before" condition of the stack on the left side of the dash and the "after" condition of the stack after the operation.

The following procedure should be used when first starting the application to put the coprocessor into a known state.

```
procedure FInit; {---}
```

Typically this would be in the setup procedure for the main windows although it may also be in the procedure that uses the coprocessor as shown in the examples following.

The following routines are used to put types found in Douloi Pascal onto the top of the math coprocessor stack:

```
function PushLongint(aLong:longint);      ( --- n )
function PushSingle(aSingle:single);      ( --- n )
function PushDouble(aDouble:double);      ( --- n )
```

The following procedures perform operations on the parameters currently pushed on the stack and leave results on the top of the stack.

```
procedure FAdd;                    (a b --- a+b)
procedure FSub;                    (a b --- a-b)
procedure FMul;                    (a b --- a*b)
procedure FDiv;                    (a b --- a/b)
procedure FSqrt;                   (a --- sqrt(a))
```

The following trigonometry routines are available. The parameters are in units of radians.

```
procedure FSin;              (a --- sin(a))
procedure FSinCos;           {a --- sin(a) cos(a))
```

The following numbers are often used and these routines provide short cuts for loading them into the calculator.

```
procedure FLdPi; {Load Pi}               (--- pi)
procedure FLdZ;  {Load Zer}              {---  0}
procedure FLd1;  {Load One}              {---  1}
```

The following routines are available for moving results from the math coprocessor back into normal Douloi Pascal variables.

```
function PopLongint:longint;        { n --- }
function PopSingle:single;          { n --- }
function PopDouble:double;          { n --- }
```

Math Coprocessor Examples

Adding Two Numbers

The following routine, implemented as a button click procedure, would be an example (overkill though it may be) of adding together two numbers.

```
procedure Click;
   var Buffer:TMathCoprocessorBuffer;
   var Answer:longint;
   begin
   FSave(Buffer);
   FInit;
   PushLongint(2);
   PushLongint(3);
   FAdd;
   Answer:=PopLongint;
   FRestore(Buffer);
   Prompter.Writeln(Answer);
   {you should see the value 5}
   end;
```

Calculating The Sin of a Number

The following routine calculates the sin of a number provided in degress. The number is read from an editor named NumberEditor and the result is placed into a text object named AnswerDisplay as well as returned in the function. Note that the FSave occurs after the readln (which involves task yielding) and the restore is performed before the writeln (which also involves yielding). The setup, calculation, and retreival of the answer are all performed in the same sample.

```
Function GetSin:Single;

   var UserNumberInDegrees:Single;
   var Answer:Single;
   var Buffer:TMathCoprocessorBuffer;

   begin
   NumberEditor.Readln(UserNumberInDegrees);
   FSave(Buffer);
   FInit;
   PushSingle(UserNumberInDegrees);
   FSin;
   Answer:=PopSingleReal;
   FRestore(Buffer);
   AnswerDisplay.Writeln(Answer);
   GetSin:=Answer;
   end;
```

Multitasking System

Purpose

Getting organized can be simplified by having more than just one person doing the job. By having a group of participants, each covering a particular responsibility, the total task is accomplished with each sub-task more easily expressed than a description of the entire task as a single unit. Motion Server provides support for high frequency multitasking to provide this kind of advantage to a motion control application program. This section describes the principles of multitasking for this system, the particular procedures used to manage tasks, and usage techniques.

Multitasking Model

Multitasking can be designed into a system such as Motion Server in different ways depending on the system's design objectives. The driving criteria for Motion Server is suitability for motion control. Although general purpose multitasking has a potential role to play in any program, the main benefit of multitasking in a motion control system is creating custom axis coordination. We would like to create programs which can prescribe specialized criteria for a motor position that has as much resolution and performance as the built-in trapezoidal profiler, i.e. we want to be able to create at the application level, through a multitasking mechanism, custom profiles to perform activities such as electronic gearing, cam trajectories, etc. that can operate while other application procedures are also operating.

This implies that the multitasking system needs to be able to run a commanded position setpoint program at the rate of the built-in profiler, 1 KHz. To properly understand application responsibilities involved in using the multitasking system some details of the system will now be discussed.

The multitasking system is part of the 1 KHz sample rate interrupt. Every millisecond the 486 is interrupted from the job of executing Windows and performs the interrupt handler. The interrupt handler performs the following functions:

Position Maintenance	*for up to 16 axis*
Motor Control Laws	*for up to 16 axis*
Profile Management	*for up to 16 profiles*

Conventional Tasks *for up to 12 tasks*

These interrupt handler components are now discussed.

Position Maintenance

The motion system is able to describe motion over a range of +/- 4 billion. The actual hardware only monitors position changes over a range of 64K counts. The Position maintenance section accumulates these smaller position deltas to maintain the absolute position of each of the axis.

Motor Control Laws

A PID control law is performed for each axis which has the servo active. The corresponding motor command for that axis is then set to reflect the results of the calculation.

Profile Management

As many as six axis of motion can be independently active (i.e. six separate single axis moves). The calculations for these profiles are done in this region of the interrupt handler so as to calculate the commanded setpoint that will be used in the next sample period.

Conventional Tasks

Six conventional tasks can be arranged to run each sample although it is much more common for only a few tasks to operate every sample while other tasks are scheduled to execute at much lower frequencies. These tasks are what you create when you fill in “click” procedures for buttons or event procedures for plates.

Last Task

You may optionally have a “last” task. This is most often a safety/limit switch task which gets a “last shot” at execution after the profiler and conventional tasks have executed. This task has the opportunity to super-seded the commanded positions that were established by the motion profiler or the conventional tasks making it the best place to manage limit switch information.

Cooperative Multitasking

Note that the tasks have the opportunity to run every millisecond and that they are part of the interrupt handler and are not in the "foreground". Windows is in the foreground. The tasks must execute "cooperatively" in the interrupt handler region in the same way that Windows applications need to "cooperate" in the foreground region. It is bad for a single task to "take over" the entire time slice, or worse, to attempt to take over the entire machine. By doing so the task overruns its fair share of the interrupt time, possibly the sample period, and possibly produces a reentrancy to the handler if it lasts too long. Although the handler can survive a limited number of reentrancies this clearly is not a steady state permissible situation and eventually the system will lock. As was mentioned in the language section on looping, it is important for tasks to "yield" to the system and not perform empty loops waiting for a system event to occur. The routines TaskOverranSample, CheckTask, and TimeRemaining can be used to determine if a task is taking too much time.

As unfortunate as a locked system may appear to be during development because of a programming mistake, in practice it has not been that inconvenient. Douloi Automation has never lost saved information or system integrity when performing a soft or hard reset from the locked condition created by this effect. However any work on an application that was performed but not saved will be lost. It is good practice to save your work before attempting to run an application.

If a locked system seems to be a severe consequence for a programming error bear in mind that although Motion Server and SAW simplifies access to a real-time multitasking motion control system, you never the less are still working with one. For the privilege of having real performance you must take the risk of possibly having real problems. Douloi Automation is available to work with you to understand application development problems. Don't hesitate to call. Save your work frequently.

Windows Mail

Many responsibilities performed by a program can be accomplished with the resources available right within a task. Motion profiling, array management and data collection, computations, etc. are performed entirely by the real time tasks themselves. However there are times when tasks need to interact with Windows and/or with the operating system, for example to display information on a graph or write information to a file.

The real time tasks do not have the authority to directly access these resources themselves. You may have heard the statement "DOS is not re-

entrant". This is the case. Tasks cannot access the operating system directly because they are often performing their work while in the midst of interrupting DOS already which would break this rule. In order to relate to Windows and DOS, tasks "send mail" to SAW requesting services on behalf of the tasks. In a manner analogous to the BIOS routines, a certain template of information is filled out by the task and "mailed". The task waits until it receives a response to the "letter" it sent. SAW, in its good time, subject to the multitasking state occurring in Windows, gets around to discovering mail has been sent. SAW then reads the letter to understand what service is needed and what data relates to that service, performs the service, and then sends mail back indicating that the service has been accomplished. The task is then free to continue executing.

The fact that a "mail" operation is being used to accomplish a certain task is completely hidden from you. You simply use a method, such as Writeln, to display some information. Internally that method results in mail.

It is important to note that task execution blocks while waiting for the service to be performed. What this means is that any operation that involves interacting with Windows or DOS is not real time. There is no way of knowing how long it will take SAW to get a chance to execute with the Windows cooperative multitasking mechanism. A "paint" operation on a graph can hold up the delivery of mail for several seconds.

Although this may appear to restrict real time usefulness this limitation can be avoided with the simple principle: *If it's time critical, don't send mail.*

How do you know if you're sending mail if the mail mechanism is concealed from the developer? The general principle is: if it produces any visible effects, or it relates in any way to the operating system, then it's being performed through mail.

It should always be possible to "factor" a task into two tasks, one "real-time" task and another "support" task so as to accomplish unhindered time critical activity. Consider as an example the following procedure which moves an axis while displaying the position of the axis during motion:

```
procedure MoveAxisAndShowPosition;
   begin
   while true do
      begin
      XAxis.BeginMoveTo(20000);
      repeat

PositionDisplay.Writeln(XAxis.ActualPosition);
      until XAxis.MoveIsFinished;
      XAxis.BeginMoveTo(0);
      repeat
```

```
      PositionDisplay.Writeln(XAxis.ActualPosition);
            until XAxis.MoveIsFinished;
            end;
         end;
```

This type of approach would be necessary if multitasking was not available. While waiting for the move to finish you maintain control panel operation while polling to determine if another move should be started. The problem with this approach is that there is no way to know how long `PositionDisplay.Writeln` will take. If another Windows application takes control and doesn't let go it may be several seconds. This means that there may be an unnecessary delay between the completion of one move and the start of the next. An additional problem with this approach is that writeln is being performed far too many times. A screen refresh rate of 5 Hz is just fine, but this will attempt to constantly update the display. A delay instruction could be included to reduce the flicker of the display, but this introduces a constantly present, unneeded delay between moves. To avoid that delay you could write a delay routine that pre-maturely exited if the polling condition was found to be true. This solution approach just seem to be getting worse.

A "two task" approach which simplifies things all around is:

```
      procedure ShowPosition;
         begin
         PositionDisplay.Writeln(XAxis.ActualPosition);
         end;

      procedure MoveAxis;
         begin
         ScheduleTask(Addr(ShowPosition),200);
         while true do
            begin
            XAxis.MoveTo(20000);
            XAxis.MoveTo(0);
            end;
         end;
```

These cleanly separates the original task into two separate tasks. One task is responsible for displaying information, and the other for performing the motion. Each is quite simple. The motion task does not send any mail and so will never be blocked. There will be no delays in the movement regardless of what Windows does. The other task may get blocked while its performing PositionDisplay.Writeln but we don't care. All that means is that the position information displayed is a little out of date. It will correct itself as soon as the offending Windows application lets go and the screen refreshes. Also note that the 5 Hz PositionDisplay update rate is provided quite easily through ScheduleTask (200 is the specified number of milliseconds in between each invocation of ShowPosition).

This illustrates one of the benefits of multitasking. By separating out the movement responsibility from the display responsibility the tasks become decoupled and each is simpler. The familiar saying "the sum of the whole is greater than the sum of the parts" might be restated as "the complexity of the whole is greater than the complexity of the sum of the parts". Decomposition is almost always a good idea.

Multitasker Commands

The following commands manage the operation of the multitasking system. Details of command syntax and usage may be found in the SAW on-line help.

`BeginTask.....`	Causes a specified task to begin execution
`SuspendTask.....`	Causes a specified task to hold off execution
`ResumeTask.....`	Causes a specified suspended task to continue executing
`ScheduleTask.....`	Causes task to periodically begin execution
`AbortTask.....`	Causes a specified task to terminate
`Yield.....`	Causes a task to wait until next sample.
`Delay.....`	Causes task to wait <n> of milliseconds
`TaskOverranSample..`	Returns true if the task took too much time
`CheckTask.....`	Escapes if a task took too much time.
`TimeRemaining.....`	Returns microseconds left in sample period.
`TaskPresent.....`	Returns true if the task is present
`TaskAddr.....`	Returns a task address given the task name

Techniques

"Saturating" limit switch routine.

The Last Task is provided primarily to allow the construction of "saturating" limit switch routines. For example, one possible limit switch behavior is to have an axis stop its movement when a limit switch is encountered while allowing all of the other axis to continue their operation as if there was no problem. This is often seen on plotters when a drawing is off the page. When the drawing comes back onto the page the stopped axis resumes operation.

To create this behavior a plate procedure can be written to detect when a limit switch asserts and to record the boundary position for the axis at that point. On subsequent cycles the commanded position for that axis may attempt to advance the axis. The commanded position might be implicitly set through a profiled moved, or explicitly set through the use of TNAxis.SetCommandedPosition in an electronic gearing task, for example. In any case, the last task gets to see the commanded position after every other agent has had an opportunity to change it. While the switch is as-

serted and the commanded position for that axis is greater than the boundary position the "saturating" limit switch routine changes the commanded position to the boundary position. When the commanded position comes back into a normal operating range the limit switch routine simply lets the value pass through unaffected. This allows the saturating behavior to work regardless of how the commanded position is accomplished.

Task synchronization

Suitable decomposition of a problem into separate tasks can simplify a motion application. There are times, however, when it is important for one task to wait until another task has finished. The management of having one task wait for another can be referred to as task synchronization.

Synchronization Approach 1, Shared Variables

A simple approach to synchronization is shared variable flags. In this technique a single boolean variable is used by two different tasks. The variable is given an initial state by the first task, the second task is started, and the first task waits until the second task changes the value indicating that the second task has finished.

As an example consider the following two tasks. The first task, named Operation1, begins Operation2 which performs some other activities and then waits for Operation2 to finish:

```
procedure Operation1;
   begin
   Operation2IsFinished:=false;
   BeginTask(TaskAddr(Operation2));
   {do some activities}
   repeat
      yield;
   until Operation2IsFinished;
   PrompterWriteln('Both Operations Done');
   end;

procedure Operation2;
   begin
   {do operation 2 activities}
   Operation2HasFinished:=true;
   end;
```

The variable Operation2IsFinished is a boolean plate variable which can be accessed by both tasks. Operation1 sets the variable to be "false" (i.e. Operation2 has not finished, in fact it has not even started yet...) and then begins Operation2 as a separate task. Operation1 continues executing instructions that follow. When all of the remaining instructions are com-

pleted Operation1 waits for Operation2 to finished by looping (remember the yield inside an otherwise empty loop) until the variable Operation2IsFinished is set to true. If this was a single tasking system this would wait forever since nothing in the loop changes the value of Operation2HasFinished. In a multitasking system there are other agents active, however. Operation2, upon completion, sets the variable to true signalling to anyone who is concerned (including Operation1) that Operation2 has finished.

Note that Operation1 takes responsibility for initially setting the variable Operation2HasFinished to false. It may seem like a good idea to have Operation2 set the variable to false when it first starts and set it to true at the end. However there are situations where this might not work correctly. Even though Operation1 performs a BeginTask to get Operation2 started, no other task actually does anything until Operation1 yields. The behavior of BeginTask is really to setup the task to run when given an opportunity to do so. If there are no yields between the time that BeginTask is given and the check is made in the repeat loop Operation2 would never have had a chance to initialize the variable to the correct value, false, and this would fail to work correctly. This is an example of a “race” condition when performing task synchronization. By having Operation1 set the boolean it is guaranteed to hold off Operation1 and the particular timing of each task no longer matters. In general, the task that will regard the flag information should perform the initialization so as to not outpace the initialization otherwise done in some other task.

Synchronization Approach 2, Task Status

Another way to have Operation1 wait for Operation2 to finish is to use the TaskPresent function. This approach would look like:

```
procedure Operation1;
   begin
   BeginTask(TaskAddr(Operation2));
   {do some activities}
   repeat
      yield;
   until Not TaskPresent(TaskAddr(Operation2));
   PrompterWriteln('Both Operations Done');
   end;

procedure Operation2;
   begin
   {do operation 2 activities}
   end;
```

Operation2 becomes present as soon as it is setup for execution. TaskPresent will report that Operation2 is present after being setup with BeginTask even if it has not had an opportunity to run yet. This eliminates the need for the intermediate variable by directly accessing the task state.

Synchronization Approach 3, Don't multitask

Another approach is to recognize that you may be over-using multitasking and to simply not multitask. If Operation1 really doesn't have anything important to do until Operation2 is finished it is much simpler to just call Operation2. Subroutine calls suspend the calling operation until the called function has had a chance to finish. The example then becomes:

```
procedure Operation1;
   begin
   Operation2;
   PrompterWriteln('Both Operations Done');
   end;

procedure Operation2;
   begin
   {do operation 2 activities}
   end;
```

Multitasking is useful but makes matters complicated when used unnecessarily. Don't forget that conventional procedure calls are available and by far represent the simplest way of keeping a set of statements in a particular order if you are willing to suspend the calling task.

Program Formatting

Purpose

The ease with which a program can be read and understood is dramatically influenced by the program's formatting. There are many formatting guidelines. This note presents one particular style which is used in Douloi Pascal examples and software catalog components.

Principles

Douloi Pascal allows spaces and lines between different statements, and even between words within a statement. The main principle in program formatting is to create a spatial representation of the program structure and particularly of statement groups or "blocks".

The main formatting principle being shown here is to indent the text whenever a new group of statements represents a new level of abstraction. The reader of the program can "modulate" the level of detail by changing how far to the right he starts reading text.

For example the most abstract part of a procedure is its name. It may be (particularly if the name is well chosen) that all you need to know about a procedure is the name. Accordingly the procedure declaration is furthest to the left and everything else about the procedure will be indented to the right. When reading, the thought that might go through the reader's mind might be:

```
procedure DrillHoles;
   {additional details I don't care about yet}
```

When reading the text your eye catches "procedure" and finds a procedure declaration. Since everything else about the procedure declaration elaborates what the name says; it will be indented to the right. Suppose you want to learn more about how the procedure works. Then you read in the next level of indentation.

```
procedure DrillHoles;

   var RowScanner:integer;
   var ColumnScanner:integer;
   const RowSpacing=100;
```

```
   const ColumnSpacing=100;
   const NumberOfColumns=10;
   const NumberOfRows=10;

   begin
   for ColumnScanner:=1 to NumberOfColumns do
    {additional details I don't care about yet}
   end;
```

The next items we are concerned with are the local variables that help describe the problem. This procedure apparently is concerned about rows and columns. It does some operation as many times as there are columns. What happens for each column? Read over another indentation level and read additional details:

```
procedure DrillHoles;

   var RowScanner:integer;
   var ColumnScanner:integer;
   const RowSpacing=100;
   const ColumnSpacing=100;
   const NumberOfColumns=10;
   const NumberOfRows=10;

   begin
   for ColumnScanner:=1 to NumberOfColumns do
    for RowScanner:=1 to NumberOfRows do
      {additional details I don't care about yet}
   end;
```

Apparently every column the procedure works through all of the rows and does something. This means that every row and column intersection will be visited and something done there. What happens at every row and column intersection? Read to the right for further details:

```
procedure DrillHoles;

   var RowScanner:integer;
   var ColumnScanner:integer;
   const RowSpacing=100;
   const ColumnSpacing=100;
   const NumberOfColumns=10;
   const NumberOfRows=10;
```

```
begin
for ColumnScanner:=1 to NumberOfColumns do
 for RowScanner:=1 to NumberOfRows
      DrillHole(RowScanner-1*RowSpacing,
      ColumnScanner-1*ColumnSpacing);
end;
```

By reading further to the right we discover that every intersection gets a hole. Note that the level of detail usually includes the "nesting" level of the construct also. Each nexted construct should be on its own level. The end of the level helps indicate the end of the construct. Consider the following conditional case:

```
function
RequiredPostageForWeight(PackageWeight:single):single;
   begin
   if PackageWeight >= 4 then
      Escape(PackageTooHeavyEscapeCode)
   else if PackageWeight >= 3 then
      RequiredPostageForWeight:=0.98
   else if PackageWeight >= 2 then
      RequiredPostageForWeight:=0.75
   else if PackageWeight >= 1 then
      RequiredPostageForWeight:=0.52
   else
      RequiredPostageForWeight:=0.29;
   end;
```

Suppose you know that the problem with this procedure occurs when the weight is greater than 2. Each conditional expression can be examined and if it is not true your eye can skip down to the next place where the level of indentation is the same. The block of instructions related to the conditional is indented in; and regarded as a greater level of detail that you are concerned about. In this particular case you want to know what happens when the condition is true.

Nested indentations are particularly helpful when you are examining nested conditions. Consider the following example:

```
procedure SetupCamPattern(NumberOfPoints:longint);
   begin
   if (NumberOfPoints mod 2) = 0 then
      begin
      if NumberOfPoints > 10 then
         LoadShallowCam(NumberOfPoints)
      else if NumberOfPoints > 4 then
         LoadMediumCam(NumberOfPoints)
      else
         LoadDeepCam(NumberOfPoints);
      end
   else
      Prompter.Writeln('Even Number Required');
   end;
```

Here we read "if the number is even, do something, otherwise explain that an even number is required." To learn what happens when the number is even you read another indentation level in. If you don't care what happens when the number is even, your eye can just drop down to the same indentation level of the if, to find the corresponding else statement that goes with it. Keywords of a particular structure are found at the same indentation level.

Summary

The formatting method described can help a program be more readable to others and even to yourself at a later time. The time required to format a program is saved many times over by the efficiency with which you can study and alter the program. The formatting style presented here is not represented as being "best" or "standard" but simply serves as an example style that you may encounter in example programs and software catalog components.

Gotchas

Purpose

Any language or tool has “gotchas”, problems that are characteristic of the tool, typical misunderstandings, or surprises intrinsic to the structure, that can mislead and confuse a developer. The following list may help you avoid some standard pitfalls that often nab pascal programmers in general or Motion Server/SAW programmers specifically.

Statements Apparently Fail to Execute

What do you suppose the following loop does?

```
...
while not aFile.EndOfFile do
   scanner:=scanner+1
   aFile.Readln(XCoord,YCoord);
   XYAxis.MoveTo(XCoord,YCoord);
Prompter.Writeln(‘done’);
...
```

It may look like it reads a series of point pairs from a file, performs moves to those coordinates, and then writes ‘done’ on a display. However what actually happens is nothing at all, except a locked system. Why? Without a begin..end compound statement around the indented group of statements the compiler regards the while loop as:

```
While not aFile.EndOfFile do
   scanner:=scanner+1
```

The while statement performs the statement which follows, in this case an increment. Because that statement never changes the state of aFile, the condition never is accomplished (problem 1) and the task never yields (problem 2). Because the loop never terminates the task overruns multiple samples and the system locks up. The correction to this problem (after performing a hard reset) is to enclose the instruction which are inside the loop with a begin..end:

```
while not aFile.EndOfFile do
   begin
   scanner:=scanner+1
   aFile.Readln(XCoord,YCoord);
   XYAxis.MoveTo(XCoord,YCoord);
   end;
Display.Writeln('done');
```

Remember that indentation, although good formatting practice, is not recognized by the compiler. Begin..ends are recognized and must be included to groups multiple statements into one statement for conditional jumps and iteration.

Unexpected Escape During File Reads

A file has been written which contains on each line a coordinate pair. The following loop reads the file but apparently attempts to perform an extra read producing a read escape despite the loop terminating on EndOfFile:

```
...
while not aFile.EndOfFile do
   begin
   aFile.Read(XCoord,YCoord);
   XYAxis.MoveTo(XCoord,YCoord);
   end;
aFile.Close;
.....
```

What's the problem here? The problem is the difference between Read and Readln. Read takes from the file enough characters to get the next variable, in this case 2 variables, one for each coordinate. That seems correct. However read does not remove the "new line" character at the end of the line. This won't be removed until the next read which will remove it in its search for the next number. On the very last line, after the very last number pair has been read, there still remains in the file one last new line. This new line, although containing no information, is still not the end of the file so EndOfFile returns false at this point. The loop then attempts to read the next number pair, removes the new line and discovers there's nothing left creating an escape event because a number was expected. The fix is to use Readln instead of Read. Readln removes the new line after getting everything it collects. This results in the file being empty and the loop terminating correctly.

A simple principle to help remember is to use Readln if the file was constructed with Writeln, and Read if the file was constructed with Write. When you have opposite combinations bugs like this can appear.

Information Being Collected Does Not Change

The following fragment is supposed to collect commanded torque values in an array to be used for plotting in another section of the application. TorqueDataLength is the upper bound of an array named TorqueData.

```
...
for scanner:=1 to TorqueDataLength do
   TorqueData[scanner]:=XAxis.CommandedTorque;
...
```

When the toque data is plotted, however, it's just a straight line. The entire array contains just one value.

What's the problem here? The problem is that we forgot to wait for new information. Programs have an opportunity to run every sample period. The program containing this sample code was started on a particular sample and because there was no instruction to yield control the entire program, including the filling of the entire data array, occurred in one sample. Interestingly enough for an array size of 100 the system did not have a problem with filling the array in one sample period so there was no overrun problem with a much longer task than the writer intended. However the torque value is only changed by calculations which occur every sample period. Within a sample the value does not change. Without a request to release control until the next sample period the loop simply collects the same value of XAxis.CommandedTorque many times. The correct and most likely intended implementation of this loop would be:

```
for scanner:=1 to TorqueDataSize do
   begin
   TorqueData[scanner]:=XAxis.CommandedTorque;
   Yield;
   end;
```

By including a "yield" instruction we are indicating that we want to wait until the next sample period before continuing with this program. Waiting allows the control law calculations to provide a new value so that the TorqueData will really represent the time history of torques over many samples.

Program Locks While Waiting for Motion To Finish

The following section of code might appear in an initialization procedure. The idea is to have the motor back up in the negative direction until a homing switch is activated. The motor should then stop:

```
...
XAxis.Jog(-40);
repeat
until InputLong(1) and Input1OnMask > 0
XAxis.Abort;
```

However when this procedure runs the computer locks up. Why does this occur?

Tasks must release control every sample and not attempt to execute programs greater than their sub-millisecond time slice. The repeat...until loop retains control for a period that might well be several seconds while the motor is looking for the home position. The correction to this problem is to include a yield instruction inside the repeat...until loop. The correct code fragment is shown below.

```
...
XAxis.Jog(-40);
repeat
  yield;
until InputLong(1) and Input1OnMask > 0
XAxis.Abort;
```

Incorrect Branching When Using Masked Inputs

The following code section is written to perform a certain procedure if Input 4 is high however the branch is never taken:

```
if InputLong(1) and Input4OnMask = On then
  PerformOperation;
```

The problem here is that On is a boolean constant with the value 1. Input4OnMask has a value of $0008 hex. The result of the "and" expression will either be 0 or $0008 but never 1 so the branch will never be taken. There are two "style" solutions to this type of misunderstanding. One approach is to use the "on" mask instead of on, ie:

```
If InputLong(1) and Input4OnMask = Input4OnMask
then
  PerformOperation;
```

An alternative style is to check for anything positive, i.e.:

```
If InputLong(1) and Input4OnMask > 0 then
  PerformOperation;
```

The advantage of this second style is that the input bit number, in the Input4OnMask symbol, only occurs once. If the input was to change to another bit you would only have to make one edit. It would be possible to forget the second occurance of the bit number in the positive value with the first style. Both approaches generate the same amount of code and take the same amount of time.

Subplate Does Not Appear When Application Starts

You've constructed an application that uses a sub-plate on which you intend to draw lines and placed into one side of the main window. However when the application starts the sub-plate is not there. Why didn't the sub-plate get created along with the application?

Sub-plates can have different releationships to their "parent" window. Sub-plates can be "merged", basically adding their elements to the set already on the parent plate, "attached", where they are created as distinct windows and get constructed when the parent gets constructed, or as "popup" windows where they get constructed only after receiving a "PopUp" instruction. Most likely, the plate style is "popup" and the plate is waiting for a PopUp instruction to display itself. What was probably intended was "attached". Double click on the plate and use the Next and Previous keys underneath the display of the current style until you find an attached plate with the desired border and try again.

Drawn Lines Do Not Appear On Plate 1

The following code fragment does not draw the desired line:

```
DrawLine(0,0,100,100);
{expecting to see line but line isn't there}
```

What's wrong here? In order to reduce screen flashing, a plate does not update its appearance until you tell it to with Update. After adding all the new geometry use Update:

```
DrawLine(0,0,100,100);
Update;
```

Drawn Lines Do Not Appear On Plate 2

The following object definition describes a data structure that contains 2 vectors. The object has been given a procedure so as to draw itself on a plate. However when used the expected line does not appear.

```
type GuidanceLine=object
   End1:T2Vector;
   End2:T2Vector;
   procedure DrawOn(aPlate:TPlate);
      begin
      DrawLine(End1.X,End1.Y,End2.X,End2.Y);
      end;
   end;
```

What's wrong? We forgot to use the parameter to DrawOn! The idea is to put the line on the specified plate. Why didn't we get a compiler error? If no receiver is specified the object definition uses the nearest plate as the receiver, in this case the plate that contains this object definition, which is visible within the current scope. The correct object definition is shown below.

```
type GuidanceLine=object
   End1:T2Vector;
   End2:T2Vector;
   procedure DrawOn(aPlate:TPlate);
      begin
      aPlate.DrawLine(End1.X,End1.Y,End2.X,End2.Y);
      end;
   end;
```

Runtime Error 104

The following section of code correctly plays back motion by reading coordinate pairs from a file. However after working correctly several times a "Runtime error 104" is generated.

```
procedure PlaybackMotion;

   var MotionFile:TFile;
   var Destination:T2Vector;

   begin
   MotionFile.Assign('MoveFile.txt');
   MotionFile.Reset;
   While not MotionFile.EndOfFile do
      begin

MotionFile.Readln(Destination.X,Destination.Y);
      XYAxis.MoveToVector(DestinationVector);
      end;
   end;
```

What's the problem here? Assigning a file requests from the operating system a "file handle" which is used to represent the file. Closing a file releases the handle back to the operating system to be recycled and used again. Because this procedure never closes MotionFile the operating system eventually runs out of file handles and produces the run time error. The correction is to close MotionFile before leaving the routine:

```
procedure PlaybackMotion;

   var MotionFile:TFile;
   var Destination:T2Vector;

   begin
   MotionFile.Assign('MoveFile.txt');
   MotionFile.Reset;
   While not MotionFile.EndOfFile do
      begin

MotionFile.Readln(Destination.X,Destination.Y);
      XYAxis.MoveToVector(DestinationVector);
      end;
   MotionFile.Close; {file handle now available for
use again}
   end;
```

Nothing Happens When a DLL Call Is Made

The following Turbo Pascal for Windows routine is trying to turn on the XAxis servo but nothing happens:

```
procedure SetupMachine;

   var EscapeCode:integer;

   begin
   TNAxisSetServo(XAxis,1,EscapeCode);
   ...
```

What's wrong here? Remember that the EscapeCode parameter is both an input as well as an output to the DLL routines. An EscapeCode value which is not 0 instructs the DLL to not do anything because a problem has occurred. This is corrected by setting EscapeCode to 0 before the call:

```
procedure SetupMachine;

   var EscapeCode:integer;

   begin
   EscapeCode:=0;
   TNAxisSetServo(XAxis,1,EscapeCode);
   ...
```

Chapter 7

7) Predefined Types

Purpose

SAW comes with some predefined object types to help simplify the creation of applications. The general nature of these object types are described here. Details of the objects and their use is found in the on-line Help reference manual.

Reading and Writing Conventions

Many objects which interact with ASCII information respond to Read, Readln, ReadLongint, ReadlnLongint, Write, and Writeln methods. These methods take a variable number of parameters of various types making them much more versatile than conventional fixed parameter, fixed type interfaces. The particular way in which an object interprets reading and writing information varies depending on the nature of the object. TFiles write by placing the information into a DOS file. ListBoxes write by adding the information to the list. Text objects write by changing the currently displayed text to an ASCII representation of the information. This uniform ASCII interface to predefined objects simplifies programming.

In general IO error management is accomplished through the exception mechanism. If an object is not able to read a ReadEscapeCode will be generated. This most often occurs when a type conversion problem exists, for example reading a value for the machine speed but having a name in the editor rather than a number. If writing is not possible, for example a file is not open for writing, a WriteEscapceCode will occur.

TNVector

Description

TNVector is a generic term for the following predefined vector types:

```
T2Vector
T3Vector
T4Vector
T5Vector
T6Vector
```

These multidimensional vectors of dimension 2 through 6 are very convenient for working with multidimensional axis groups, i.e. you can use a 3 dimensional vector to store the destination for the move of a three dimensional machine, a T3Axis axis group.

Fields

TNVector object contain the following fields:

```
x:longint;
y:longint;
z:longint;
u:longint;
v:longint;
w:longint;
```

Vectors only contain as many components as their dimension, ie a T3Vector does not have a U component. These components can be handled as longints by specifying the vector name followed by a period followed by the component name, i.e.

```
XAxis.MoveTo(DestinationVector.X);
```

Methods

Vector infix operators are supported automatically because of Douloi Pascal's component-by-component application of infix operators to every member of an object structure. Note that vector cross or dot product multiplication is not implied by a vector times a vector but rather component by component multiplication. Additional methods supported by TNVectors include:

```
procedure Init(X:longint;Y:longint.......);
procedure Length:longint; {returns vector magni-
tude}
```

Examples

Imagine that you had a three dimensional machine named "Mill". The following button click procedure would move the mill along a constant vector displacement each time you clicked the button. The movement is with respect to a "BasePosition" of type T3Vector that has previously been established.

```
procedure Click;
   var DisplacementVector:T3Vector;


   begin
   DisplacementVector.Init(1000,2000,3000);
   BasePosition:=BasePosition+DisplacementVector;
   Mill.MoveToVector(BasePosition);
   end;
```

The following button click procedure should write out "1414":

```
procedure Click;
   var aVector:T2Vector;

   begin
   aVector.Init(1000,1000);
   Prompter.Writeln(aVector.Length);
   end;
```

Chapter 7

TFile (SAW only)

Description

TFiles are used to read and write information to DOS files. TFiles are an "objectified" version of the familiar Pascal file IO conventions. Write,Writeln,Read, and Readln, are not global procedures as in conventional Pascal but rather methods to a "receiver" which indicates where the information goes.

Methods

TFile objects respond to the following methods:

```
procedure TFile.Assign(Filename:string);
```
allocates TFile,associates file
```
procedure TFile.Reset;
```
prepares file for reading
```
procedure TFile.Rewrite;
```
prepares file for writing
```
procedure TFile.Close;
```
completes file management
```
procedure TFile.Write(...);
```
appends info to current line
```
procedure TFile.Writeln(...);
```
appends to line, advances line
```
procedure TFile.Read(...);
```
reads next item in line
```
procedure TFile.Readln(...);
```
reads next item, advances line
```
function  TFile.ReadLongint:longint;
```
returns next longint in file
```
function  TFile.ReadlnLongint:longint;
```
returns next longint and advances line
```
function  TFile.EndOfFile:boolean;
```
indicates if file is finished

TFiles know how to Writeln and Readln information. IO Errors that might occur during file handling are indicated with escapes, generally ReadEscapeCode and WriteEscapeCode. TFiles operate by sending mail to Windows which performs the DOS operations on behalf of the TFile. It is not possible for TFiles to directly access DOS because of the DOS reentrance limitation. Accordingly, do not expect TFile operations to be real-time. They are subject to physical time delays as well as Windows coopera-

tive multitasking delays. An example of a physical delay is reading disk information which is not in the disk cache requiring the hard disk read/write heads around to get the information. An example of a Windows delay might be dragging an application title block to move the application to a new location. Windows "blocks" all other Windows activity until the title bar is released,

Examples

The following code section might be used to perform a multiaxis "playback" of vector information stored in a file named "playback.vec". Note that this routine may be put on "hold" if Windows becomes preoccupied and does not read its mail.

```
procedure PlaybackVectorFile;

   var PlaybackFile:TFile;
   var Destination:T2Vector;

   begin
   PlaybackFile.Assign('playback.vec');
   PlaybackFile.Reset;
   while not PlaybackFile.EndOfFile do
      begin
      PlaybackFile.Readln(
         Destination.X,Destination.Y);
      XYAxis.MoveToVector(Destination);
      end;
   PlaybackFile.Close;
   end;
```

It may be important for that previous process to operate with the possibility of interruption. The best way to handle that is to move all the information into an array and have the system move from the array. This removes the mail-based file IO from operating while in the midst of motion. The following example moves information from a file into an array.

```
procedure FillVectorArray;

   var VectorFile:TFile;
   var scanner:integer;
   var NumberOfVectors:integer;

   begin
   VectorFile.Assign('playback.vec');
   scanner:=0;
   while not VectorFile.EndOfFile do
      begin
      scanner:=scanner+1;
```

```
      VectorFile.Readln(
         VectorArray[scanner].X,
         VectorArray[scanner].Y);
      end;
   VectorFile.Close;
   NumberOfVectors:=scanner;
   for scanner:=1 to NumberOfVectors do
      XYAxis.MoveToVector(VectorArray[scanner]);
   end;
```

The following routine creates a file of numbers followed by their squares.

```
procedure WriteSquares;

   var SquareFile:TFile;
   var Scanner:longint;

   begin
   SquareFile.Assign('squares.txt');
   for Scanner:=1 to 10 do
      SquareFile.Writeln(
         'The square of ',Scanner,
         ' is ',Scanner*Scanner);
   SquareFile.Close;
   end;
```

A file of vectors that might be used for a vector playback could be generated with the following click procedures.

```
procedure ResetButton.Click;
   begin
   Recorder.Assign('playback.vec');
   Recorder.Rewrite;
   end;

procedure AddPointToFileButton.Click;
   begin
   Recorder.Writeln(XAxis.ActualPosition,'
',YAxis.ActualPosition);
   end;

procedure FinishFile.Click;
   begin
   Recorder.Close;
   end;
```

Note that in the "AddPointToFileButton" routine the XAxis value and the YAxis value is seperated by an additional string literal parameter which is a space string. If this was missing the two values would be concatinated together in the file making one very long number instead of two seperate numbers. Also note that there is not direct support available for writing out objects. You must write out the components of the object individually.

Chapter 7

TPrompter (SAW only)

Description

TPrompters are used to display some information and suspend the execution of a program until that information has been acknowledged by the user. TPrompters are very similar to (and implemented with) the Windows MessageBox routines. The purpose is very similar to the "Prompt" command in a DOS batch file. As well as providing a predefined type, SAW provides a predefined instance of a TPrompter named "Prompter". Most of the time you can simply use Prompter rather than creating your own. TPrompters are most often used for "operator synchronization", i.e. to write messages such as "Load the part and press ok to continue".

Methods

TPrompters respond to the following methods:

```
TPrompter.Init;
TPrompter.Done;
TPrompter.Write(...);
TPrompter.Writeln(...);
```

TPrompters know how to Write and Writeln information. For TPrompters, write accumulators information for display, and writeln invokes the prompter to popup. All of the accumulated information is displayed. The TPrompter object then waits for the user to push the "ok" button before continuing with the program.

Init and Done only need to be used if you are creating your own TPrompter object. Note that these routines have constructor,destructor behavior, i.e. you really have to call them if you are declaring your own TPrompter before using any other methods or a Windows failure will occur.

Examples

The following code section might be found in a machine that was performing a machining operation on a part:

```
procedure MachinePart;
   begin
   Prompter.Writeln('Load next part');
   PerformMachiningOperation;
   end;
```

The following example might be used to do the same operation with an internally provided TPrompter:

```
procedure MachinePart;

   var PartPrompter:TPrompter;

   begin
   PartPrompter.Init;
   PartPrompter.Writeln('Load next part');
   PartPrompter.done;
   PerformMachingingOperation;
   end;
```

In the current release of SAW there is little benefit to creating your own prompter. This technique may be useful in the future when prompters have attributes that you would want to setup once and simply achieve by referring to the appropriately configured prompter.

Chapter 8

8) Advanced Motion Capabilities

Purpose

This section illustrates how the different features of Motion Server can be combined to provide advanced motion capabilities. Although some motion controllers provide these capabilities as "built in" to the controller, Motion Server is able to provide similar capabilities at the more flexible and tailorable application level rather than unchangeable firmware level. This list is no way is comprehensive but serves rather to illustrate how familiar modes might be implemented.

Electronic Gearing

Description

The conventional meaning of Electronic Gearing is a "driven" shaft rotating at some fixed speed ratio to a "driving" shaft. If the driving shaft rotates one turn, the driven shaft rotates half a turn in the same direction (for a ratio of 0.5, for example). If the driving shaft rotates 4 revs in the opposite direction, the driven shaft rotates 2 revs in the opposite direction. The driven shaft behaves as if it contained a gear in mesh with a gear on the driving shaft.

To be strictly correct, most commercial implementations of electronic gearing do not really implement the gear model since in a mechanical system it would be possible to apply a torque to the driven shaft that results in rotating the driving shaft. Gears trains of high efficiency can be made to move by driving from either end. To be less ambiguous about this case some manufacturers refer to electronic gearing more accurately as "position tracking" or "position following" mode. Note that achieving the bi-directional behavior is possible with Motion Server and is discussed under the "Bi-directional Force Reflection" mode.

Fundamental Principles

If we consider the case of just one shaft being the driving shaft and the other always being the driven shaft we can state the behavior of electronic gearing with the following statement:

$$Pn = Pd(Gr)$$

where:

Pn = Position of Driven Shaft

Pd = Position of Driving Shaft

Gr = Gear ratio

Implementation

Implementing electronic gearing simply involves setting the position of the driven axis to some factor times the position of the driving axis every sample period. Suppose the XAxis is the driving shaft, the YAxis is the driven shaft, and the gear ratio is to be 0.5. The following procedure would imlpement the electronic gearing equation:

```
procedure PerformGearing
   begin
   YAxis.SetCommandedPosition(
      XAxis.ActualPosition div 2);
   end;
```

This procedure properly updates the commanded position of the YAxis. In order for this to occur on an ongoing basis we need to schedule the task to run every millisecond. This might be done in the setup procedure of the main plate, or by a button's click procedure which you push when you want the mode to begin. An example of the latter case is:

```
TurnOnElectronicGearingButton.Click;
   begin
   ScheduleTask(TaskAddr(PerformGearing),1);
   end;
```

In implementing any advanced mode there are behaviors which may or may not be desirable. Often the application will clearly indicate what the more desired behavior would be. Having the mode implemented in the application level allows you to tailor the mode to be suitable for those needs.

For example, note that the PerformGearing procedure bases the position of the driven shaft on the actual position of the driving shaft. This is most suitable if the driving shaft is not being servo controlled, i.e. is a encoder based handwheel. However if the driving shaft is under servo control this may result in a compromise in performance. The slave not only tracks the position of the driving shaft but attempts to follow the errors of the driving shaft. You may instead really want the driven shaft to be following at a fixed ratio with respect to where the driving shaft is supposed to be, not with respect to where it actually is. This difference would be implemented by referring to the XAxis commanded position rather than actual position for the PerformGearing procedure as shown below.

```
procedure PerformGearing;
   begin
   YAxis.SetCommandedPosition(
      XAxis.CommandedPosition div 2);
   end;
```

Note that gearing of the YAxis is absolute, based on the position of the XAxis. This may not be what's desired if you were to run the YAxis on some other criteria and then "engage" the YAxis to be a driven shaft at some location which is not a simple ratio of X. Accomplishing a "phase offset" between the YAxis and XAxis is accomplished by adding an offset in the PerformGearing routine. This routine then becomes.

```
procedure PerformGearing
   begin
   YAxis.SetCommandedPosition(
      XAxis.ActualPosition div 2
      + PhaseAdjustment);
   end;
```

In this procedure PhaseAdjustment might be a global variable defined in a plate procedure.

Limitations

One limitation of this approach is continuous motion gearing. If the driving shaft was told to jog at a constant speed indefinitely, eventually there would come a point the where driving shaft "wrapped around" its position counter. Although Jog mode handles this correctly, the driven shaft would see the driving shaft position suddenly change from a very large positive number to a very large negative number and attempt to jump to this new location. The system would shut down the YAxis when it experienced this unreasonable request. If you need to implement electronic gearing on a continuous basis the PerformGearing routine would need to measure the change in position of the driving shaft between this sample and the last, calculate a ratio on this delta, and then add to the driven shaft position this ratioed delta. This approach solves the continuous motion problem but has the possibility of accumulating small errors over time. The absolute position ratioing approach has the advantage of sustaining no accumulating errors. Which method is best depends on the nature of the application. Having the ability to tailor the motion mode

allows you to responsd to the requirements of your particular problem. For additional details on incremental electronic gearing feel free to call Douloi Automation.

Electronic Gearing with Trapezoidal Phase Advance

Description

Electronic gearing most often involves a fixed phase angle between the driving shaft and driven shaft. However there are times when phase advancing the driven shaft is desirable. For example, a process might note the actual position of a piece of material as being several inches behind where it was expected to be on a moving carriage that is being driven in electronic gearing mode. It would be appropriate to advance that carriage ahead the lacking 2 inches while maintaining the carriages relationship to the entire process.

Fundamental Principles

A statement of the position of a driven shaft with respect to a driving shaft including phase advance is:

$$Pn=Pd(Gr) + Ta$$

where:

Pn = Position of Driven Shaft

Pd = Position of Driving Shaft

Gr = Gear ratio

Ta = Fixed Phase Advance

The more general expression is:

$$Pn=Pd(Gr) + T(t)$$

where:

Pn = Position of Driven Shaft

Pd = Position of Driving Shaft

Gr = Gear ratio

T(t) = Phase advance angle as a function of time

Implementation

The desired phase advance angle, or total amount of phase advance, is usually a number that results from some registration information in the process, for example in the initial illustration the total amount of distance that the carriage was lagging was 2 inches. The phase advance angle shown in the equation above needs to smoothly change in value from 0 to the desired total phase advance angle. It would be possible to create a function which generated a smoothly varying numeric value to use as a function in the electronic gearing equation however this would be re-inventing the wheel. The activity of creating smoothly varying numbers is extremely common in motion controllers and is typically caller "profiling". The commanded setpoints for a motor during a trapezoidal move is calculated by the profiler with all of the conveniences of having a settable accel, decel, and slew. The simplest way to achieve smooth phase changes is to make use of the profiler. The simplest way to use the profiler for general numeric activity is through a "virtual axis".

A virtual axis behaves just like a regular axis however there is no motor attached. In Motion Server the virtual axis are named RAxis, SAxis, and TAxis. The RAxis has a commanded position just like the XAxis does. The RAxis can have its accel set just like the XAxis. The RAxis can be told to move and its commanded position will change to the new position with the smooth motion characteristic of trapezoidal motion profiles. We will take advantage of this smoothly changing position value to implement a smooth phase advance in the real mechanical system.

The PerformGearing procedure shown in the last motion capability discussion is now shown here with an additional phase angle term:

```
procedure PerformGearing;
   begin
   YAxis.SetCommandedPosition(
      XAxis.ActualPosition div 2
      + RAxis.CommandedPosition);
   end;
```

Before this routine is schedule the following prepartion routine is run, most likely in the setup procedure of the associated plate:

```
procedure InitializeElectronicGearing;
   begin
   RAxis.SetCommandedPosition(0);
   RAxis.SetAccel(1);
   RAxis.SetDecel(1);
   RAxis.SetSpeed(10);
   ScheduleTask(TaskAddr(PerformGearing),1);
   end;
```

When this initialization routine begins, the RAxis reports as its commanded position the value 0. The RAxis was last assigned to have a commanded value of 0 and it has not been asked to do anything yet, so it returns as its position 0. The electronic gearing begins and operates as has been discussed. Now it is time to perform a phase advance. This is accomplished by simply moving the RAxis to a new location, the amount of phase advance desired. For example the following click procedure would advance the phase by 20000 counts:

```
procedure AdvancePhaseButton.Click;
   begin
   RAxis.MoveTo(20000);
   end;
```

The RAxis commanded position value smoothly changes from the value 0 to the value 20000 with the well behaved properties of a trapezoidal profile. The procedure PerformGearing is superimposing, in real time, two independent criteria for positioning the YAxis, the electronic gearing criteria and the phase advance criteria. Superimposition of motion critieria is a very powerful technique yet very easy to implement.

Electronic Cam

Description

"Electronic Cam" is a motion mode where an axis performs periodic motion based on usually a geometric specification (as distinct from accel, decel, speed time related specifications). The axis moves back and forth as if driven from a mechanical cam. The conceptual cam is sometimes "driven" by an independent piece of information such as the speed of a winding mandrel or some other uncontrolled "master" in the machine.

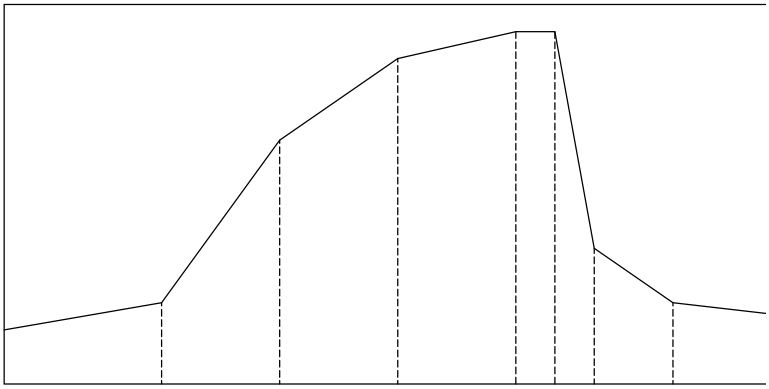
Fundamental Principles

The foundational specification in an electronic cam application is the the cam's shape. This is most easily done by representing the shape in an array of T2Vectors. This point-pair array contains for the X component the angular location of the point pair, and for the Y value, the displacement of the cam at that location.The total extent of the X component values represents 360 degrees of cam rotation. The array should describe a "single valued" function, i.e. a curve fitting through the points should have only one Y value for every X value. From this array it is possible to interpolate between point-pair vectors to calculate, for any arbitrary input angle, the corresponding displacement of the cam. This "mapping" principle is then used every sample period to calculate the commanded setpoint displacement for an axis based on the input "angle" of the cam which can come from any source. Examples of sources might be a "virtual axis" for profiled movement of the cam, or an encoder monitoring the speed of an uncontrolled axis external to the control system, or a sensor monitoring material flow. Note that this approach does not require a large number of vectors since interpolation can be performed between vectors.

Implementation

The main components of an electronic cam application include the cam's descriptive geometric array, the procedure to interpolate between these descriptive points to calculate the CamOutputPosition, and a routine to set the commanded position every millisecond based

on some cam-driving criteria. An example of a descriptive cam geometry is shown below:



X=Cam Input Position, Y=Cam Output Position

The domain, or x axis of the figure, represents the range of positions that would be the "input" to the cam. If the cam is to "driven" by an encoder the length of this x axis would be the encoder resolution of this driving encoder. This would cause the cam to behave as if it was mechanically attached to the driving encoder shaft.

The range, or y axis of the figure, represents the corresponding displacement that should occur given an input position. Note that the points that describe the curve are not necessarily evenly spaced. The size of this range can match the size of displacement you would like to achieve. If you want a variable displacement, the y axis scale of the cam description can be set arbitrarily and the final position calculated by multiplying the output displacement from the curve by a scaling factor to achieve variable cam amplitudes.

The simplest way to describe a cam geometry is with an array of T2Vectors, two dimensional vectors. The X component of any particular vector represents the x position of the curve point in the figure. The Y position corresponds to the amplitude of the cam's displacement at that point. The number of vectors is determined by the number of points you would like to use to describe the cam. In the figure, 9 points are used. The array is most easily created as a SAW "plate" variable. The conventional declaration is shown below:

```
const CamArrayLength=9;
var CamArray:array[1..CamArrayLength] of T2Vector;
```

There are many ways to fill the array with information. You can read information from a file, calculate the cam geometry, or simply assign it. The following procedure would initialize the cam geometry with a set of values covering a domain of 2000 counts:

```
procedure FillCamArray;
   begin
   CamArray[1].Init(0,200);
   CamArray[2].Init(400,250);
   CamArray[3].Init(600,700);
   CamArray[4].Init(900,800);
   CamArray[5].Init(1100,900);
   CamArray[6].Init(1200,900);
   CamArray[7].Init(1300,400);
   CamArray[8].Init(1500,300);
   CamArray[9].Init(2000,200);
   end;
```

You would probably want to call this procedure in the setup routine of your main plate so as to have it all ready to go when the application starts. The X values range from 0 to 2000, always increasing. The Y values range from 200 to 1000. Note that the last value Y value is the same as the first. This is important since a physical cam has a continuous position and does not suddenly change displacement. This software cam needs to be continuous also at the "wrap around" point. Y displacement motion should flow smoothly off the "end" of the cam array and connect to the start of the array by having the same value at both ends.

The next step is to create a function which will provide a displacement value given any number between 0 and 2000, the range of cam rotation. This will be accomplished by interpolating between the points provided that specify the cam. The function CamOutputPosition is shown below.

```
function
CamOutputPosition(InputPosition:longint):longint

   var BeforeVector:T2Vector;
   var AfterVector:T2Vector;
   var DeltaVector:T2Vector;
   var scanner:integer;
   var AfterVectorFound:boolean;

   begin
   InputPosition:=InputPosition mod 2000;
   AfterVectorFound:=false;
   scanner:=2
   repeat
```

```
   if CamArray[scanner].X > InputPosition then
      AfterVectorFound:=true
   else
      begin
      scanner:=scanner+1;
      if scanner > CamArrayLength then
         Escape(CamOutputPositionFailure);
      end;
until AfterVectorFound;

AfterVector:=CamArray[scanner];
BeforeVector:=CamArray[scanner-1];
DeltaVector:=AfterVector-BeforeVector;
CamOutputPosition:=BeforeVector.Y
   + DeltaVector.Y*(CamInputPosition
   - BeforeVector.x) div DeltaVector.X
end;
```

The procedure starts by using the "mod" operator. The InputPosition could represent any rotational location. The "mod" provides the remainder of rotation realized after all of the integer number of turns have been removed giving a number in the range 0 to 1999. A loop is then performed which scans through the CamArray looking for the vector which is "next" to the InputPosition on the high side. The technique shown here is a simple linear search, suitable for arrays of less than 20 vectors or so. If you have more vectors you can save time with a "binary search" technique. Consult Douloi Automation for further recommendations if you have a high resolution CamArray. The vector found beyond the InputPosition is called the AfterVector. The BeforeVector is the previous one. The InputPosition lies in between. The DeltaVector is the difference, in both X and Y, between the AfterVector and the BeforeVector, i.e. the DeltaVector is what gives you the AfterVector when added to the BeforeVector. Note that the DeltaVector is quite simply calculated as the difference between the two vectors. Douloi Pascal supports infix operators on vectors for addition and subtraction. The CamOutputPosition is then calculated to be the Displacement of the BeforeVector plus an additional amount based on scaling the difference between the vectors by the excursion into that section of the CamArray.

To achieve motion with the cam array a procedure must be written that "connects" system information to the cam. Let's suppose that the ZAxis is being used to provide the cam input position, i.e. the Z axis encoder is connected to a section of the machine, perhaps externally driven, that represents where the electronic cam should be. The XAxis displacement will be the electronic cam output. The following

procedure would make the connection between these elements:

```
procedure SetCamPosition;
   begin
   XAxis.SetCommandedPosition(
      CamOutputPosition(ZAxis.ActualPosition));
   end;
```

It may be desirable to scale the cam output so as to provide various sized cams. This routine is a good place to do such scaling such as the variation below shows:

```
procedure SetCamPosition;
   begin
   XAxis.SetCommandedPosition(
      round(CamOutputPosition(
         ZAxis.ActualPosition)*1.53));
   end;
```

In order for this cam to take effect the procedure SetCamPosition needs to operate every millisecond. This is done by scheduling the cam to run with ScheduleTask. A good place for this might be a button click procedure. Note that the first step, before scheduling the task to run, is to get the XAxis in position so that no sudden change in position occurs when the cam "engages":

```
procedure Click;
   begin
   XAxis.MoveTo(
      round(CamOutputPosition(
      ZAxis.ActualPosition)*1.53));
   ScheduleTask(TaskAddr(SetCamPosition),1);
   end;
```

If you are planning to scale the cam it is better to create a constant to represent the number and use that constant (i.e. const CamScaleFactor=1.53). Note that the constant appears in two seperate places - the SetCamPosition routine and the button click procedure which needs the information so as to contact the cam with the XAxis before engaging the cam. Using constants helps these two different uses of the information "stay in synch".

Limitations

An electronic cam is a geometric description of what will become a movement. It is quite possible to create cam geometries which are unrealistic (just as there can be unrealistic mechanical cam geometries). Sudden changes in the slope of the cam produce large accelerations. Slopes of the cam represent speeds. There will be a cam input velocity beyond which the servo will not be able to track. The Slew, Accel, and Decel that describe a trapezoidal profile do not come into play here since electronic camming is providing another basis for determining the motor's commanded position. With the flexibility of specifying an arbitrary commanded set point for the motor comes the responsibility of making sure that you ask the motor to do something reasonable. In this type of application the main expression of what constitutes reasonable or unreasonable is in the shape described by the cam geometric array. Douloi Automation is available to assist you through any misunderstandings that may occur in the application of this technique. Please feel free to call.

Tangent or "Knife Cutter" Servoing

Description

Tangent of "Knife Cutter" servoing is a motion mode where an axis, often the ZAxis, controls an oriented tool, such as a knife, while the X and Y axis are performing a contour. As the X and Y axis move, the Z continually needs to reorient the knife so that the knife is cutting in the direction of motion rather than scraping sideways against it. The method shown has the capacity to orient the knife to the actual contour experienced, not just the commanded contour intended.

Fundamental Principles

The basic approach is to monitor the displacements of the X and Y axis. When the displacement is numerically significant, the arc tangent function in the math coprocessor is used to calculate what angle represents this current direction of travel. The ZAxis is then told to move to this angle. The tricky thing about tangent servoing is handling "wind up" correctly. For example, if you were cutting a spiral on a planar surface, such as the groove in an LP record, the range of angles returned by an arctangent function would always be in the range +180 to -180. However, you don't want the knife to spin 360 degrees as you cross through the angle that on one side is -180 and on the other is +180. Although -180 and 180 are the same angle, they are 360 degrees apart from the servo's point of view. In order to prevent the ZAxis from "snapping" backwards with 360 degree spins some additional measures must be taken to track total angular accumulation.

Implementation

The following procedure calculates the appropriate angle for a tangent servoing application:

```
function TangentAngle:longint;

   var AnchorVector:T2Vector; static;
   var CurrentVector:T2Vector;
   var DeltaVector:T2Vector;
   const NumericThreshold=50;
```

```
var CurrentAngle:single;
var LastAngle:single; static;
var RevAccumulator:longint;
var LastAnswer:longint;


begin
XYAxis.GetActualPositionVector(CurrentVector);
DeltaVector:=CurrentVector-AnchorVector;
if DeltaVector.Length > NumericThreshold then
   begin
   FPushLongint(DeltaVector.Y);
   FPushLongint(DeltaVector.X);
   FPArcTan;
   PopSingle(CurrentAngle);
   if CurrentAngle - LastAngle > pi then
      RevAccumulator:=RevAccumulator-1
   else if CurrentAngle-LastAngle < -pi then
      RevAccumulator:=RevAccumulator+1;
   LastAngle:=CurrentAngle;
   AnchorPosition:=CurrentPosition;
   LastAnswer:=
      round((CurrentAngle/
      (2*pi)+RevAccumulator)*CountsPerRev);
   end
TangentAngle:=LastAnswer;
end;
```

The variables identified as "static" in the declaration section are best defined in a plate procedure rather than in this procedure so that they can be initialized prior to their use here. The AnchorVector represents a base point from which an angle calculation will be made. The DeltaVector is calculated as the different between the current position and the anchor. When the length of the delta vector is large enough, we can expect there to be good numeric resolution for calculating the angle. Without this check we'd be taking the arctan of displacements of only a few counts leading to very quantized calculations. The math coprocessor is needed to perform the actual arctan function real time. The FInit function should be placed in the plate setup procedure to prepare the math coprocessor for use. After calculating the arctan, the RevAccumulator is maintained by checking for "large leaps" in position. Such large leaps are indicitive of crossing over the arctan range and for the actual motion system represent accumulated rotations of the cutting knife. If sufficient movement for good numeric resolution has not occured yet, the function just returns the last answer from the previous calculation.

In order to have the knife cutter move by this calculation a relationship between the calculation and the mechanism must be made. This is provided by the following procedure:

```
procedure SetTangentAxis:
   begin
   Try
      ZAxis.BeginMoveTo(TangentAngle)
   Recover
      begin
      if EscapeCode <> MotionOverrunEscapeCode then
         Escape(EscapeCode);
      end;
   end;
```

SetTangentAxis tells the ZAxis to begin moving to the TangentAngle. This motion is performed with a trapezoidal velocity profile so that the ZAxis moves in a well behaved manner regardless of how suddenly the angle might change, i.e. turning a right angle corner during the cutting operation. BeginMoveTo is used instead of MoveTo because it is necessary to immediately continue calculating the angle. It is quite likely that the ZAxis will never get to the first destination since the destination will be updated on the next sample giving the ZAxis a continually moving target for the motion. The recover block is provided to catch the case where the ZAxis is not able to turn around quickly enough (Motion Overrun). In this case the ZAxis will simply stop (which is necessary if you need to turn around) and will be able to head towards the destination the next time around when that move request is made again. Some applications may require a "snappier" response from the ZAxis. These application can simply set the ZAxis position directly from the angle information and go around the profiler. This routine would then be just:

```
procedure SetTangentAxis;
   begin
   ZAxis.SetCommandedPosition(TangentAngle);
   end;
```

The only problem with this is the sudden change that will be asked of the ZAxis if the XYAxis makes a sharp corner. This could result in sudden and surprising motion as well as more demands on the power electronics and mechanical structure than is necessary for the application.

For SetTangentAxis to have an ongoing effect it must scheduled. The following routine might be placed in a click procedure for a button named "engage":

```
procedure Click;
   begin
   LastAngle:=0;
   XYAxis.GetActualPosition(AnchorPosition);
   RevAccumulator:=0;
   LastAnswer:=0;
   ZAxis.MoveTo(0);
   ScheduleTask(Addr(SetTangentAxis),1);
   end;
```

This procedure resets some of the variables used to recognize changes in system state, moves the ZAxis to an initial position, and schedules the SetTangentAxis routine to run every controller sample.

Limitations

As illustrated, angles are based on the actual positions that the machine realizes. This may be good or bad, depending on the nature of the application. For machines where the X and Y axis are handwheels, this technique could allow automatic rotation of the knife while an artist directed the contour. If the motion is machine generated, it is possible to base the CurrentVector not on the actual machine positions but rather on the commanded positions providing a smoother theoretical basis for the angle rather than empiral basis. The value of NumericThreshold must tradeoff between being small enough to not delay the calculation of an angle in a timely way while at the same time not being so small that the quantization effect causes the knife to move in a "chunky" manner.

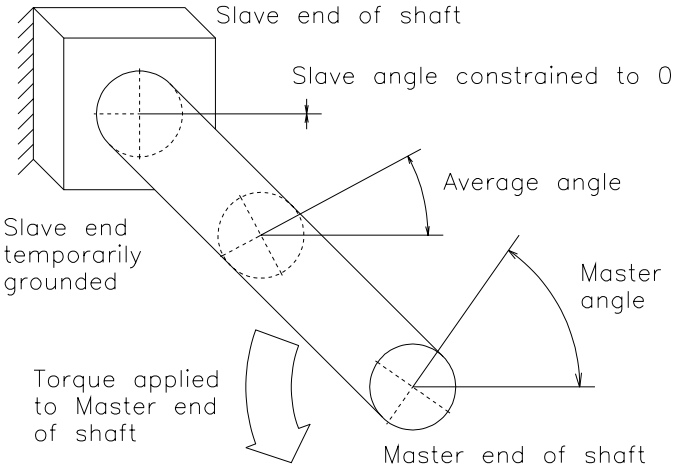
Bi-directional Force Reflection

Description

Bi-directional force reflection is a more complete implementation of electronic gearing. With a physical gear train of high efficiency it is possible to apply torque to either end of the drive train and have motion result. If opposing torques are encountered by the output end of the gear train those torques are realized at the driving end of the gear train. This section describes how to implement such a bi-directional behavior for a gear ratio of 1 to 1.

Fundamental Principles

The following figure shows an elastic shaft model that will be used to determine how to implement bi-directional force reflection.



One end of the shaft represents a master joint and the other end of the shaft represents the corresponding slave joint. Consider the case that the slave has encountered an obstacle and, constrained not to rotate, is temporarily grounded. A torque applied to the master end of the shaft causes an angular strain, theta, proportional to that applied torque. Half way down the length of the shaft a strain of Theta/2 has occurred. The “point of symmetry” in this problem is the middle of the shaft since either mechanism can direct the system

behavior in force reflection mode. By establishing a reference frame along the rotated theta/2 section of the shaft midpoint; it can be seen that each end of the shaft is attempting to elastically return to 0 angular displacement with respect to this frame.

The most fundamental behavior of a position servo is to apply a restoring torque proportional to an angular displacement from a commanded set point. Accordingly, one simple implementation of bi-directional force reflection is to use a PD position servo for each mechanism and continually command each servo to move to the average position of the two.

Implementation

Implementing bi-directional force reflection requires continually calculating the average position between the driving, master end of the shaft and the driven, slave end of the shaft. This might be implemented with a procedure such as is shown below.

```
procedure PerformForceReflection

   var AveragePosition:longint;

   begin
   AveragePosition:=
      (XAxis.ActualPosition+YAxis.ActualPosition)
         div 2;
   XAxis.SetCommandedPosition(AveragePosition);
   YAxis.SetCommandedPosition(AveragePosition);
   end;
```

The procedure PerformForceReflection will need to operate on a continual basis. Before turning it on, however, it would be good for one of the two axis to move to the position of the other so as to not have a sudden change in position expected on the first sample. An appropriate procedure to launch the force reflection might be.

```
TurnOnForceReflection.Click;
   begin
   YAxis.MoveTo(XAxis.ActualPosition);
   ScheduleTask(Addr(PerformForceReflection),1);
   end;

   begin
   YAxis.SetCommandedPosition(
      XAxis.ActualPosition div 2 +
      PhaseAdjustment);
   end;
```

Limitations

For force reflection to work it is necessary for the integrators of the servos to be off; i.e. XAxis.SetIntegrator(0) and YAxis.SetIntegrator(0) or the "elastic" behavior, which the method is based on, is not operating. Also note that there should be no significant torque offsets. A torque offset cannot be distinguished from an external torque, each producing a position error that results in the opposite axis following and both motors spinning even without an external torque.

Chapter 9

Using the ServoLib Dynamic Link Library

Purpose

There are several ways to make use of the ServoLib DLL to simplify the construction of motion control Windows applications. This section describes alternative techniques and provides a command reference that provides the needed information to link to the library from your own Windows programming environment.

Functionality through Douloi Pascal

It is possible to access the multitasking capabilities and high speed program execution of Douloi Pascal from your Windows language system. In this approach you would write, as a conventional text file, Douloi Pascal procedures and variable declarations such as in the tutorial and example programs for SAW. You then have your application compile this text file into high speed object code through a ServoLib DLL call. After the program has been compiled, the Douloi Pascal routines can be started or scheduled to run periodically through the DLL. This approach is very useful when you want high speed performance even though your Windows language system might not be high speed (i.e. an interpreted environment). This also allows you to start special behaviors such as electronic gearing or tangent servoing which operate on their own while you manipulate the servo system through other means.You are not able to access SAW objects with this approach because SAW is not present; however much of the real-time system behavior that you have studied in the Servo Application Workbench can be accessed.

Functionality through Direct Calls

It is also possible to access primarily the motion system through direct DLL calls. These calls have basically the same names as their Douloi Pascal method names with the exception of procedures with dimension-dependent parameter list lengths. These procedure names are prefixed with their class name. A syntax rearrangement is needed since DLL's do not support object syntax as part of the interface definition. Parameters from a normal

object procedure call become sandwiched between a new first parameter, which is the receiver, and a new last parameter, which is the error code. For example, if you wanted to move the XAxis by 20000 counts in Douloi Pascal you would type:

```
XAxis.MoveBy(20000);
```

Any problems with this procedure would result in an escape. Using the DLL interface however, you must type:

```
T1AxisMoveBy(XAxis,20000,ErrorCode);
```

The first parameter is what had been the receiving object in Douloi Pascal. The last parameter is filled with what would have been an escape code in Douloi Pascal. ErrorCodes of 0 indicate that no problem was encountered. The exact syntax is governed by the nature of your language system. Included is a "DLL Interface Unit" for Borland Pascal, Borland C++, and a Global File for Visual Basic.

Direct calls are very helpful when you are programmably deciding what to do and cannot afford to wait for the compiler to process new Douloi Pascal programs for you. The best strategy is to use both approaches at the same time, each one doing what it does best. You can compile and "spawn" processes which perform real-time custom coordination or data collection activities on their own while you direct the motion of the motors with direct DLL calls.

Usage

You are free to make direct calls and access the functionality of the Servo Library only after you have first called InitializeServoSystem. This call prepares the system for operation. If you forget to make this call the DLL will fail and cause your application (and possibly development system) to fail with it resulting in loss of work and a restart of your application development environment.

Most functions take a pass-by-reference integer parameter for reporting escape code information. After a call, this error reporting variable will be 0 if all went well and non-zero to indicate the particular error. In all cases except InitializeServoSystem, this variable is also an input parameter. If the value is not 0 the call will immediately exit and not attempt the operation leaving the variable unchanged. This allows you to construct linear sequences of motion system calls without having to constantly check if a failure occurred. If a failure does occur in the sequence; the offending routine exits with a non-zero error code and subsequent commands will be internally skipped. At the end of the sequence, you can interrogate the

variable to determine if the entire sequence operated correctly or if there was a problem. Example1 demonstrates this technique. If you do detect that an error has occured, you must set the error variable to 0 yourself before making another call. Otherwise the routine will see the non-0 error code and not even attempt its operation.

Examples on the uses of the DLL are provided in the following secitons. The same set of examples is implemented in the four language systems: Turbo C++, Turbo Pascal for Windows, Visual Basic, and Servo Application Workbench (for comparison purposes). Skip the languages you are not concerned with and go to the examples in your language system. The interface mechanism (i.e. prototype headers and .LIB file for C++ , interface unit for TPW, and global library declarations for VB) is provided in the appropriate LINK2xxx directory. If your language system supports projects then open up the examples as projects. The examples should clarify the details of usage.

Note that, conforming to the Windows API model, pointers are manipulated as longints, the cross-language universal 32 bit pointer type, offering the most general usage but least pointer-type protection.

The details of the DLL commands are found in the help system. Click on "DLL Command Reference" from the help index. The following pages provides some example programs of how the DLL might be used.

For those first moving to C++ or Visual Basic from a Pascal background/mindset be aware that the following action, quite intuitive and correct to a Pascal programmer, will create a compiler-detected but non-obvious error:

```
SetServo(XAxis,1,ErrorCode);
```

Although this looks completely reasonable it does not work. Procedures and functions in both C++ and Visual Basic must have parameter parenthesis even if they don't have parameters. Without the parenthesis, both languages interpret XAxis as something besides the XAxis function in the ServoLib and the program does not compile. Remember to add an empty parameter list to all procedures and functions, i.e.

```
TNAxisSetServo(XAxis(),1,ErrorCode);
```

Turbo Pascal for Windows DLL Examples

Turbo Pascal for Windows makes use of the ServoLib DLL most conveniently through the ServoInt Interface unit. This unit should be "included" in the uses clause of your program. Actual access to the DLL is performed by references from ServoInt. You may find it necessary to edit ServoInt if you intend to locate the ServoLib DLL in a directory different from the current directory when your application runs.

The following examples illustrate how calls can be made to the motion system as well as the compiler.

TPW Example 1 - Direct Access

Description

This example uses direct calls to move the XAxis motor to several different locations and to write out commanded position information. This would represent the simplest use of ServlLib.

Source Code

```
{************************************************************************}
{************************************************************************}
{**********                                                    **********}
{**********                    TPW DLL Example 1               **********}
{**********                                                    **********}
{**********------------------------------------------------------**********}
{********** Created By: Randy Andrews | Creation Date: 05-oct-92  **********}
{**********------------------------------------------------------**********}
{********** Description                                         **********}
{**********                                                    **********}
{********** This represents a simple program to move the XAxis    **********}
{********** to several different positions. This illustrates      **********}
{********** initialization and direct calls.                   **********}
{**********                                                    **********}
{************************************************************************}
{************************************************************************}
```

```
program TPWDLL_1;

uses
  ServoInt,
  WinCrt;

var ErrorCode:integer;

begin
Writeln('TPW DLL Example 1');
InitializeServoSystem(800,0,     ErrorCode);
EngageHighSpeedClock(ErrorCode);
TNAxisSetServo(XAxis,  1,  ErrorCode);
T1AxisMoveTo(XAxis,    0,  ErrorCode);
T1AxisMoveTo(XAxis, 5000,  ErrorCode);
T1AxisMoveTo(XAxis,10000,  ErrorCode);
T1AxisMoveTo(XAxis,    0,  ErrorCode);
T1AxisMoveBy(XAxis, 2000,  ErrorCode);
T1AxisMoveBy(XAxis,-4000,  Errorcode);
T1AxisMoveTo(Xaxis,    0,  Errorcode);
Writeln(T1AxisCommandedPosition(XAxis));
if ErrorCode <> 0 then
   begin
   MessageBox(0,'Motion Problem','Status',mb_ok);
   ErrorCode:=0; {note need to explicitly set error code to 0}
   end;
TNAxisSetServo(XAxis,  0,  ErrorCode);
DisengageHighSpeedClock;
Writeln('done');
end.
```

Explanation

The first operation performed with the motion system is its initialization. The initialization routine sets the error code variable to 0. After initializing the servo system the XAxis is directed to move to different destinations. It is extremely important that the dimension of the procedure match the dimension of the function, ie T1AxisMoveTo should only be used with a T1Axis parameter.

Note that, while motion is ocurring, Windows interaction comes to a halt. By using the MoveTo and MoveBy routines the program waits for the moves to finish. Although simple to synchronize, this approach is very limiting for an interactive application. The following examples show ways to not have this limitation.

TPW Example 2 - Direct Access

Description

This example expands on the first by starting a motion and then performing another operation while the motion is taking place.

Source Code

```
{************************************************************************}
{************************************************************************}
{**********                                                    **********}
{**********              TPW DLL Direct Call Example 2         **********}
{**********                                                    **********}
{**********------------------------------------------------------**********}
{********** Created By: Randy Andrews | Creation Date: 12-oct-92  **********}
{**********------------------------------------------------------**********}
{********** Description                                         **********}
{**********                                                    **********}
{********** This moves an axis while performing other activities  **********}
{********** during the move. This illustrates the use of direct   **********}
{********** calls to learn about motion status during movement.   **********}
{**********                                                    **********}
{************************************************************************}
{************************************************************************}

program TPWDLL_1;

uses
  ServoInt, {interface unit to ServoLib DLL}
  WinCrt;   {"conventional" system IO simulation unit}

var ErrorCode:integer;
var MoveOver:boolean;
var MoveDurationCounter:longint;

begin
Writeln('TPW DLL Example 2');
MoveDurationCounter:=0;

InitializeServoSystem(800,0,ErrorCode);
EngageHighSpeedClock(ErrorCode);
SetServo(XAxis,  1,ErrorCode);

T1AxisBeginMoveBy(XAxis,50000,ErrorCode);
if ErrorCode <> 0 then
   begin
   Writeln('Motion System Problem: ',ErrorCode);
   DisengageHighSpeedClock;
   exit;
   end;
```

```
writeln('Free to continue...');
repeat
   MoveOver:=MoveIsFinished(XAxis,ErrorCode);
   if ErrorCode <> 0 then
         begin
         Writeln('Interogation problem');
    DisengageHighSpeedClock;
         exit;
         end;

   {Perform other operations here while waiting}
   Inc(MoveDurationCounter);

until MoveOver;
Writeln('Move Duration Counter: ',MoveDurationCounter);
Writeln('done');
end.
```

Explanation

As in the first example, the first operation is to initialize the system. Instead of performing motion with "MoveBy" and "MoveTo" the command "BeginMoveBy" is used. Commands that start with "Begin" imply that they do not wait for their operation to be complete. The move is started and program execution immediately continues. The program then performs other operations (in this case simply incrementing a variable) and periodically checks to see if the motion is completed by interrogating the servo system to find out when motion is complete.

Although a small improvement over the first example in terms of allowing other operations to continue during the motion program, this is not a very complete solution either. Although other activities can occur during motion Windows is still "locked out" (i.e. try to move the window by dragging on the title bar during operation of the motion). The next example illustrates how to have an independent motion program operating at the same time as Windows interaction.

TPW Example 3 - Combined Access

Description

This example uses a seperate Douloi Pascal program to operate the XAxis while Windows continues operation in an independent manner.

Douloi Pascal Source Code

```
{Douloi Pascal Program for DLL Example 3}
{filename: ManyMove.dps}

procedure PerformManyMoves;
   begin
   XAxis.MoveTo(10000);
   XAxis.MoveTo(0);
   XAxis.MoveTo(20000);
   XAxis.MoveTo(0);
   XAxis.MoveTo(30000);
   XAxis.Moveto(0);
   XAxis.MoveTo(40000);
   XAxis.MoveTo(0);
   end;
```

Source Code

```
{*************************************************************************}
{*************************************************************************}
{**********                                                     **********}
{**********              TPW DLL Direct Call Example 3          **********}
{**********                                                     **********}
{**********-----------------------------------------------------**********}
{********** Created By: Randy Andrews | Creation Date: 13-oct-92  **********}
{**********-----------------------------------------------------**********}
{********** Description                                          **********}
{**********                                                     **********}
{********** This illustrates how to begin a seperate task to    **********}
{********** perform motion while Windows continues to perform   **********}
{********** operations with the user.                           **********}
{**********                                                     **********}
{*************************************************************************}
{*************************************************************************}
program TPWDLL_3;

uses
   ServoInt, {interface unit to ServoLib DLL}
   WinCrt;   {"conventional" system IO simulation unit}

var ErrorCode:integer;
var BookMark:array[0..79] of char;
```

```
var ErrorString:array[0..79] of char;
var Row,Column:integer;

begin
Writeln('TPW DLL Example 3');

InitializeServoSystem(800,0,ErrorCode);
Writeln('Initializing Compiler...');
InitializeCompiler;
Writeln('Compiling Program...');
Compile('..\ManyMove.dps',BookMark,Row,Column,ErrorCode,ErrorString);
if ErrorCode <> 0 then
   begin
   writeln('Compiler problem: ',ErrorCode);
   exit;
   end;

EngageHighSpeedClock(ErrorCode);
SetServo(XAxis,  1,ErrorCode);

BeginTask(TaskAddr('PerformManyMoves'),ErrorCode);
if ErrorCode <> 0 then
   begin
   Writeln('Problem starting procedure PerformManyMoves');
   exit;
   end;

Writeln;
Writeln('Task launched, TPW program done.');
Writeln('Do not forget to turn off ignition before');
Writeln('leaving Windows.');
end.
```

Explanation

Examples 1 and 2 could have been done with a conventional PC based motion control card. This example is the point where Motion Server's architecture begins to make a contribution. The motion task is specified in the file ManyMove.dps (dps stands for Douloi Pascal Source). Note that the program is written in Douloi Pascal and not in the language of your DLL invoking host environment (i.e. not in TPW or C++ or VB).

This program is then compiled by your program at run-time through the Compile command implemented in the ServoLib DLL. The different parameters given to the Compile command help focus in on where errors are located in the event there is a mistake in the Douloi Pascal program. Note that InitializeCompiler needs to called before you use the Compile command, however it only needs to be called once at the beginning of the application regardless of the

number of times Compile is used. Most of InitializeCompiler's activity is including the "system constants" that are described in the file STANDARD.INC. If you do not use these constants you can reduce the time delay of InitializeCompiler by deleting the symbols you do not need.

The host program then begins the Douloi Pascal program with the command BeginTaskAtCName (as distinct from a pascal style name). Strings in ServoLib are passed as null terminated, "ASCIIZ" strings, or "C style" strings. The motion program is now operating, running the motor, while the host Windows application is free to do whatever it would like, including terminating! The motion system is quite independent of any particular Windows application once it has been started and does not required the application to remain present to operate. This degree of independence requires some thoughtful consideration since you would not want to start an operation and then lose the means of stopping it. Also note that an operating motion program will continue to operate even if the Windows application that started it experiences a UAE. For these reasons it is a good idea to hook up a hardware switch to one of the several User Disable inputs so as to be able to shutdown the motion system in the event that your Windows application does not perform as expected.

Note that the program turns on the high speed clock, but does not turn it off. You must turn off the high speed clock yourself with the ServoIgnition utility, or more likely, in the close procedure of your Window applications Main Window. Make sure the "ignition is turned off" before leaving Windows or invoking a DOS shell.

Turbo C++ for Windows DLL Examples

To make use of the ServoLib DLL from a Turbo C++ application, place an include directive in the application source code that includes the ServoLib.h header file. You will also need to include as a component to the project the ServoLib.Lib file through Project\Add.

The following examples illustrate how calls can be made to the motion system as well as the compiler.

C++ Example 1 - Direct Access

Description

This example uses direct calls to move the XAxis motor to several different locations and to write out commanded position information. This would represent the simplest use of ServlLib.

Source Code

```
// Borland Turbo C++ Example 1
// The following example is based on a borland Object Windows Library demo
// program and must be compiled with OWL

#include <owl.h>
#include "tcwdll_1.h"
#include "\servo\servolib.h"  // header file for ServlLib DLL

int ErrorCode;

class TMotionApp : public TApplication {
public:
   TMotionApp(LPSTR Name, HINSTANCE hInstance,
         HINSTANCE hPrevInstance, LPSTR lpCmd,
               int nCmdShow)
               : TApplication(Name, hInstance,
                     hPrevInstance, lpCmd, nCmdShow) {};
   virtual void InitMainWindow();
};
```

```
class TTestWindow : public TWindow {
public:
   TTestWindow(PTWindowsObject AParent, LPSTR ATitle);
   virtual void CMCreate(TMessage& Msg) = [CM_FIRST + CM_CREATE];
};

TTestWindow::TTestWindow(PTWindowsObject AParent, LPSTR ATitle)
   : TWindow(AParent, ATitle)
{
   AssignMenu("COMMANDS");
}

void TTestWindow::CMCreate(TMessage&)
{
   InitializeServoSystem(800,808,&ErrorCode);
   EngageHighSpeedClock(&ErrorCode);
   SetServo(XAxis(),  1,&ErrorCode);
   T1AxisMoveTo(XAxis(),    0,&ErrorCode);
   T1AxisMoveTo(XAxis(), 5000,&ErrorCode);
   T1AxisMoveTo(XAxis(),10000,&ErrorCode);
   T1AxisMoveTo(XAxis(),    0,&ErrorCode);
   T1AxisMoveBy(XAxis(), 2000,&ErrorCode);
   T1AxisMoveBy(XAxis(),-4000,&ErrorCode);
   T1AxisMoveTo(XAxis(),    0,&ErrorCode);
   if (ErrorCode != 0) then
   {
         MessageBox(HWindow,"Motion Problem","Status",MB_OK);
         ErrorCode = 0;
   }
   MessageBox(HWindow,"Motion Done","Status",MB_OK);
   DisengageHighSpeedClock();
}

void TMotionApp::InitMainWindow()
{
   MainWindow = new TTestWindow(NULL, "Motion Application");
}

int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
   LPSTR lpCmd, int nCmdShow)
{
   TMotionApp MotionApp("Motion Application", hInstance, hPrevInstance,
         lpCmd, nCmdShow);
   MotionApp.Run();
   return(MotionApp.Status);
}
```

Explanation

After initializing the servo system the XAxis is directed to move to different destinations. It is extremely important that the dimension of the procedure match the dimension of the function, ie T1AxisMoveTo should only be used with a T1Axis parameter.

Note that, while motion is ocurring, Windows interaction comes to a halt. By using the MoveTo and MoveBy routines the program waits for the moves to finish. Although simple to synchronize, this approach is very limiting for an interactive application. The following examples show ways to not have this limitation.

C++ Example 2 - Direct Access

Description

This example illustrates how to perform other program activities during a motion.

Source Code

```
// Borland Turbo C++ Example 1
// The following example is based on a borland
// Object Windows Library demo program and must
// be compiler with OWL

#include <owl.h>
#include "tcwdll_2.h"
#include "servolib.h"  // header file for ServlLib DLL

int ErrorCode;
long MoveDurationCounter;

class TMotionApp : public TApplication {
public:
  TMotionApp(LPSTR Name, HINSTANCE hInstance,
           HINSTANCE hPrevInstance, LPSTR lpCmd,
           int nCmdShow)
           : TApplication(Name, hInstance,
                      hPrevInstance, lpCmd, nCmdShow) {};
  virtual void InitMainWindow();
};

class TTestWindow : public TWindow {
public:
  TTestWindow(PTWindowsObject AParent, LPSTR ATitle);
  virtual void CMStart(TMessage& Msg) = [CM_FIRST + CM_START];
};

TTestWindow::TTestWindow(PTWindowsObject AParent, LPSTR ATitle)
  : TWindow(AParent, ATitle)
{
  AssignMenu("COMMANDS");
}

void TTestWindow::CMStart(TMessage&)
{

   InitializeServoSystem(800,808,&ErrorCode);
   MoveDurationCounter=0;
   EngageHighSpeedClock(&ErrorCode);
   SetServo(XAxis(),  1,&ErrorCode);
   T1AxisBeginMoveBy(XAxis(),50000,&ErrorCode);
```

```
   MessageBox(HWindow,"Free To Continue","Status",MB_OK);

  while (MoveIsFinished(XAxis(),&ErrorCode) == 0)
    MoveDurationCounter++;

  if (ErrorCode != 0)
  {
   MessageBox(HWindow,"Motion Problem","Error",MB_OK);
   ErrorCode=0;
  }
  MessageBox(HWindow,"Motion Done","Status",MB_OK);
  DisengageHighSpeedClock();
}

void TMotionApp::InitMainWindow()
{
  MainWindow = new TTestWindow(NULL, "Motion Application");
}

int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            LPSTR lpCmd, int nCmdShow)
{
  TMotionApp MotionApp("Motion Application", hInstance, hPrevInstance,
         lpCmd, nCmdShow);
  MotionApp.Run();
  return(MotionApp.Status);
}
```

Explanation

As in the first example, the first operation is to initialize the system. Instead of performing motion with "MoveBy" and "MoveTo" the command "BeginMoveBy" is used. Commands that start with "Begin" imply that they do not wait for their operation to be complete. The move is started and program execution immediately continues. The program then performs other operations (in this case simply incrementing a variable) and periodically checks to see if the motion is completed by interrogating the servo system to find out when motion is complete.

Although a small improvement over the first example in terms of allowing other operations to continue during the motion program,

this is not a very complete solution either. Although other activities can occur during motion Windows is still "locked out" (i.e. try to move the window by dragging on the title bar during operation of the motion). The next example illustrates how to have an independent motion program operating at the same time as Windows interaction.

C++ Example 3 - Combined Access

Description

This example uses a seperate Douloi Pascal program to operate the XAxis while Windows continues operation in an independent manner.

Douloi Pascal Source Code

```
{Douloi Pascal Program for DLL Example 3}
{filename: ManyMove.dps}

procedure PerformManyMoves;
   begin
   XAxis.MoveTo(10000);
   XAxis.MoveTo(0);
   XAxis.MoveTo(20000);
   XAxis.MoveTo(0);
   XAxis.MoveTo(30000);
   XAxis.Moveto(0);
   XAxis.MoveTo(40000);
   XAxis.MoveTo(0);
   end;
```

Source Code

```
// Borland Turbo C++ Example 3
// The following example is based on a borland
// Object Windows Library demo program and must
// be compiler with OWL

#include <owl.h>
#include "tcwdll_3.h"
#include "servolib.h"  // header file for ServlLib DLL


int ErrorCode;
int Line;
int Row;

class TMotionApp : public TApplication {
public:
  TMotionApp(LPSTR Name, HINSTANCE hInstance,
           HINSTANCE hPrevInstance, LPSTR lpCmd,
           int nCmdShow)
           : TApplication(Name, hInstance,
                     hPrevInstance, lpCmd, nCmdShow) {};
```

```
  virtual void InitMainWindow();
};

class TTestWindow : public TWindow {
public:
  TTestWindow(PTWindowsObject AParent, LPSTR ATitle);
  virtual void CMStart(TMessage& Msg) = [CM_FIRST + CM_START];
};

TTestWindow::TTestWindow(PTWindowsObject AParent, LPSTR ATitle)
  : TWindow(AParent, ATitle)
{
  AssignMenu("COMMANDS");
}

void TTestWindow::CMStart(TMessage&)
{

   InitializeServoSystem(800,808,&ErrorCode);
   InitializeCompiler();
   Compile("..\\manymove.dps",NULL,&Line,&Row,&ErrorCode,NULL);
   if (ErrorCode != 0)
         {
         MessageBox(HWindow,"Compiler Problem","Error",MB_OK);
         return;
         }

   EngageHighSpeedClock(&ErrorCode);
   SetServo(XAxis(),1,&ErrorCode);
   BeginTask(TaskAddr(("PerformManyMoves"),&ErrorCode);

   if (ErrorCode != 0)
         {
         MessageBox(HWindow,"Motion Problem","Error",MB_OK);
         ErrorCode=0;
         }
   MessageBox(HWindow,"Task Started, Application and Windows free to do other
               things","Status",MB_OK);
}

void TMotionApp::InitMainWindow()
{
  MainWindow = new TTestWindow(NULL, "Motion Application");
}

int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
            LPSTR lpCmd, int nCmdShow)
{
   TMotionApp MotionApp("Motion Application", hInstance, hPrevInstance,
         lpCmd, nCmdShow);
   MotionApp.Run();
   DisengageHighSpeedClock(); /* shutdown system before leaving */
   return(MotionApp.Status);
}
```

Explanation

Examples 1 and 2 could have been done with a conventional PC based motion control card. This example is the point where Motion Server's architecture begins to make a contribution. The motion task is specified in the file ManyMove.dps (dps stands for Douloi Pascal Source). Note that the program is written in Douloi Pascal and not in the language of your DLL invoking host environment (i.e. not in TPW or C++ or VB).

This program is then compiled by your program at run-time through the Compile command implemented in the ServoLib DLL. The different parameters given to the Compile command help focus in on where errors are located in the event there is a mistake in the Douloi Pascal program. Note that InitializeCompiler needs to called before you use the Compile command, however it only needs to be called once at the beginning of the application regardless of the number of times Compile is used. Most of InitializeCompiler's activity is including the "system constants" that are described in the file STANDARD.INC. If you do not use these constants you can reduce the time delay of InitializeCompiler by deleting the symbols you do not need.

The host program then begins the Douloi Pascal program with the command BeginTaskAtCName (as distinct from a pascal style name). Strings in ServoLib are passed as null terminated, "ASCIIZ" strings, or "C style" strings. The motion program is now operating, running the motor, while the host Windows application is free to do whatever it would like. The motion system is quite independent of any particular Windows application once it has been started and does not require the application to remain present to operate. This degree of independence requires some thoughtful consideration since you would not want to start an operation and then lose the means of stopping it. Also note that an operating motion program will continue to operate even if the Windows application that started it experiences a UAE. For these reasons it is a good idea to hook up a hardware switch to one of the several User Disable inputs so as to be able to shutdown the motion system in the event that your Windows application does not perform as expected.

The procedure DisengageHighSpeedClock is called when the application finishes operating. This call should be made before you leave the application and definitely before you attempt to leave Windows or invoke a DOS shell. If you must go to DOS during the operation of the application use the Servo Ignition utility to manually turn off the high speed clock while you spend time in DOS.

Visual Basic DLL Examples

To make use of the ServoLib DLL from Visual Basic you will need to include the libraries declarations in the Global File of the application. The simplest way to do this might be to use Notepad to view the vb_inter.glo file provided in the \SERVO installation directory. Edit\Copy the entire file onto the Windows clipboard and paste it into the global file for your visual basic application.

The following examples illustrate how calls can be made to the motion system as well as the compiler.

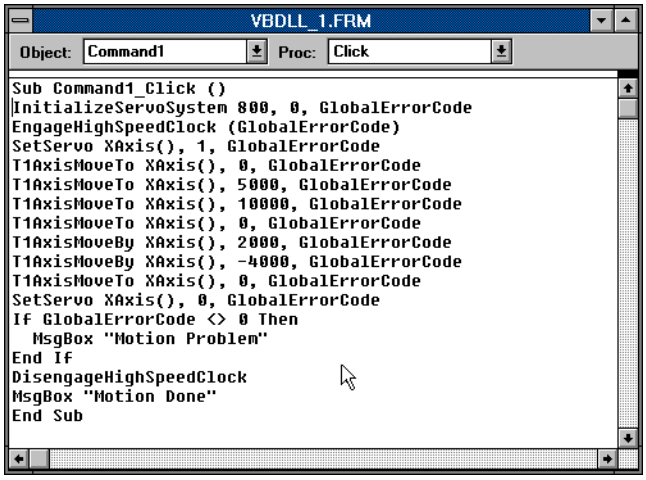
Visual Basic Example 1 - Direct Access

Description

This example uses direct calls to move the XAxis motor to several different locations and to write out commanded position information. This would represent the simplest use of ServlLib.

Source Code

The following was written in the click procedure of a single button on the default form.



Explanation

After initializing the servo system the XAxis is directed to move to different destinations. It is extremely important that the dimension of the procedure match the dimension of the function, ie T1AxisMoveTo should only be used with a T1Axis parameter.

Note that, while motion is ocurring, Windows interaction comes to a halt. By using the MoveTo and MoveBy routines the program waits for the moves to finish. Although simple to synchronize, this approach is very limiting for an interactive application. The following examples show ways to not have this limitation.

The Servolib DLL provides the procedure Stop(..), used to tell a T1Axis or TNAxis group to stop moving Visual Basic uses STOP as a keyword. To prevent a problem, the visual basic global file, used to import the library Stop(...) procedure, aliases the name to StopMotion to make it distinct from the STOP reserved word.

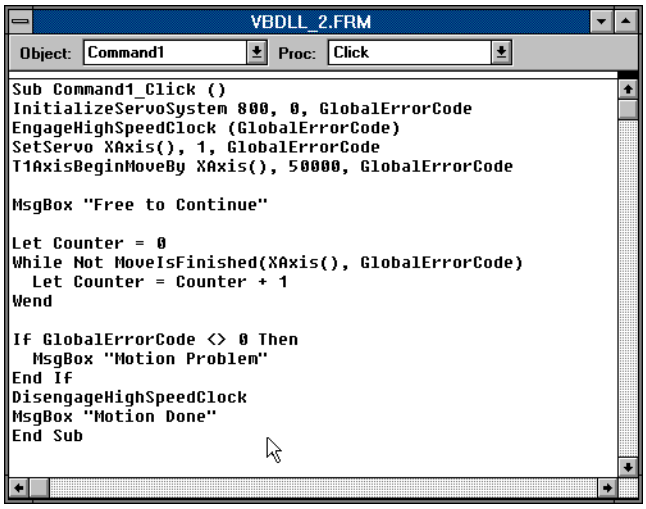
Visual Basic Example 2 - Direct Access

Description

This example illustrates how to continue with program execution during a motion.

Source Code

The following was written in the click procedure of a single button on the default form.



Explanation

As in the first example, the first operation is to initialize the system. Instead of performing motion with "MoveBy" and "MoveTo" the command "BeginMoveBy" is used. Commands that start with "Begin" imply that they do not wait for their operation to be complete. The move is started and program execution immediately continues. The program then performs other operations (in this case simply incrementing a variable) and periodically checks to see if the motion is completed by interrogating the servo system to find out when motion is complete.

Although a small improvement over the first example in terms of allowing other operations to continue during the motion program, this is not a very complete solution either. Although other activities

can occur during motion Windows is still "locked out" (i.e. try to move the window by dragging on the title bar during operation of the motion). The next example illustrates how to have an independent motion program operating at the same time as Windows interaction.

Visual Basic Example 3 - Combined Access

Description

This example uses a seperate Douloi Pascal program to operate the XAxis while Windows continues operation in an independent manner.

Douloi Pascal Source Code

```
{Douloi Pascal Program for DLL Example 3}
{filename: ManyMove.dps}

procedure PerformManyMoves;
   begin
   XAxis.MoveTo(10000);
   XAxis.MoveTo(0);
   XAxis.MoveTo(20000);
   XAxis.MoveTo(0);
   XAxis.MoveTo(30000);
   XAxis.Moveto(0);
   XAxis.MoveTo(40000);
   XAxis.MoveTo(0);
   end;
```

Source Code

The following code is in the click procedure of a button placed onto the default form.

```
Sub Command1_Click ()
InitializeServoSystem 800, 0, GlobalErrorCode
InitializeCompiler
Compile "..\ManyMove.dps", GlobalBookMark, GlobalRow, GlobalColumn,
  GlobalErrorCode, GlobalErrorString
EngageHighSpeedClock (GlobalErrorCode)
SetServo XAxis(), 1, GlobalErrorCode
BeginTask TaskAddr("PerformManyMoves"), GlobalErrorCode
If GlobalErrorCode <> 0 Then
  MsgBox "Motion Problem"
End If
MsgBox "Application and Windows free to do other things"
End Sub
```

Explanation

Examples 1 and 2 could have been done with a conventional PC based motion control card. This example is the point where Motion Server's architecture begins to make a contribution. The motion task is specified in the file ManyMove.dps (dps stands for Douloi Pascal Source). Note that the program is written in Douloi Pascal and not in the language of your DLL invoking host environment (i.e. not in TPW or C++ or VB).

This program is then compiled by your program at run-time through the Compile command implemented in the ServoLib DLL. The different parameters given to the Compile command help focus in on where errors are located in the event there is a mistake in the Douloi Pascal program. Note that InitializeCompiler needs to called before you use the Compile command, however it only needs to be called once at the beginning of the application regardless of the number of times Compile is used. Most of InitializeCompiler's activity is including the "system constants" that are described in the file STANDARD.INC. If you do not use these constants you can reduce the time delay of InitializeCompiler by deleting the symbols you do not need.

The host program then begins the Douloi Pascal program with the command BeginTaskAtCName (as distinct from a pascal style name). Strings in ServoLib are passed as null terminated, "ASCIIZ" strings, or "C style" strings. The motion program is now operating, running the motor, while the host Windows application is free to do whatever it would like, including terminating! The motion system is quite independent of any particular Windows application once it has been started and does not required the application to remain present to operate. This degree of independence requires some thoughtful consideration since you would not want to start an operation and then lose the means of stopping it. Also note that an operating motion program will continue to operate even if the Windows application that started it experiences a UAE. For these reasons it is a good idea to hook up a hardware switch to one of the several User Disable inputs so as to be able to shutdown the motion system in the event that your Windows application does not perform as expected.

Note that the program turns on the high speed clock, but does not turn it off. You must turn off the high speed clock yourself with the ServoIgnition utility, or more likely, in the close procedure of your Window applications Main Window by calling the routine DisengageHighSpeedClock. Make sure the "ignition is turned off" before leaving Windows or invoking a DOS shell.

SAW Implementation of DLL Examples

The examples motions that have been illustrated through DLL access from conventional languages systems are also presented in Servo Application Workbench. SAW is not accessible from another language system but rather is its own language system, more similar to Visual Basic than any of the other languages illustrated in this example set. This is included in the section on DLLs to provide a basis for comparison.

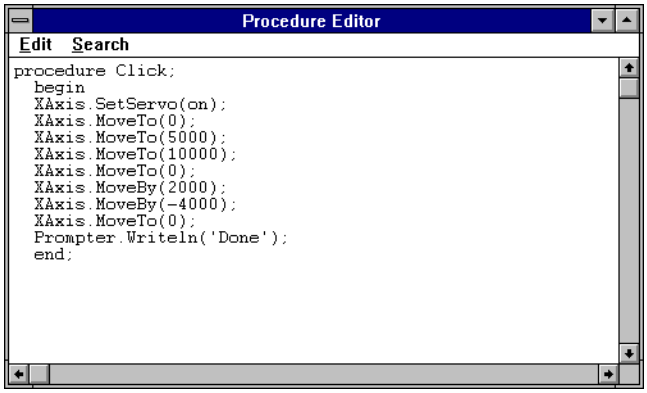
SAW Implementation of Example 1

Description

This example moves the XAxis motor to several different locations and to write out commanded position information.

Source Code

The following was written in the click procedure of a single button on the default plate.



Explanation

Initialization is handled for you when SAW first starts. It is not necessary to worry about the dimension of the method when used in the object format of Douloi Pascal as compared to the DLLs. If the wrong number of parameters is provided a compiler error will inform you of that. All TNAxis machines use the same procedure names, ie TNAxis machines of dimension 1 through 6 all know how to MoveTo a set of coordiantes. Each procedure is matched to its respectively dimensioned machine so as to collect the correct number of parameters. Explicit error checking is not required since a problem will genereate an escape and notify the operator directly. You have the ability to trap escapes to alter the error behavior.

Note that, while motion is ocurring, Windows interaction comes to a halt just like the DLLs. There many situations where it is important to not continue until a procedure is finished. Buttons do not allow Windows activity to occur until their procedures are finished. This is not restrictive, however, since buttons can have as their behaviors the scheduling of tasks.

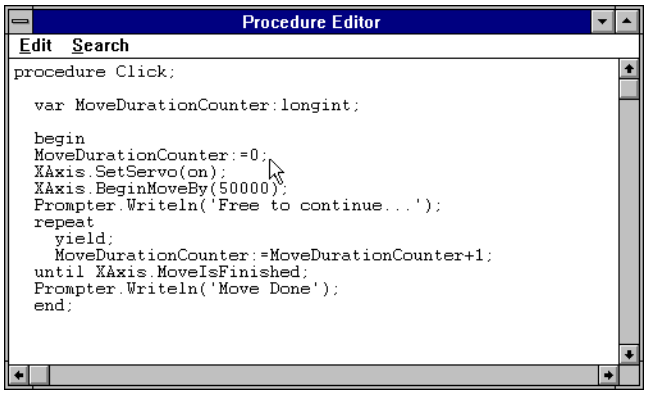
SAW Implementation of Example 2

Description

This example illustrates how to continue with program execution during a motion from within a single task.

Source Code

The following was written in the click procedure of a single button on the default form.



Explanation

This uses BeginMoveBy to start the motion and then performs other activities checking to see if the motion is complete.

Although a small improvement over the first example in terms of allowing other operations to continue during the motion program, this is not a very complete solution either. Although other activities can occur during motion Windows is still "locked out" (i.e. try to move the window by dragging on the title bar during operation of the motion). The next example illustrates how to have an independent motion program operating at the same time as Windows interaction.

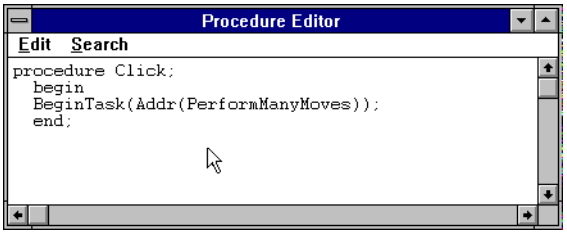
Saw Implementation of Example 3

Description

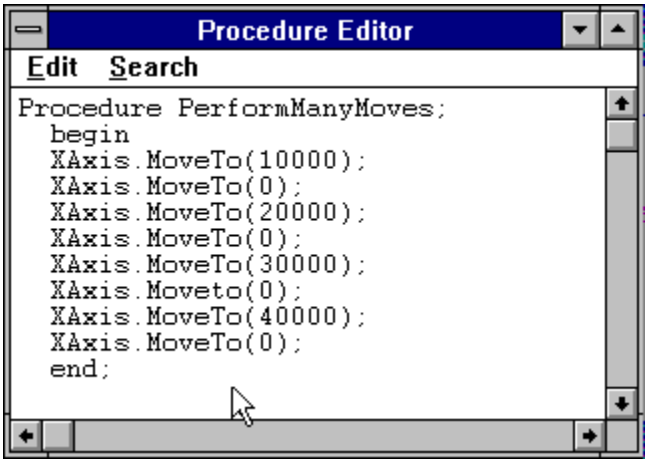
This example operates the XAxis while Windows continues operation in an independent manner.

Source Code

The follwoing code is in the click procedure of a button placed on the default plate.



The procedure PerformManyMoves is created as a plate procedure and has the same form as the Douloi Pascal ManyMove.dps file.



Explanation

By putting the XAxis activity in a seperate procedure {PerformManyMoves} and starting that procedure as a seperate task {BeginTask(Addr(PerformManyMoves)}the button is able to complete its job very quickly and allow execution to return to Windows. The operation of PerformManyMoves then proceeds on its own to completion.

Chapter 10

10) System Design Issues

Safety

Purpose

Motion control applications, among the broad range of application of computer technology, present some of the most physical consequences to software quality to be found. It is entirely possible to ask a machine to perform an action which may not be safe to perform. Safety, primarily of the operators and people near equipment, and secondarily of the equipment itself, should be part of any motion control application design from the beginning. This section describes different design issues you may want to consider to enhance the safety of your application.

Limitations of Application

Douloi Automation does not authorize the use of Motion Server and/or SAW for any application which involves human life support activity where the failure of the motion system could endanger the lives of people.

Responsibility

You are ultimately responsible for the safety consequences of a machine controlled through Motion Server and SAW. There is no way for Douloi Automation to inspect and assume responsibility for the day-to-day operation of a machine being controlled by Motion Server and/or SAW. What constitutes safe behavior is application dependent. The decision of what constitutes an error situation and the appropriate responses to that error situation are in the problem and knowledge domain of the user of the motion system, and are appropriately your responsibility. Douloi Automation is available to help advise and direct you in the creation of safety responses by the motion control system however Douloi Automation assumes no responsibility.

Built-In Safety Features

Motion Server provides several features to assist in creating a safe system. The primary feature with respect to system integrity is the dual Watchdog Safety System. This is a hardware system in the Motion Server hardware which continually confirms that the hardware is receiving its due attention from the on-board processor. If for any reason the processor does not participate in the motion control relationship which Motion Server hardware expects, the watchdog hardware disables the amplifiers and shuts off the servos. Examples of failures that this system would protect against include software failures in Motion Server or SAW.

Motion Server provides a safety feature which is useful during motion. Axis which are servoing are expected to have their ActualPosition close to the CommandedPosition. If there is a large difference between the two there most likely is some problem. This difference, the ErrorPosition, is monitored by Motion Server. If an axis is supposed to be servoing, and the ErrorPosition is greater than the ErrorLimit of an axis then the axis is disabled. Note that the check really only occurs during motion. Although the check is performed while an axis is stationary there is no way for Motion Server to distinguish between a motor correctly holding position and a failed encoder. Both report 0 position change.

Motion Server hardware provides amplifier enable lines, one for each axis in the system. These digital signals are connected to the enable input of an amplifier to indicate to the amplifier when it should regard the analog input. If the amplifier enable is "high" the amplifier provides power as directed by the analog input. If the level is "low" the amplifier provides no power regardless of the analog input. These affords two different ways for the control system to request 0 amplifier output, through a 0 analog voltage and as well through a digital signal. Typically the amplifier enable signal is "pulled up" internally to allow the amplifier to be enabled as the default condition making the use of the amplifier enable line optional. However, if the amplifier provides for an amp enable signal you are advised to use it to enhance the safety of the system.

Chapter 10

Limit Switches

Limit Switches are often provided on equipment to represent "end of mechanical travel". Limit switches typically are used to shut down a system since in normal operation you should never tell the machine to go to its very limit. Sometimes limit switches are used for initialization to establish the mechanical coordinate system.

Motion Server does not provide built-in support for limit switches since Douloi Automation regards limit switch behavior to be application dependent. However there is a library of limit switch behaviors available which can possibly be used directly or more likely modified to suit your particular needs. These limit switch behaviors are available through the LIMITS catalog. The behaviors "merge" to your application's main plate and automatically start operation through the SETUP procedure which is part of the limit switch behavior. You can browse through the LIMITS catalog to determine if there is a limit switch behavior suitable for your application.

Emergency Stop Considerations

If you provide a way to start a machine you should provide at least one way to stop it. You will need to determine what type of stop is appropriate. Although it may only seem necessary to provide a "stop at end of cycle" it is generally advisable to provide some means to stop the machine much sooner in the event of machine motion endangering someone.

You can include an abort button in a SAW application. Although this is a good thing to do don't expect someone to perform precision mouse clicks during a crisis. A better solution is a hardware button. The large, red, round, latching "mushroom" buttons are very suitable. Their purpose is almost universally understood to mean "emergency stop" even without a descriptive legend. You can swing at such a button and successfully hit it much quicker than finding the mouse and clicking a Windows control.

This button should be connected to the EStop input. This can be accessed with a 6 pin plug or by placing the jumper on pins 1 and 2, the pins closest to the Motion Server mounting bracket. In this latter case, input bit 1 on the general purpose IO cable becomes the EStop input. The EStop input is used to make a normally closed continuous loop to ground. If any EStop button in the loop is opened, the Motion Server hardware shuts down power to the drive regardless of what the software is requesting.

The watchdog system always turns the servos off. The interpretation of a watchdog event is that controller integrity has been lost. Without the controller operating correctly the servos are basically not being controlled at all and any expressed request for power is potentially dangerous.

An emergency stop, on the other hand, represents not a loss of system integrity but some insight from an operator that the machine needs to stop. Accordingly it is possible to consider an emergency shutdown plan that gracefully decelerates the motors etc. to perform a stop. The more conservative choice is to leave the User Disable jumper in place and simply turn the motors off. However coasting motors or motors dropping from a gravity load may pose a greater safety hazard than leaving the motors active.

Note that regardless of whether the User Disable jumper is in place or not, you must have a routine which notices the level of this line and takes action, i.e. telling the machine to abort or stop. If you remove the User Disable jumper, the input has no effect at all on the hardware. By removing the jumper you are taking full responsibility to determine what should happen during an emergency stop and scheduling a task to notice that the line has asserted so as to start that task.

All three user disable inputs are "or-ed" together, that is to say that closing any switch to ground will result in the same effect. You may also use any of the remaining 24 input lines as "emergency stop" signals by including them in the routine which is scheduled to check for system integrity.

Safety is important. If you have any questions about how to design safety into your system please contact Douloi Automation for recommendations. This type of support comes with the purchase price of a controller so please do not hesitate to call.

Initialization

Purpose

Most motion control systems are "incremental" motion control systems. These systems can precisely change positions but do not have any fundamental "hardware" knowledge of where they actually are. They only know their change in position from some initial starting condition. The purpose of this section is to help direct you through design choices that will allow your machine to repeatably know where it is after start-up so as to insure consistent operation of your motion application.

Traditional Homing Strategy

"Homing" is used to describe a process which a machine goes through so as to precisely know its position. Typically this process is done once, when the machine is first turned on. Once the control system knows where the machine is position information should be reliably retained automatically.

The homing process typically uses homing sensors. These sensors "trip" at some particular point in the workspace, often near the limits of motion. The machine is commanded, in its uninitialized state, to move in a particular direction known to be towards the home sensor. When the sensor trips the machine has gained some reference information about where it physically is. Because home sensors are usually not very precise the home sensors are used in conjunction with the index pulse on a 3 channel encoder to provide high performance initialization. Effectively, the home sensor is able to get the machine to within a motor rev of the home position, and the index pulse is used to identify what particular count within that motor rev. The combination of these two provides repeatability, to within a count, of machine operation from one power-on cycle to the next.

The process usually is performed by initially moving at a moderate speed for the "corner" where the home sensors are, and stopping upon finding the home sensors. The machine then travels in the reverse direction at a very slow rate waiting for the home sensors to "untrip". At that point the machine then performs a relative move by a small amount, determined during the machine's initial construction, so as to optimize the location of the index marker on the encoder. The machine then moves at a slow speed and waits for the index pulses to occur. At this point the position of the machine is accurately known.

The additional move between the "untrip" of the home switch and the search for the index pulse is provided to maximize the likelihood of finding the correct motor rev. The home switch basically gets the machine to within a motor rev and the index pulse to within a count. However it would be possible in certain situations to select the wrong motor rev. Imagine that after the home switch has "untripped" that the index pulse is currently active, i.e. you happen to be right on the index pulse. If you had untripped a small amount sooner, you would immediately come upon the index pulse as you advanced from this point. If, however, the home switch had untripped a moment later you would have already missed the index pulse and wouldn't find it again for an entire rev. This ambiguity of an entire motor rev is a large uncertainty. To avoid this uncertainty the machine should start the search for the index pulse one-half motor rev back from where it actually is. Then the home switch has an entire plus or minus half rev of tolerance to miss with and still pick the right motor rev. The delta move, then, is chosen when the machine is first assembled so as to be in the neighborhood of half a rev away. The alternative is to mechanically set the location of the index pulse to be half a motor rev away from the home position however such mechanical adjustment is usually inconvenient when changing a number in the program accomplishes the same result. You can setup an offset file that is read during initialization to contain the machine-specific calibration numbers so that the software running a series of machines can all be identical.

Using high speed position capture it is possible for the initialization process to occur at a higher speed since the capture can be used for both the home input and index pulse allowing the initialization process to significantly speed up. However, since initialization is normally performed just once during the day it usually is not very important for initialization to be fast.

Motion Server does not have any built-in home behaviors however you may choose a home behavior from the HOME catalog which includes the above procedure as well as some others. Please contact Douloi Automation for guidance on initialization if you have further questions.

Chapter 11

11) Command Summary

Purpose

The following is a list of all of the commands available. Additional details about the commands can be found in the on-line help.

Primitive Data Types

Integer ..... 16 bit numeric data type
Longint ..... 32 bit numeric data type
Single ..... 32 bit floating point number
Double ..... 64 bit floating point number
String ..... 31 character variable for holding letters
Boolean ..... true/false type
TMathCoprocessorBuffer ..... Temporary storage area type for math coprocessor state

TNVector Objects - Multidimensional Vectors with N ranging from 2 to 6

Init ..... Set the components of the vector to the indicated values
Length ..... Return the magnitude of the vector
XComp ..... Returns x component of vector (DLL only)
YComp ..... Returns y component of vector (DLL only)
ZComp ..... Returns z component of vector (DLL only)
UComp ..... Returns u component of vector (DLL only)
VComp ..... Returns v component of vector (DLL only)
WComp ..... Returns w component of vector (DLL only)

Math Coprocessor Operations (Douloi Pascal only)

FInit ..... Initialize math coprocessor
FAdd ..... Add top two numbers
FSin ..... Replace the top number with its sin
FSinCos ..... Replaces the top number with its sine and cosine
FSqrt ..... Replace the top number with its square root
FLdPi ..... Push Pi onto the coprocessor stack
FDiv ..... Replace the top two numbers by top-1 divided by the top
FMul ..... Replace the top two numbers with their product
FChs ..... Change the sign of the top most number
FPaTan ..... Replace the top two numbers with the arctan of top-1 / top
FSave ..... Stores the coprocessor state into a MathCoprocessorBuffer
FRestore ..... Replaces a previously filled MathCoprocessorBuffer
PopLongint ..... Transfer the top number into the specified longint variable
PopSingle ..... Transfer the top number into the specified single variable

PopDouble ..... Transfer the top number into the specified double variable
PushLongint ..... Transfer the specified longint onto the coprocessor stack
PushSingle ..... Transfer the specified single onto the coprocessor stack
PushDouble ..... Transfer the specified double onto the coprocessor stack

Multitasking

BeginTask ..... Spawn a new real-time execution thread
ScheduleTask ..... Periodically spawn a new real-time execution thread
AbortTask ..... Stop the execution of a task currently running
SuspendTask ..... Temporarily prevent a task from running
ResumeTask ..... Allow a suspended task to continue executing
ExecuteTask ..... Perform task and wait for completion before continuing (DLL only)
SetIgnition ..... Turns on and off high speed multitasking
Yield ..... Give other tasks a chance to run (Douloi Pascal only)
Delay ..... Suspends task for specified number of samples
TaskOveranSample ..... Returns true if a task took too much time. (Douloi Pascal only)
CheckTask ..... Causes an escape if a task took too much time (Douloi Pascal only)
TimeRemaining ..... Amount of time remaining in sample period (Douloi Pascal only)
ExecuteWindowsApplication ..... Launch a Windows application (SAW only)
TaskAddr ..... Provides address of task procedure for use by other calls

IO Operations

Motion Server IO
SetOutputEnable ..... Tell Motion Server compare output bits to become active
SetOutputBit ..... Change state of output bit on Motion Server hardware
InputBit ..... Returns the true/false value of an input signal

General Expansion Bus IO - Douloi Pascal Only
PortWriteByte ..... Write to third party IO space card in 8 bit manner
PortWriteWord ..... Write to third party IO space card in 16 bit manner
PortReadByte ..... Read from third party IO space card in 8 bit manner
PortReadWord ..... Read from third party IO space card in 16 bit manner
MemWriteByte ..... Write to third party memory space card in 8 bit manner
MemWriteWord ..... Write to third party memory space card in 16 bit manner
MemReadByte ..... Read from memory space card in 8 bit manner
MemReadWord ..... Read from memory space card in 16 bit manner

Safety

ResetWatchdog ..... Allow tripped safety system to resume servo activity
WatchdogHasTripped ..... Returns status of watchdog system
UserHasDisabled ..... Indicates if any disable input is asserted

Numeric

Sin ..... Returns sin of parameter in degrees (SAW only, not real time)
Cos ..... Returns cos of parameter in degrees (SAW only, not real time)
ArcTan ..... Returns arctan of parameter as degrees (SAW only, not real time)
Sqrt ..... Returns square root of parameter
HighInteger ..... Retreives the upper 16 bits of a 32 bit longint {Douloi Pascal only}
LowInteger ..... Retreives the lower most 16 bits of a 32 bit longint {Douloi Pascal only}

Exception Handling

Escape ..... Generates an exception to {Douloi Pascal only}
WriteVerboseEscapeCode ..... Displays text explanation of escape code (SAW only)
EscapeCode ..... Returns current escape value {Douloi Pascal only}

TPlate Objects - Assembly Foundations/Drawing Surfaces (SAW only)

Close ..... Causes plate created with POPUP to go away
Popup ..... Causes plate to be created, i.e. a dialog box
Update ..... Causes drawing activities to be updated and take effect
Clear ..... Erases all graphics on plate
ClearLast ..... Removes last graphic added to plate
DrawLine ..... Draws line between two specified points
DrawRectangle ..... Draws a rectangle specified by two corner points
DrawDiamond ..... Draws diamond inscribed in specified rectangle
DrawEllipse ..... Draws ellipse inscribed in specified rectangle
DrawRoundedRectangle ..... Draws rectangle with rounded corners
SetCoordinateFrame ..... Establishes coordinate frame for drawing
SetLineStyle ..... Sets the outline style of the next graphic created
SetLineColor ..... Sets the outline color for the next graphic created
SetBodyColor ..... Sets the fill or "body" color of the next graphic
Fit ..... Sets coordinates to fit around data in array
Plot ..... Draws data in array
Enclose ..... Stretches coordinate frame to include array
GetOpenFilename ..... Provides filename dialog to get a filename for opening
GetSaveFilename ..... Provides filename dialog to get a filename for saving

TStatic Object - Static Text/Display Object (SAW only)

Read ..... Read information from display
Readln ..... Read information from display
Write ..... Write information to display
Writeln ..... Write information to display
Clear ..... Erases text
SetDecimalPlace ..... Specifies output format for single numbers

TEditor Object - Single Line Text Editor (SAW only)

Read ..... Read information from editor
Readln ..... Read information from editor
Write ..... Write information into editor
Writeln ..... Write information into editor
Clear ..... Erases text
SetDecimalPlace ..... Specifies output format for single numbers

TListBox Object - List Box Text Selection Object (SAW only)

Read ..... Read information from list box
Readln ..... Read information from list box
Write ..... Add information to list box
Writeln ..... Add information to list box
Clear ..... Resets list to contain no items
SetDecimalPlace ..... Specifies output format for single numbers
SetSelectionIndex ..... Preset list box selection to indicated index
GetSelectionIndex ..... Returns the index of the current list box selection

TFile Object - DOS File Access Object (SAW only)

Assign ..... Associates TFile to DOS filename
Close ..... Concludes associated established with Assign
Rewrite ..... Prepares file for writing
Reset ..... Prepares file for reading
EndOfFile ..... Returns true if no more information is in file

THPGLFile Object (SAW only)

Assign ..... Associates TFile to DOS filename
Close ..... Concludes associated established with Assign
Rewrite ..... Prepares file for writing
Reset ..... Prepares file for reading
EndOfFile ..... Returns true if no more information is in file
ReadCommand ..... Returns HPGL command information

TPrompter Object - Message Box Object (SAW only)

Init ..... Creates Windows mechanism for prompting
Write ..... Write information into prompter
Writeln ..... Write information into prompter and interact with user

TNAxis Object - Multi-Axis Motion Object

Configuration
Init ..... Associate object with a system axis
SetMotorType ..... Configures motor for servo or stepper operation
SetEnable ..... Allow amplifier to power motor

SetMotor ..... Turns motor operation on and off
SetLoopInversion ..... Include an additional sign inversion in control law
SetCoordinateInversion ..... Reverse which way is regarded as the positive direction
SetAccel ..... Set acceleration rate for trapezoidal moves
SetDecel ..... deceleration rate for trapezoidal moves
SetSpeed ..... Set speed of slew phase of trapezoidal moves
SetGain ..... Set compensation parameter for servo
SetZero ..... compensation parameter for servo
SetIntegrator ..... compensation parameter to eliminate steady state position error
SetErrorLimit ..... Set permissible tracking error before disable occurs
SetPositiveLimit ..... Set boundary for movement in the positive direction
SetNegativeLimit ..... Set boundary for movement in the negative direction
SetActualPosition ..... Define current position coordinate
SetCommandedPosition ..... Set commanded position for non-trapezoidal moves
SetCommandedPositionVector ..... Set commanded position for group
ArmInputCapture ..... Prepares axis to latch position based on input signal
ArmIndexCapture ..... Prepares axis to latch position based on index signal
SetCommandedTorque ..... Set output voltage when not servoing

Motion
MoveTo ..... Move to absolute coordinate
MoveBy ..... Move to relative coordinate
BeginMoveTo ..... Start move to absolute coordinate
BeginMoveBy ..... Start move to relative coordinate
MoveToVector ..... Perform absolute coordianted move to vector parameter
MoveByVector ..... Perform relative coordinated move by vector parameter
BeginMoveToVector ..... Start absolute coordinated move to vector parameter
BeginMoveByVector ..... Start relative coordinated move by vector parameter
MoveAlongCurve ..... Perform coordinated multiaxis motion along curve
BeginMoveAlongCurve ..... Begin coordinated curved motion
AppendMoveTo ..... Add absolute vector segment to curve description
AppendMoveBy ..... Add vector segment to curve description relative to last segment
AppendArc ..... Add circular or helical arc description to continuous path curve
Clear ..... Erase any established motion curve info
LinkTo ..... Associate TNAxis with vector array of same dimension
Jog ..... Move indefintely at constant speed
Stop ..... Gently stops any motion that may be in progress
BeginStop ..... Begins to stop but immediately does next instruction
Abort ..... Suddenly aborts any motion that my be in progress

Query
Gain ..... Return current compensator gain value
Zero ..... Return current compensator zero value
Integrator ..... Return current compensator integrator value
Accel ..... Return current acceleration parameter in counts per second squared
Decel ..... Return current deceleration parameter in counts per second squared
Speed ..... Return current speed in counts per second

ActualPosition ..... Return current actual motor position
CommandedPosition ..... Return ideal or target position for motor
DestinationPosition ..... Return absolute coordinate of end of move
ErrorPosition ..... Return discrepency between current and ideal position
CapturePosition ..... Return position recorded when latch event occurred
GetActualPositionVector ..... Fill in vector with actual axis coordinates
GetCommandedPositionVector ..... Fill in vector with ideal coordinates
GetErrorPositionVector ..... Fill in vector with discrepencies between actual and ideal locations
MoveIsFinished ..... Return true if move has finished
CommandedTorque ..... Return current analog output value
ProfileVelocity ..... Return current ideal profile velocity
CaptureHasTripped ..... Indicate if latch event has occurred
MotorIsOn ..... Return true if motor is currently powered and active
EnableIsOn ..... Return true if amplifier is powered

Escape Code Constants

DivideBy0EscapeCode
SampleOverrunEscapecode
AxisNotValidEscapecode
SpeedNegativeEscapeCode
NotImplementedEscapeCode
CurveBufferNotLinkedEscapeCode
UserAccelIs0OrNegativeEscapeCode
UserDecelIs0OrNegativeEscapeCode
AxisNotAvailableEscapeCode
UnableToResumeTaskEscapeCode
UnableToBeginTaskEscapeCode
AppendMoveToOverflowEscapeCode
InsufficientNumberOfSegmentsEscapeCode
MotionOverRunEscapeCode
AxisIsBusyEscapeCode
FileResetEscapeCode
FileRewriteEscapeCode
ReadEscapeCode
WriteEscapeCode
ObjectNotInitializedEscapeCode
ConversionErrorEscapeCode
ParameterOutOfRangeEscapeCode
WatchdogFailedToResetEscapeCode
OptionNotPresentEscapeCode

Mathematical Constants

pi

Boolean Constants

True
False
On
Off

Torque Descriptions

MaxTorque= 2039;
MinTorque= -2040;

Pen line styles

Solid =0;
Dash =1;
Dot =2;
DashDot =3;
DashDotDot=4;

Pen colors

Black =0;
Red =1;
Green =2;
Blue =3;
Yellow =4;
Magenta=5;
Cyan =6;
White =7;

HPGL Command Constants

HPGLUndefinedCommand=0;
HPGLEndOfFile=1;
HPGLInit=2;
HPGLSC=3;
HPGLSelectPen=4;
HPGLPenUp=5;
HPGLPenDown=6;
HPGLPlotAbsolute=7;

Chapter 12

12) Cables and Connectors

Description

Cabling to Motion Server is performed through flat ribbon cables terminated with IDC connectors.

Axis Group Connectors

There are (4) 60 pin connectors for axis information called "Axis Group" connectors. Each 60 pin ribbon cable supports (4) axis of signals. The 60 pin ribbon cable can be split apart into (4) identical 15 pin axis sub-cables. The signals have been chosen in a very regular pattern so that all of the 15 pin sub-cables are identical in layout.

I/O Connector

There is (1) 50 pin connector containing 48 bits of configurable I/O. Signals are configured as input or output in 4 bit groups.

E-Stop Connector

There is (1) 6 pin header used to configure E-Stop with a jumper or to cable to EStop. The jumper can be used to disable E-Stop, connect I/O signal 1 to be E-Stop, or can serve as a cable connector for an external E-Stop cable assembly.

External Bus Connector

There is (1) 26 pin connector which supports an external 8 bit bus allowing Motion Server to control additional hardware elements.

Axis Signal Descriptions

Encoder A+, A-, B+, B-, I+, I-

Functional Description

Encoder signals provide position feedback from a rotary or linear encoder. In general these signals are provided in a "quadrature" format indicating both position and direction change. Encoders are necessary for servo motors and optional for stepper motors. Differential signals are desirable demonstrating improved noise immunity, however single-ended encoders may also be used. When using a single ended encoders connect the signals to the "+" inputs. The "-" inputs have a "pull-center" resistors connecting the "-" inputs to the differential receivers to a 2 volt reference. This provides a default "-" signal level in the absence of the actual signal. In certain rare cases it may be necessary to change this default reference value. This can be done by removing or switching an resistor network which is socketed on the board. Consult Douloi Automation before attempting any change.

The "I" signal is the index pulse for an optical encoder. This signal can be used for higher speed, more repeatable homing, or for encoder-drift detection.

Electrical Description

Encoder signals go into a 3486-style differential receiver. The receivers are rated for a maximum differential mode voltage of +/- 25 volts and common mode voltage of +/- 15 volts. In most cases the encoder signals are 5 volt signals.

Amp Enable High, Amp Enable Low

Functional Description

The amplifier enable signal is a digital output which allows the motor amplifier to apply power. If the amplifier is not enabled, the amplifier will not produce motor current regardless of the level of the motor command voltage. Different amplifiers have different conventions for what "enable" means. Some apply power if the signal is a high logic level. Some apply power on a low logic level. To accommodate these differences both a high and a low level signal are provided. Review amplifier documentation to learn which level is required. Douloi Automation recommends setting the amplifier (if the option is available) to be inactive until a deliberate amp enable signal is provided by the controller. Providing both a high and low level signal places the decision of amplifier enable sense into the machine wiring harness, not an on-controller jumper which could be misconfigured.

Electrical Description

The amplifier enable signals are driven by a 74LS07 with open collector outputs.

Position Capture

Functional Description

The Position Capture input can be used for high-speed registration applications. The position of the encoder is recorded in hardware in response to a position capture signal. Most often the signal is used as a homing input. Even without an encoder, the level of the signal can be monitored in software with the CaptureBit command.

Electrical Description

The Position Capture signal is the "+" side of a 3486 differential receiver. The "-" side of the receiver goes to a 2 volt reference. Standard TTL level can be used and voltage up to 24 volts maximum can be tolerated. There is no on-board pullup resistor for this input. If the sensor being used is an open-collector style drive, a 4.7k pullup resistor to +5 volts (available on the axis connector) should be used.

Position Compare

Functional Description

Position Compare is an output signal that is set when the encoder hardware detects a specific encoder position. The output can also be used as a general purpose output.

Electrical Description

Position Compare is a TTL level output with a 12 ma sink and approximately no source capability. This signal is the most "exposed" signal on the axis connector set coming directly from a FPGA device on the board with no additional buffering or protection.

Motor Command

Functional Description

The Motor Command signal is a +/- 10 volt signal most often used to specify requested current from a servo motor amplifier. The signal can also represent requested voltage or velocity depending on the amplifier mode selected. In most cases torque mode is most suitable

Electrical Description

The Motor Command signal is +/- 10 volts with 3 ma drive. Many amplifiers have differential receivers. In this case, use the motor command signal on the "+" side of the receiver and ground (from the axis cable set) on the negative side. Providing Motor Command and ground in a twisted pair can improve noise immunity.

Step Pulse, Direction

Functional Description

Step Pulse and Direction signals are used for controlling stepper motors. Standard firmware supports narrow (1 microsecond) step pulses. Alternate firmware for 4-axis controllers is available for supporting wide step pulses (30 microseconds) if the stepper driver is unable to respond to narrow pulses.

Electrical Description

Step Pulse and Direction signals are open collector outputs driven by a 74LS07.

+5 Volts, Ground

Description

+5 Volts and Ground are available for providing encoder power, sensor power, and pull-up references. These signals come directly from the PC's power supply.

Pin Numbering Conventions

There are two different connector styles most often used with the Motion Server Controller. The first is "2-row IDC" style connectors, which are the style commonly used with computer disk-drive cabling etc. In this convention, the pin number corresponds to the wire number, counting sequentially from the end of the wire. This produces a "back and forth" counting pattern on the IDC connector.

The other connector often used is a D subminiature style. This connector has a pin definition which can often be read on the connector itself. Small, inscribed numbers next to the pins indicate that the pin numbering is sequential along the length of the connector, and then resumes at the beginning of the next row. This is quite different from the "back and forth" convention of the 2 row IDC connector. It is most convenient to use D connectors by "splitting apart" the 60 pin IDC cable and then crimping IDC style D connectors. NOTE THAT THE PIN NUMBERING CONVENTION FOR D-CONNECTORS ATTACHED TO THE RIBBON CABLE IS DIFFERENT THAN THE IDC 2-ROW CONVENTION FOR THE CABLE ITSELF. Please refer to the proper table when preparing to wire to the controller.

Axis Group Connector Definitions, 2-Row IDC

The following Table defines the connectors for the axis groups. These connectors are designated "Axis 1-4", "Axis 5-8", "Axis 9-12", and "Axis 13-16" on the printed circuit board silk screen. The signal definitions is a regular pattern both along the connector, and from one connector to the next. For example, Pin 3 is always an Encoder B+ signal with the axis defined by which connector the pin is on. Each pin in any particular connector has 3 other counterparts spaced a multiple of 15 away. For example, pin 18 (pin 3 + 15) is also an Encoder B+ signal as well as pin 33 (pin 3 +30) and pin 48 (pin 3 + 45)

Pin Number	Description	Axis 1-4	Axis 5-8	Axis 9-12	Axis 13-16
1	Encoder A+	Axis 1	Axis 5	Axis 9	Axis 13
2	Encoder A-	Axis 1	Axis 5	Axis 9	Axis 13
3	Encoder B+	Axis 1	Axis 5	Axis 9	Axis 13
4	Encoder B-	Axis 1	Axis 5	Axis 9	Axis 13
5	Encoder I+	Axis 1	Axis 5	Axis 9	Axis 13
6	Encoder I-	Axis 1	Axis 5	Axis 9	Axis 13
7	Amp Enable High	Axis 1	Axis 5	Axis 9	Axis 13
8	Amp Enable Low	Axis 1	Axis 5	Axis 9	Axis 13
9	Position Capture	Axis 1	Axis 5	Axis 9	Axis 13
10	Position Compare	Axis 1	Axis 5	Axis 9	Axis 13
11	Motor Command	Axis 1	Axis 5	Axis 9	Axis 13
12	Step Pulse	Axis 1	Axis 5	Axis 9	Axis 13
13	Direction	Axis 1	Axis 5	Axis 9	Axis 13
14	+5 Volts	Axis 1	Axis 5	Axis 9	Axis 13
15	Ground	Axis 1	Axis 5	Axis 9	Axis 13
16	Encoder A+	Axis 2	Axis 6	Axis 10	Axis 14
17	Encoder A-	Axis 2	Axis 6	Axis 10	Axis 14
18	Encoder B+	Axis 2	Axis 6	Axis 10	Axis 14
19	Encoder B-	Axis 2	Axis 6	Axis 10	Axis 14
20	Encoder I+	Axis 2	Axis 6	Axis 10	Axis 14
21	Encoder I-	Axis 2	Axis 6	Axis 10	Axis 14
22	Amp Enable High	Axis 2	Axis 6	Axis 10	Axis 14
23	Amp Enable Low	Axis 2	Axis 6	Axis 10	Axis 14
24	Position Capture	Axis 2	Axis 6	Axis 10	Axis 14
25	Position Compare	Axis 2	Axis 6	Axis 10	Axis 14
26	Motor Command	Axis 2	Axis 6	Axis 10	Axis 14
27	Step Pulse	Axis 2	Axis 6	Axis 10	Axis 14
28	Direction	Axis 2	Axis 6	Axis 10	Axis 14
29	+5 Volts	Axis 2	Axis 6	Axis 10	Axis 14
30	Ground	Axis 2	Axis 6	Axis 10	Axis 14

Pin Number	Description	Axis 1-4	Axis 5-8	Axis 9-12	Axis 13-16
31	Encoder A+	Axis 3	Axis 7	Axis 11	Axis 15
32	Encoder A-	Axis 3	Axis 7	Axis 1	Axis 15
33	Encoder B+	Axis 3	Axis 7	Axis 11	Axis 15
34	Encoder B-	Axis 3	Axis 7	Axis 11	Axis 15
35	Encoder I+	Axis 3	Axis 7	Axis 11	Axis 15
36	Encoder I-	Axis 3	Axis 7	Axis 11	Axis 15
37	Amp Enable High	Axis 3	Axis 7	Axis 11	Axis 15
38	Amp Enable Low	Axis 3	Axis 7	Axis 11	Axis 15
39	Position Capture	Axis 3	Axis 7	Axis 11	Axis 15
40	Position Compare	Axis 3	Axis 7	Axis 11	Axis 15
41	Motor Command	Axis 3	Axis 7	Axis 11	Axis 15
42	Step Pulse	Axis 3	Axis 7	Axis 11	Axis 15
43	Direction	Axis 3	Axis 7	Axis 11	Axis 15
44	+5 Volts	Axis 3	Axis 7	Axis 11	Axis 15
45	Ground	Axis 3	Axis 7	Axis 11	Axis 15
46	Encoder A+	Axis 4	Axis 8	Axis 12	Axis 16
47	Encoder A-	Axis 4	Axis 8	Axis 12	Axis 16
48	Encoder B+	Axis 4	Axis 8	Axis 12	Axis 16
49	Encoder B-	Axis 4	Axis 8	Axis 12	Axis 16
50	Encoder I+	Axis 4	Axis 8	Axis 12	Axis 16
51	Encoder I-	Axis 4	Axis 8	Axis 12	Axis 16
52	Amp Enable High	Axis 4	Axis 8	Axis 12	Axis 16
53	Amp Enable Low	Axis 4	Axis 8	Axis 12	Axis 16
54	Position Capture	Axis 4	Axis 8	Axis 12	Axis 16
55	Position Compare	Axis 4	Axis 8	Axis 12	Axis 16
56	Motor Command	Axis 4	Axis 8	Axis 12	Axis 16
57	Step Pulse	Axis 4	Axis 8	Axis 12	Axis 16
58	Direction	Axis 4	Axis 8	Axis 12	Axis 16
59	+5 Volts	Axis 4	Axis 8	Axis 12	Axis 16
60	Ground	Axis 4	Axis 8	Axis 12	Axis 16

Axis Group Connector Definitions, D-Style

If the 60 pin axis cable is split into (4) 15 pin groups, it is possible to attach 15 pin IDC style connectors for a simple cable assembly. However the D connector pin numbering convention does not correspond to the wire number sequentially across. When using IDC D connectors please refer to the following table:

D Pin Number	Description
1	Encoder A+
2	Encoder B+
3	Encoder I+
4	Amp Enable High
5	Position Capture
6	Motor Command
7	Direction
8	Ground
9	Encoder A-
10	Encoder B-
11	Encoder I-
12	Amp Enable Low
13	Position Compare
14	Step Pulse
15	+5 Volts

I/O Connector Definition

The 50 pin connector provides TTL level inputs and outputs. Outputs sink 12 ma. The pin number is the I/O number with the exception of 49 (+5) and 50 (ground). Input or output sense is configured in 4 bit groups. The groups are defined by "splitting" the connector into (2) 1x50 strips, and then slicing those strips into (12) groups of (4) bits each. This partitioning was chosen so that the even-pin strip could be configured as inputs allowing a standard OPTO-22 cable to plug into the connector without contention between the cable grounds (located on all the even pins) and signals normally available on those pins.

	Description	Pin	Pin	Description	
Group 1	I/O 1	1	2	I/O 2	Group 2
Group 1	I/O 3	3	4	I/O 4	Group 2
Group 1	I/O 5	5	6	I/O 6	Group 2
Group 1	I/O 7	7	8	I/O 8	Group 2
Group 3	I/O 9	9	10	I/O 10	Group 4
Group 3	I/O 11	11	12	I/O 12	Group 4
Group 3	I/O 13	13	14	I/O 14	Group 4
Group 3	I/O 15	15	16	I/O 16	Group 4
Group 5	I/O 17	17	18	I/O 18	Group 6
Group 5	I/O 19	19	20	I/O 20	Group 6
Group 5	I/O 21	21	22	I/O 22	Group 6
Group 5	I/O 23	23	24	I/O 24	Group 6
Group 7	I/O 25	25	26	I/O 26	Group 8
Group 7	I/O 27	27	28	I/O 28	Group 8
Group 7	I/O 29	29	30	I/O 30	Group 8
Group 7	I/O 31	31	32	I/O 32	Group 8
Group 9	I/O 33	33	34	I/O 34	Group 10
Group 9	I/O 35	35	36	I/O 36	Group 10
Group 9	I/O 37	37	38	I/O 38	Group 10
Group 9	I/O 39	39	40	I/O 40	Group 10
Group 11	I/O 41	41	42	I/O 42	Group 12
Group 11	I/O 43	42	44	I/O 44	Group 12
Group 11	I/O 45	45	46	I/O 46	Group 12
Group 11	I/O 47	47	48	I/O 48	Group 12
	+5 Volts	49	50	Ground	

EStop Connector Definition

The EStop connector has 6 pins defined as follows

pin 1	Not Connected (pin 1 is closest to the mounting bracket, rear of PC)
pin 2	Ground
pin 3	E-Stop input
pin 4	I/O 1 from 50 pin connector
pin 5	12 Volt Input for Unlocking Flash Memory
pin 6	12 Volt Source from PC

Placing a jumper between pins 2 and 3 enables the E-Stop (which must be maintained at ground against its 4.7k pullup). This is not recommended if doing anything besides bench testing free spinning motors.

Placing the jumper between pins 3 and 4 redirects the EStop to be from the general I/O connector where an OPTO-22 module rack may be hooked in, or some other IO interconnect that has been chosen for general purpose I/O

A third option is to put a 6 x 1 plug into this header with a cable for pins 2 and 3. A normally closed switch would serve as an E-Stop switch. If the switch disconnected, or the cable was missing, the controller will not enable power to the amplifiers.

The on-board Flash memory chip is used to store application programs. If the board contains a 28F class Flash Memory chip, a 12 volt level must be supplied to the chip to "unlock" the chip and permit alteration of its contents. This level can be provided by turning on switch number 4 on the board itself. In some applications, accessing switch 4 may be inconvenient. In this case, an external switch can be provided that connects pin 5 and pin 6 allowing the memory device to be programmed. Alternately, consult with Douloi Automation regarding a newer 29F class memory chip which does not require the 12 volt level.

External Bus Connector

The remaining 26 pin connector provides a simplified 8-bit bus that can be used to connect to additional hardware. Note that Douloi provides a PC/104 "bridge" accessory that is driven by this connector. The PC/104 format allows the use of many third part cards

Power signals from this connector should only be for signal-level power. If you need any significant current, use a disk-drive connector. Additional details about the use of this bus are available from Douloi Automation on request.

Pin	Description
1	Data 0
2	Data 1
3	Data 2
4	Data 3
5	Data 4
6	Data 5
7	Data 6
8	Data 7
9	Addr 0
10	Addr 1
11	Addr 2
12	Addr 3
13	Addr 4
14	Addr 5
15	Addr 6
16	Select
17	Write/Read
18	Comm_Capture_1
19	Comm_Capture_2
20	Comm_Capture_3
21	Comm_Capture_4
22	Reset
23	+12 Volts
24	-12 Volts
25	+5 Volts
26	Ground

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z