NISTIR 5601

Expert Control System Shell Version 1.0 User's Guide

Stephen A. Osella

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Manufacturing Engineering Laboratory
Factory Automation Systems Division
Gaithersburg, MD 20899

NIST

NISTIR 5601

Expert Control System Shell Version 1.0 User's Guide

Stephen A. Osella

U.S. DEPARTMENT OF COMMERCE
Ronald H. Brown,
Secretary of Commerce

Technology Administration
Mary L. Good,
Under Secretary for Technology

National Institute of Standards and Technology
Arati Prabhakar,
Director

February 1995

Disclaimer

No approval or endorsement of any commercial product by the National Institute of Standards and Technology is intended or implied. Certain commercial equipment, instruments, or materials are identified in this report in order to facilitate understanding. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

This publication and accompanying software was produced by the National Institute of Standards and Technology, an agency of the United States Government, and by statute is not subject to copyright in the United States. Recipients of this software assume all responsibility associated with its operation, modification, maintenance, and subsequent redistribution.

Table of Contents

List of Illustrations

Figure

CHAPTER 1
INTRODUCTION

The Expert Control System Shell (ECSS) is a software tool for rapidly prototyping and deploying control systems of arbitrary complexity. With the ECSS, a control system designer creates a controller which consists of a graphical user interface (called the operator front panel) and, optionally, a collection of expert system rule-based programs. A controller can be operated in "manual mode" where an operator interacts directly with the system to be controlled without using the expert system, in "automatic mode" where the expert system is used to perform most, if not all, system interactions, or in a combination of the two modes.

There are basically four steps involved in developing a functional controller with the ECSS: 1) creating controls (actuators and displays), 2) importing device interface functions, 3) creating rule-based programs, and 4) importing expert system functions. When used in purely manual mode, expert system programs and functions are not used. Most applications, however, require some level of automatic control.

The ECSS allows generic interface to devices through the use of device drivers. The device drivers must be customized for each application and must be written by the control system developer. Device drivers are code resources which are imported into a controller file. Device driver programs have a common code interface and respond to five messages: initialize, terminate, sample, read , and write. Creating and importing device drivers is discussed in detail in chapter 4.

1.1. Computer System Requirements

The ECSS runs only on Macintosh computers having at least a 68020 CPU, a Floating Point Unit 68881, at least 4 Mbytes of RAM, and running System 7.0 or higher. A Power Macintosh version is being developed. Currently, ECSS version 1.0 only permits code resources written in C and created using Symantec's THINK C compiler.

It is helpful if the user is familiar with the Macintosh Operating System essentials such as how to launch applications, opening files, selecting, etc. Refer to your Macintosh User's Guide for additional information.

1.2. Software Installation

The distribution diskette comes with three items at the root-level: 1) the ECSS application, 2) the Tutorial folder, and 3) the Examples folder. To install the software, simply copy these three items to the folder of your choice.

1.3. Conventions and Organization

In the User's Guide, **bolded** words indicate user interface items such as menu items, dialogs, dialog items, etc. Words in *italics* are used for names of objects such as files, control name, etc. Occasionally, statements in "bullet" form (i.e., •) appear in the User's Guide and are intended to make the user aware of particularly important aspects of the ECSS's operation.

The User's Guide will be discussed in the following order: 1) a tutorial on how to create a controller, 2) creating controllers, 3) creating and importing device interface functions, 4) creating rule-based programs, 5) importing expert system functions, and 6) operating controllers.

CHAPTER 2
TUTORIAL

This chapter presents an example, in tutorial form, of how to create a functional controller with the ECSS. In particular, the tutorial is a step-by-step guide to re-creating the '*dc_motor.cfg*' example (found in the *Examples:D.C. Motor* folder of the distribution diskette) which is an interactive simulator of a D.C. motor controller. This example includes 1) a controller front-panel, 2) an expert system program, and 3) a simulator device driver.

2.1. The D.C. Motor Control System

The system to be controlled is an experimental apparatus, shown in Figure 2.1, consisting of a D.C. motor, a fly-wheel (supplemental inertia), a magnetic brake (fixed load), a gear-box, and a tachometer. The input voltage $V_s(t)$ drives the motor shaft connected to the fly wheel and brake-wheel. The motor shaft is also connected to the gear-box transmission where the shaft's rotational speed is reduced by a factor of 30. Finally, the rotational energy from the gear-box is converted by the tachometer to produce the output voltage $V_t(t)$.

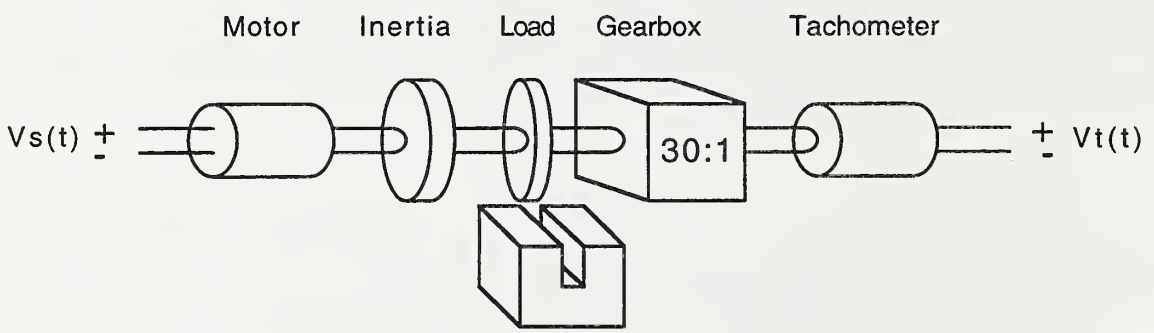

Figure 2.1: The D.C. Motor System.

The mathematical model of the D.C. motor system is given by the transfer function:

$$G(s) = \frac{Y(s)}{U(s)} = \frac{K_m}{(s + \frac{1}{\tau_m})}$$

which means that the system is being modelled as a first-order system. K_m and τ_m are two system parameters; in particular, τ_m is the time-constant of the system. The controller created in this tutorial will permit the changing of these two parameters. The device driver simulator incorporates a gearbox step-down, tachometer gain transfer function, and a voltage reducer into a "composite" motor transfer function G(s).

The overall D.C. motor control system block diagram is shown in Figure 2.2. The structure of the controller $G_D(z)$ was arbitrarily chosen to be of the Proportional-Integral-Derivative (PID) variety. A PID expert system function is built-in to the ECSS. This function takes as arguments the three PID factors: K_p, K_i, and K_d, which will be actuators in the controller file, and also takes as an argument the "error" E(z) which is the difference of the desired and the actual motor R.P.M. converted to volts by K_e. The device driver simulator returns the actual motor "speed" which is converted to R.P.M. by K_{fb}. A Zero-Order-Hold (ZOH) circuit preceeds the motor transfer function.

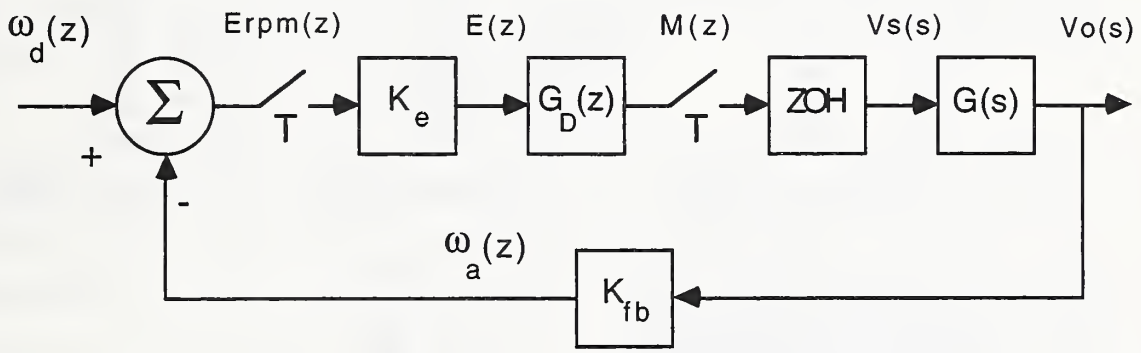

Figure 2.2: The D.C. Motor Control System Block Diagram.

2.2. Creating a Controller File

Launch the ECSS in the normal way that application programs are executed on the Macintosh. This will initiate an open file dialog to open an already existing controller file. Press the **Cancel** button to terminate the open file dialog.

Select the **File** ; **New Controller...** menu item and, via the create file dialog, navigate to the *Tutorial* folder (created during software installation), enter the name '*dc_motor.cfg*', and push the **Save** button. A window will appear which can be resized as desired.

2.3. Adding Controls

After completing the tutorial, the controller window will resemble the one shown in Figure 2.3. The tutorial makes use of four types of controls: button, digital, slider, and a strip chart. The buttons, digitals, and sliders are actuators and the strip chart is a display.

- All control names must be entered exactly as they appear in Figure 2.3.

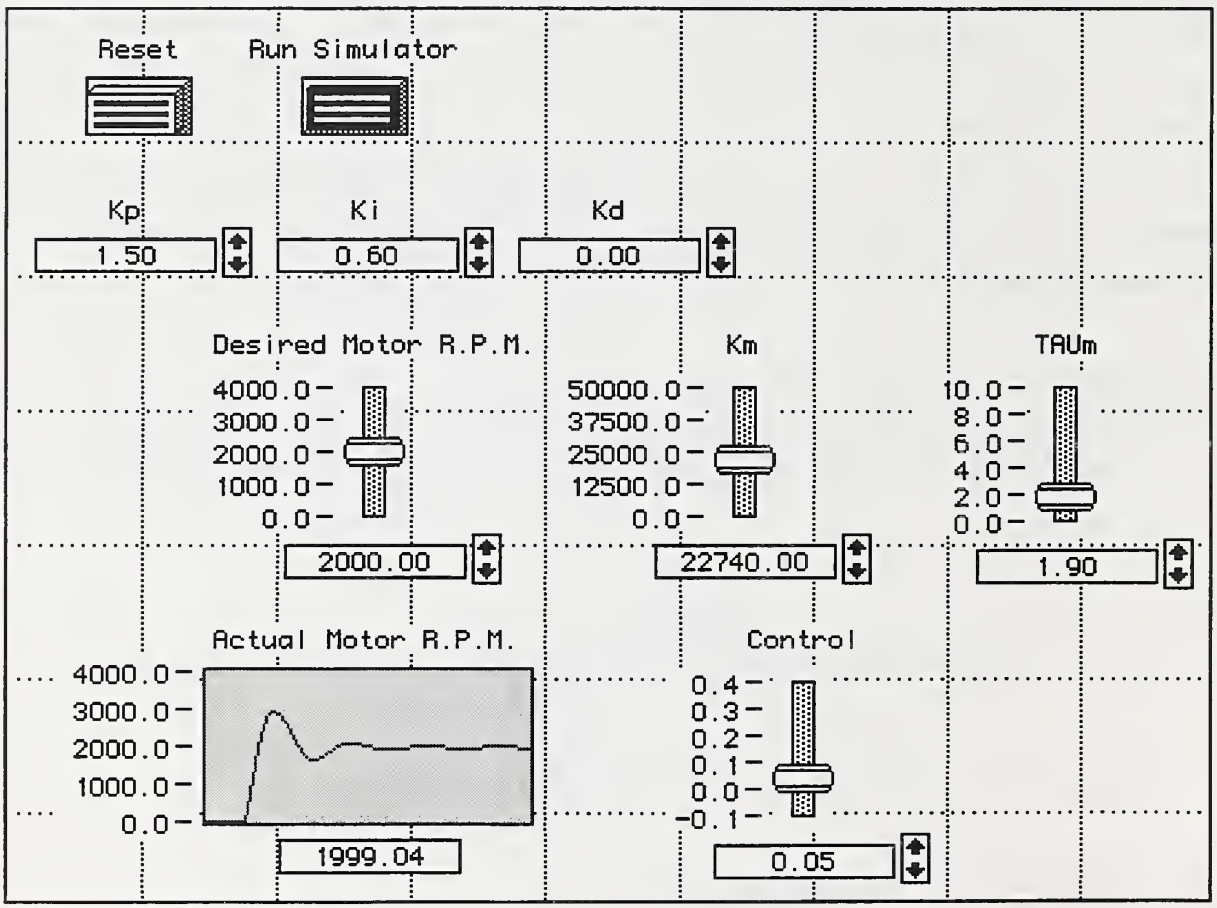

Figure 2.3: D.C. Motor Controller Front Panel.

To create the *Reset* push-button, select the **Controller** ; **Controls** ; **Boolean** ; **Button** menu item. The button icon will appear near the middle of the window. Name the new button control by first changing to the wrench tool by selecting the **Wrench** item in the **Tools** menu. Then, with the wrench cursor, click on the button to pop-up a menu and select the **Show Label** item. Enter the name (*Reset*) in the ensuing dialog and click **OK**. Again, click on the button icon

to pop-up its menu and select the **Output Information...** item. Select the **Internal** radio button in the dialog and click **OK**. Lastly, move the button icon to the appropriate position by first changing to the hand tool by selecting the **Hand** item in the **Tools** menu and then dragging the icon to the position shown in Figure 2.3. Repeat this procedure for the *Run Simulator* button.

Create the *Kp* control by selecting the **Controller** ; **Controls** ; **Numeric** ; **Digital** menu item. Set the control's name as before and make the control an internal control. Because digital controls are, by default, displays, change the control to be an actuator by selecting the **Change To Actuator** item in the control's pop-up menu. A characteristic of digital actuators are the arrows which can be used to change the control's value. Lastly, move the control to the appropriate position. Repeat this process for the *Ki* and *Kd* controls.

Create the *Desired Motor R.P.M.* control by selecting the **Controller** ; **Controls** ; **Numeric** ; **Vertical Slider** menu item. Set the control's name as before and make the control an internal control by selecting the **Output Information...** item. Select the **Show Digital Display** item in the control's pop-up menu to show the digital display and arrows. To modify the axis labels and control dimensions, click the wrench cursor on the axis markers and select the **Axis Information...** menu item. Change the **Maximum** field to 4000.0, the **Divisions** to 4, the **Pixels Per Division** to 12, and the **Precision** to 1. Click **OK** to dismiss the dialog. Lastly, move the control to the appropriate position.

The other three vertical sliders are different in that they are external (device) actuators. That is, they are associated directly with an external device driver which is covered in the next section. As before, create the control named *Control* by selecting the **Controller** ; **Controls** ; **Numeric** ; **Vertical Slider** menu item and then set the control's name. Click on the icon to pop-up its menu and select the **Output Information...** item. By default, the **Device** radio button in the dialog will be selected. Enter the name *DC_MOTOR* in the **Device** text-field. Channel 0 is also the default, so you do not have to change this field. Click **OK** to dismiss the dialog. As before, show the digital display and arrows. Change the axis information so that it resembles those in Figure 2.3. Entering negative numbers can be a little tricky. First enter the number part (e.g., 0.1), and then put the cursor at the beginning and enter the minus sign. Lastly, move the control to the appropriate position. Repeat this process for the *Km* and *TAUm* controls except enter channel 1 for *Km* and channel 2 for *TAUm* in the **Output Information...** dialog. Make sure the control names and channel numbers are correct because they must correspond with those in the rule-based program and device driver.

Create the *Actual Motor R.P.M.* control by selecting the **Controller** ; **Controls** ; **Chart** ; **Strip** menu item and then set the control's name. Modify the axis information and show the digital display. Because strip-charts are displays (and not actuators) the digital display's arrows

will not appear. Change the width of the strip-chart by changing the number of samples displayed by selecting **Options...** in the pop-up menu and setting the **Number Of Samples** to 120 in the dialog. Click on the icon to pop-up its menu and select the **Input Information...** item. By default, the **Device** radio button in the dialog will be selected. Enter the name *DC_MOTOR* in the **Device** text-field. Channel 0 is again the default, so you do not have to change this field. Click **OK** to dismiss the dialog. Lastly, move the control to the appropriate position.

2.4. Adding the Device Driver

Creating and importing device interface functions (drivers) is discussed in detail in Chapter 4. For this tutorial, use the already created device driver called *DC_MOTOR* in the Tutorial folder. To import the device driver, perform the following steps:

1) Select the menu item: **Controller** ; **Devices** ; **Add...**
2) Open the *DC_MOTOR* resource file
3) Select the *DC_MOTOR* code resource in the dialog
4) Double-click on the name, click on the **OK** button, or press the **return** key

2.5. Compiling the Expert System Program

Open the pre-written expert system rule-based program by selecting the **Files** ; **Open...** menu item and selecting the file named *dc_motor.k*. Compile the program by selecting **Controller** ; **Modules** ; **Compile**. A compilation-status window will appear (rather briefly) indicating the progress of the compilation. The compilation window will disappear after the compilation is finished. Because a module (associated with the expert system program) is added to the controller, it is advisable to save the controller file after the compilation is finished. You can close the 'dc_motor.k' window if you so desire.

2.6. Running the Controller

Before running the controller there is one more thing left to do. Set the sampling period by selecting **Operate** ; **Set Sample Period...** and enter *0.1* in the dialog. This is the sampling period in seconds.

To run the controller, select **Operate** ; **Run**. The cursor will automatically change to the finger tool. Click, hold, and drag the *Desired Motor R.P.M.* slider's thumb (i.e., knob) so that the slider digital display reads 2000.0. Then click on the *Run Simulator* push-button to start the

simulation. You will see the *Control* slider move up and down before settling on a constant control value (in volts). You can set the desired R.P.M. at any time or you can change the system parameters by actuating the *Km* and *TAUm* actuator sliders. You can modify the responsiveness of the PID digital controller by changing the PID constants either by typing in the digital display or by using the arrows. The *Reset* push-button is used to reset the PID controller and reset the simulation to a default 'zero' starting point. To stop the controller, select **Operate** ; **Stop**.

CHAPTER 3
CREATING CONTROLLERS

This chapter discusses the essentials of how to create and open controller files, and how to add and modify actuators and displays. Collectively, actuators and displays are called controls.

3.1. Creating and Opening Controller Files

The first two menu items of the **File** menu vary depending on whether or not a controller file is open. If a controller file is not open, the menu items will read **New Controller...** and **Open Controller...**, respectively. The **New Controller...** item is used to create a new controller file while the **Open Controller...** item is used to open an already existing controller file. Only one controller file can be open at any given time.

Upon selecting the **New Controller...** menu item, you will be prompted for a file name. Navigate to the appropriate folder and enter the new controller's file name. Then, click on the **Save** button or hit the **return** key.

To open an already existing controller file, select **Open Controller...** menu item. Navigate to the appropriate folder and select the file. Then, double-click on the selected file, click on the **Open** button, or hit the **return** key. Controller files can also be opened using the **Switch To** item in the **Controller** menu. This option allows controller files to be quickly re-opened. Every time a controller file is opened (or newly created), an item is added to the **Switch To** menu (item). The currently opened controller file is disabled (i.e., ghosted) while all the previously opened files are enabled. A controller file can be re-opened by selecting the appropriate menu item.

3.2. Adding Controls

Once a controller file is opened, controls can be created and/or modified. There are five types of controls: **Numeric**, **Boolean**, **Chart**, **Graph**, and **Timer**. They are found in the **Controller** menu and **Controls** menu item. Each control is either an actuator or a display; actuators are used to effect some type of control and displays are used to visually indicate process variable values. Numeric controls are analog or discrete, while Boolean are binary (on/off). Chart and Timer controls can only be displays. Graphs can either be displays or actuators. There are various control styles for each type of control.

To add a control, select the **Controller** ; **Controls** menu item and then select menu item corresponding to the type and style of the control desired. The new control will initially be placed near the middle of the controller window. Use the hand tool to move the control to the desired location (see section 3.3).

3.3. Moving Controls

To move a control, first select the **Hand** tool. Then click on a control to select it. Lastly, drag the control to the desired location. You can select a number of controls to move simultaneously by holding down the **Shift** key when clicking on a control.

You can also select a number of controls by not clicking on any control. Then a cross and rectangle will appear with which you "trap" the controls. When the mouse button is released, the selected controls are "marqueed". The selected controls can then be moved simultaneously by clicking-and-holding any one of the controls and dragging the outline of the selected controls.

A grid can be used to "attract" the controls to grid boundaries by selecting **Turn Grid On** item in the **Controller** menu. To see the grid, select the **Show Grid** item also in the **Controller** menu.

3.4. Modifying Controls

To modify a control, select the **Wrench** tool (the one that looks like a bone) from the **Tools** menu and then click on the part of the control you wish to modify. A pop-up menu will appear. There are generally five parts to a control: the main body, thumb, label, digital display, and axis markers. The pop-up menu items vary depending on the type and style of control and the part of control selected. The thumb (e.g., the handle part of a slider) is used only for actuation; if selected with the **Wrench** tool, it is as if the main body is being selected.

3.4.1. The Main Body

What constitutes the main body of a control depends on the type and style of control. In general, however, the main body is the "picture" portion of the control. For example, the main body of a strip-chart is the chart rectangle (by default displayed in yellow) and the main body of a toggle is the actual toggle picture.

The **Change To Display/Actuator** menu item is used to modify a control's status from an actuator to a display, and vice versa. Most control types have this capability; however, certain control types, such as charts and timers, do not.

The **Show/Hide Label** menu items are used to toggle the display of the control label (i.e., its name). If the control has not been previously given a name, the first time **Show Label** is selected, the ECSS will prompt you to specify a name.

The **Replace** menu item is in actuality the same as the **Controls** menu item in the **Controller** menu. Upon selecting a replacement type and style of control, the current control is replaced with the new one. With this version of the ECSS, the new control's parameters are initialized. Future versions of the ECSS will retain previous control parameters.

All control types except for Boolean controls have the capability to show a digital display of the current control value. The showing and hiding of the digital display is enabled using the **Show/Hide Digital Display** menu item. Of course, Digital controls (i.e., **Numeric/Digital**) always show their digital displays.

Chart and Graph controls have an **Options...** menu item that can be used to modify parameters intrinsic to charts and graphs. Upon selecting this menu item, a dialog, that depends on the type of control, will appear. For example, the **Foreground Color** and **Background Color** dialog buttons can be clicked on to modify the colors used to draw the chart or graph.

The internal value representation of Numeric controls can be modified by selecting the corresponding data type sub-menu item of the **Data Type** menu. The internal representation is an important aspect of a control because it defines 1) the storage requirements when control runs are logged to disk (see chapter 7) and 2) the device interface requirements (see Chapter 4).

Actuator controls present an **Action** menu item with **Standard** and **Momentary** sub-menu items. The **Action** menu item sets the "action" of the thumb upon its release. If **Standard** is enabled (i.e., is checked in the menu) then the thumb will remain at the position (and thus value) where it was released. On the other hand, the **Momentary** menu item dictates that the thumb should return to the default value upon release of the thumb. In ECSS version 1.0, the default value is pre-defined to be the minimum axis-marker value. In future versions, this default value will be user-defined.

Actuator controls also present an **Output** menu item with **Upon Release** and **Continuous** sub-menu items. If **Upon Release** is enabled, the ECSS will send a write message only when the thumb is released. On the contrary, if **Continuous** is enabled, the ECSS will continuously send write messages, as fast as possible, until the thumb is released.

Two very important pop-up menu items are the **Output Information...** item for actuators and **Input Information...** item for displays. An actuator can either be Device or Internal. An external actuator sends write commands to the associated device named in the **Device** field as well as being accessible by the expert system. Internal actuators can only be accessed by the expert system programs.

A display can either be Device, Internal, or Function. A device display "connects" directly with a device and sends read commands to its device interface function; therefore, you have to specify a device name and channel number (see Chapter 4). If the display is a function then an expression is specified in the **Function** field. The Backus-Naur-Form (BNF) grammar for an expression is the following (bolded characters indicate literals):

<expression>	::=	<number> | <control_name> | <constant> | <variable> | <function> | (<operator> <expression> <expression>)
<number>	::=	<digits> | <digits> **.** <digits>
<digits>	::=	<digit> | <digit> <digits>
<digit>	::=	**0 .. 9**
<control_name>	::=	<identifier> | <string_ identifier>
<identifier>	::=	<letter> | <letter> <identifier_symbols>
<identifier_symbols>	::=	<identifier_symbol> | <identifier_symbol> <identifier_symbols>
<identifier_symbol>	::=	Any keyboard character except white-space characters
<string_ identifier>	::=	**"** < any keyboard character > **"**
<constant>	::=	<identifier>
<variable>	::=	<identifier>
<function>	::=	<identifier> (<expression_list>)
<expression_list>	::=	<expression> | <expression> **,** <expression_list>
<operator>	::=	* | / | + | -

There is no operator precedence apart from parentheses and expressions are evaluated left-to-right. There are a number of built-in constants, variables, and functions which are listed in Appendix B.

As an example, the infix expression:

$$10 + (4.0 * \cos((2\pi * 0.1 * t) + \pi/4))$$

would be entered into the Function field as:

(+ 10 (* 4.0 cos ((+ (* (* 2 (* PI 0.1)) t) (/ PI 4)))))

where PI stands for π and t is a built-in variable (see Appendix B) that simply keeps track of the seconds that have passed since the controller began operation. The use of prefix notation considerably simplifies the expression parser.

3.4.2. The Label

Selecting a control's label with the **Wrench** tool pops-up a menu with three items: **Hide Label**, **Modify Label...**, and **Text Attributes...** . The **Hide Label** menu item causes the label not to be displayed. The **Modify Label...** item brings-up the dialog which was originally used to first enter the control's label. The **Text Attributes...** item is used to modify the display attributes (font, size, and justification) of the label text.

3.4.3. Digital Displays

The pop-up menu for digital displays has three items: **Hide Digital Display**, **Text Attributes...**, and **Precision...** . The **Hide Digital Display** menu item causes the label not to be displayed. The **Text Attributes...** item is the same as for labels. The **Precision...** is used to specify the number of digits after the decimal point to be displayed.

The precision also defines the amount of increment/decrement performed when using the arrow-keys. Therefore, if the precision is equal to 2 (i.e., two digits after the decimal point), the increment/decrement amount will be equal to 0.01 which is 10 to the power -2. The precision depends on the controls internal data type. If the data type is not floating-point, the precision will be automatically set to 0 and cannot be changed.

3.4.4. Axis Markers

The pop-up menu for vertical axis-markers has two items: **Axis Information...** and **Text Attributes...** . The **Axis Information...** menu item brings-up a dialog with five items: **Maximum**, **Minimum**, **Division**, **Pixels Per Division**, and **Precision**. The **Text Attributes...** item is the same as for labels and digital displays.

The **Maximum** and **Minimum** dialog items are used to specify the maximum and minimum values that will be displayed by the control (e.g., in the chart window). These limits are solely used to specify the *display* extremes; the control value can achieve any value permissible by its corresponding data type. The **Division** and **Pixels Per Division** items are used to specify the number of divisions between tick mark and the number of pixels per division, respectively.

The **Precision** item is used to specify the number of digits after the decimal place that will be displayed on the axis markers. As with digital displays, if the data type is not floating-point, the precision will be set to 0 and cannot be changed. The axis marker precision and the digital display precision can be set to different values. The former is used only to define how the axis markers should be displayed. The digital display precision, however, is important because it also specifies the number of digits to print-out when converting log files to text files (see Chapter 7).

3.5. Cutting, Copying, and Pasting Controls

Controls can be removed and duplicated using the **Cut**, **Copy**, and **Paste** menu items of the **Edit** menu. To **Cut** a control means to remove the control from the current controller. A control is **Cut** by first selecting the control (in the same way as if you were going to move it) and then selecting the menu item **Edit** ; **Cut** or by entering **command-X**. The control is removed from the control but is temporarily saved onto the clipboard. The control (or controls) can immediately be pasted (i.e., put) back by selecting the menu item **Edit** ; **Paste** or by entering **command-V**. However, if any other control is cut or copied, the previous cut (or copy) is rendered obsolete.

A control (or controls) can be copied by selecting the control, selecting the menu item **Edit** ; **Copy** or by entering **command-C**, and then pasting the copied controls. The controls are pasted "in place", so that it will appear as if nothing was pasted; however, the controls are simply stacked one on top of the other. Use the **Move** tool to move the copied control to the desired location. A copied control actually is copied first to the clipboard; therefore, you can make multiple copies of a control since the template resides on the clipboard until the next cut or copy.

3.6. Saving and Copying Controller Files

Additions and modification to controller files only become permanent when a controller is saved to disk. To save a controller file, press the **command-S** key or select the **File** ; **Save** menu item. If a controller file has not been modified, the **Save** menu item is disabled. It is only enabled when a change to the controller file has been made.

Controller files can be copied (within the ECSS) in two ways. First, if the **File ; Save As...** menu item is selected, a file request dialog will appear requesting the location (volume and folder) and name of the new controller file. Upon entering the file name (and pressing return), the ECSS will copy the contents of the current controller file to the new controller file and then switch to the new one. In a similar fashion, if the **File ; Save A Copy As...** menu item is selected, the contents of the current controller file will be copied to the new controller file; however, the ECSS will not switch to the new one, but will keep the current one open instead.

CHAPTER 4
CREATING AND IMPORTING DEVICE INTERFACE FUNCTIONS

In writing a Device Interface Function (DIF), you will be basically following the guidelines for writing a code resource (See "Building Code Resources" in the THINK C User's Manual). The device interface functions have a specific format and must set up a code resource global variable environment. Use the "DIF_template.π" THINK C project and "DIF_template.c" source file as a starting point and use the example DIF's as guides.

4.1. Device Interface Function Project Files

Duplicate one of the example DIF project files or create a new THINK C code-resource project. In your project for the DIF, set the project type in the following manner:

Project Type (Radio Button): Code Resource
File Type: DICR
Creator: RSED
Name: The name specified here is the one that you will use when connecting the resource to controls in the ECSS (see Section 4.5)
Type: DICR

Also, when compiling the DIF, make sure that the **68020** and **68881** options are set for your THINK C project and enable the **Purgeable** attribute flag.

4.2. Writing Device Interface Functions

The starting point of a DIF is the main function which has the following syntax:

```
void main (long cmd, long channel_nbr, Ptr data)
```

where *cmd* is a long int whose value depends on the specific command (See Section 4.3), *channel_nbr* is a long int whose value is the id of the channel to read from or write to, and *data* is a generic pointer whose value (i.e., what it points to) depends on the command.

Upon entering the main function, a DIF must set up the code resource's global variable environment. This is done by the two function calls:

```
RememberA0 ();
SetUpA4 ();
```

which are provided by the Macintosh system library. You must include the file "SetUpA4.h" in the DIF source file to access these functions. At this point, the code resource's global variables are accessible and everything is right with the world.

Before exiting the DIF, the old (i.e., the ECSS code) global variable environment must be reset. This is done by calling the function:

```
RestoreA4 ();
```

4.3. Device Interface Function Commands

There are five commands to which a DIF must respond: Initialize, Terminate, Sample, Read, and Write. They are discussed individually in this section.

4.3.1. Initialize

The *Initialize* command (*cmd* == 0) is sent every time the controller is run. This facility should attempt to clear any buffers, check the communication lines, etc. The parameter *data* points to a Boolean variable. Set **data* to TRUE (1) or FALSE (0) depending on the success or failure of the device initialization procedure, respectively.

4.3.2. Terminate

The *Terminate* command (*cmd* == 1) is sent upon stopping a run. This facility should de-allocate any memory, turn any devices left on to off, etc. No return value is expected.

4.3.3. Sample

The *Sample* command (*cmd* == 2) is sent periodically to the DIF. The sampling frequency depends on the sampling period selected (see Chapter 7). The DIF should sample *all* sensors upon receiving this command. This ensures that sampling is synchronized for all sensors and in the ECSS. The reading of the sensors (by the ECSS) is performed by the read command, which is discussed in the next sub-section.

4.3.4. Read

The *Read* command (*cmd* == 3) is sent to read the value of the sensor associated with a controller display. For each display in a controller, the read command is sent to read the particular channel associated with that display. For the read command, the parameter *data* is a pointer to a variable whose type depends on the internal representation of the display's value (see Section 3.4.1). Note that for graphs, the variable is an array. Therefore, in the DIF, the parameter *data* must be dereferenced using the appropriate type casting (see the example DIF's).

- It is up to the control system developer to make sure that the internal representation of a display's value and the data type for the corresponding channel in the DIF are the same.

4.3.5. Write

The *Write* command (*cmd* == 4) is sent when an actuator is acted upon either through the user interface or in a rule-based program. The write command is similar to the read command in that the *data* parameter is a pointer to a variable whose type depends on the internal representation of the actuator's value. Again, in the DIF, the parameter *data* must be dereferenced using the appropriate type casting.

4.4. Building DIF Code Resources

After having set the DIF project type and written the DIF program, the code resource has to be created. In the THINK C application, perform the following steps:

1) Select the menu item: **Project ; Build Code Resource...**
2) Navigate to the desired folder and save the code resource file with a file name of your choice. This is the name of the code resource file.

4.5. Importing DIF Code Resources

After a DIF code resource has been created, it has to be imported within the controller file. Make note of the *code resource* name (not the file name) because it is this name that is used to name the device in the **Input/Output Information** dialogs explained above.

To import a DIF code resource into a controller file from the ECSS, perform the following steps:

1) Select the menu item: **Controller** ; **Devices** ; **Add...**
2) Navigate to and open the resource file containing the DIF code resource
3) Select the DIF code resource in the dialog
4) Double-click on the name, click on the **OK** button, or press the **return** key

It is generally a good idea to save the ECSS controller file after importing a DIF.

- Make sure your DIF is debugged before importing it within the ECSS.

A good way to debug your DIF code is to create a test program which calls the device driver main routine. To do this you will have to rename the device driver *main* function and comment-out the lines of code pertaining to setting-up global variables, i.e., RememberA0(), SetUpA4(), and RestoreA4().

CHAPTER 5
CREATING EXPERT SYSTEM PROGRAMS

Creating rule-based programs is probably the trickiest part of working with the ECSS. If you have had any experience with rule-based programming (Prolog, OPS5, CLIPS, etc.) you will be advantaged. In the ECSS, the expert system is forward-chaining (data-driven). Each program consists of constants, global variables, facts, and rules. The ECSS allows the incorporation of a number of expert system programs that is limited only by the amount of available memory.

Each expert system program is associated, abstractly, with a *control module* of the same name. The programs/control modules operate concurrently and have the capability to send and receive messages to and from one another. This facility is discussed in section 5.4.

5.1. BNF Grammar

The BNF grammar for an ECSS rule-based program is the following (bolded words indicate literals):

<program>	::=	<program-items>
<program-items>	::=	<program-item> | <program-item> <program-items>
<program-item>	::=	<constants> | <variables> | <facts> | <rule>
<constants>	::=	(**CONSTANTS** <constant_patterns>)
<constant_patterns>	::=	<constant_pattern> | <constant_pattern> <constant_patterns>
<constant_pattern>	::=	(<identifier> <item>)
<variables>	::=	(**VARIABLES** <variable_patterns>)
<variable_patterns>	::=	<variable_pattern> | <variable_pattern> <variable_patterns>
<variable_pattern>	::=	(<identifier> <item>)
<facts>	::=	(**FACTS** <patterns>)
<rule>	::=	(**IF** <conditions> **THEN** <actions>)
<conditions>	::=	<condition> | <condition> <conditions>
<condition>	::=	<pattern> | <variable> <pattern>

Note: If the variable is specified, then the *fact* matching the pattern is bound to the variable. This is necessary be able to retract facts from the knowledge-base.

<actions> ::= <action> | <action> <actions>
<action> ::= (<identifier> <pattern>)
<patterns> ::= <pattern> | <pattern> <patterns>
<pattern> ::= (<items>)
<items> ::= <item> | <item> <items>
<item> ::= <literal> | <constant> | <variable> | <function> | <pattern>
<literal> ::= <number> | <identifier> | <string> | <boolean>
<identifier> ::= alpha-numeric characters (starting with a letter) and the underscore.
<string> ::= " <printable_ascii_characters> "
<boolean> ::= **TRUE | FALSE**
<constant> ::= #<identifier>
<variable> ::= ?<identifier> | @<identifier>

Note: ?<identifier> indicates a local variable (to the rule) and @<identifier> indicates a global variable.

<function> ::= =(<identifier> <pattern>) | =<variable>(<identifier> <pattern>)

Note: If the variable is included, the variable is bound to the function's return value.

There are some restrictions to the above BNF grammar. For instance, fact patterns cannot contain function items. Furthermore, no check is made to see if global variables are bound prior to the function call. All restrictions are fairly intuitive and will be caught by the compiler.

There are a number of built-in functions in the ECSS's expert system library and they are listed in Appendix A.

5.2. Compiling Expert System Programs

Expert system programs must be "compiled" before they can be used to operate a controller. Actually, the programs are not compiled in the sense that they are converted to machine code. Instead, they are converted into an internal ECSS data structure that permits relatively more efficient computation; however, they are still interpreted.

A rule-based program is compiled in one of three ways: 1) by explicitly requesting that a program be compiled, 2) by explicitly "bringing-up-to-date" all the programs in the controller, and 3) indirectly compiling all the programs in the controller when the controller is run. To explicitly compile a rule-based program, the program file must first be open and then the menu item **Controller** ; **Modules** ; **Compile** must be selected (**command-K**). This menu item is only enabled when a program file window is the active window and then only if the file has been modified since the last compilation of the program. Upon selecting the **Compile** menu item, a message window is displayed which shows the number of constants, global variables, facts, and rules that have so far been compiled. The message window is automatically closed after the program is compiled. In the (unlikely) event of a syntax error, the ECSS will display a "Compilation Error" dialog which specifies the type of error and then it will position the cursor to the place in the program file where the error occurred. On occasion, the cursor placement is not the exact position of the error, but instead is the next position where an anomalous condition occurred.

All the rule-based programs in the controller can be compiled at the same time by selecting the **Controller** ; **Modules** ; **Bring Up To Date** menu item or by entering **command-U**. Only the programs which need compilation are compiled. Again, the compilation message window will appear but will be reset for each program. In the event of a compilation error, the ECSS will abort further compilation, will position the cursor to the point of the error, and will put up the error message dialog. If the program file is not open, the ECSS will first open the file and then position the cursor.

Lastly, programs are automatically brought up-to-date before the controller is run (see Chapter 7). If there is a compilation error, the controller is *not* run and the usual compilation error events occur. A controller is run if (but not only if) all programs have been compiled successfully.

5.3. Adding and Deleting Control Modules

Each expert system program is associated with a *control module* of (exactly) the same name. A control module can be considered as a "manager" in the hierarchical-control paradigm or an "agent" in the autonomous-agent paradigm. A program/control-module can be added to the controller in one of two ways: 1) explicitly and 2) implicitly when the program is compiled successfully. A control module can be explicitly added to a controller either by selecting **Controller** ; **Modules** ; **Add File** when a program file window is the active window and the file has not already been added to the controller, or by selecting **Controller** ; **Modules** ; **Add...** at anytime a controller file is open. This method initiates a open-file dialog.

Anytime a program file is compiled successfully and the corresponding control module has not already been added to the controller, the ECSS will automatically include that program/control-module to the active controller. Control modules can be removed from a controller by selecting the **Controller** ; **Modules** ; **Delete...** menu item. This item initiates a dialog which prompts you to select the control module you wish to delete. The ECSS will ask you to verify the removal of the selected control module before actually removing it. Select the **Cancel** button in the dialog to terminate the module deletion procedure.

5.4. Message Passing Between Control Modules

Control modules can be considered as the building-blocks for the creation of a distributed artificial-intelligence-based control system. The ECSS allows the inclusion of a virtually unlimited number of control modules that operate concurrently (processed sequentially) and can synchronize their activity and/or cooperate via message passing. As illustrated in Figure 5.1, a control module can *send a message* to another module. The receiving module can *get the message* from its message port, process the message by looking at the *message arguments*, and can *reply to the message* or can *free the message*. The message sending module can then *get the reply*, process it, and optionally free it.

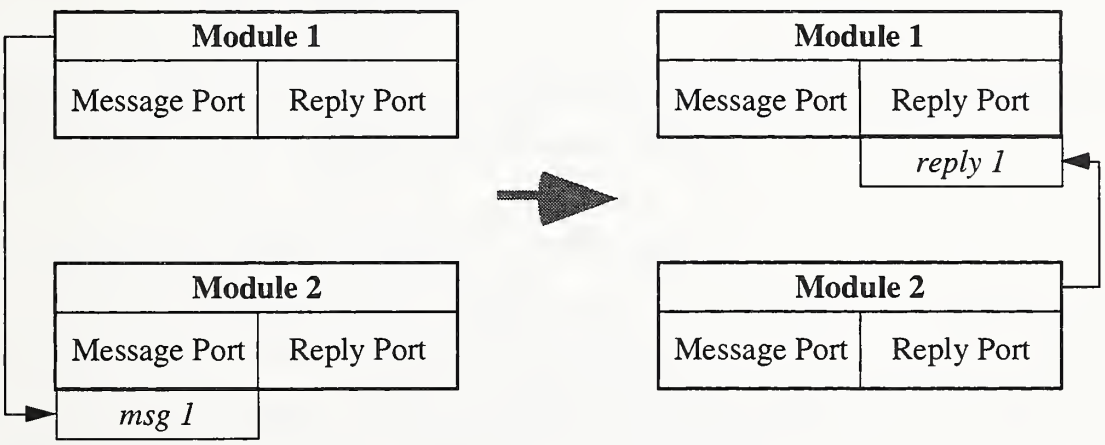

Figure 5.1: Message passing between control modules.

To perform these six functions, the ECSS provides six message related functions: **send_msg**, **get_msg**, **msg_arg**, **send_reply**, **get_reply**, and **free_msg**. These functions are listed, along with their required argument and return data types, in Appendix A. Message arguments can be any ECSS data type (i.e., boolean, integer, real, string, and pointer). The return value type of the **msg_arg** function varies depending on the data type of the message argument.

The processing of a control module can be temporarily suspended pending the reception of a message, or a reply to a message, by using the built-in function **wait**. Upon calling this function, the ECSS will put the module "to sleep" immediately after completely processing the rule which includes this function as an action. When the module is sent a message (or replied to), the ECSS will "wake-up" the module and it will continue as if nothing ever happened.

CHAPTER 6
CREATING AND IMPORTING EXPERT SYSTEM FUNCTIONS

The creation of an Expert System Function (ESF) proceeds in the same manner as the creation of device interface functions except that expert system functions do not have any prescribed format. The function's arguments and return value data types are defined by the call to the function in the expert system program.

6.1. Expert System Function Project Files

In your THINK C project for the ESF set the project type in the following manner:

Project Type (Radio Button): Code Resource
File Type: KBCR
Creator: RSED
Name: This is the name that will be used in expert system programs to call the function.
Type: KBCR

Again, when compiling the ESF, make sure that the **68020** and **68881** options are set for your THINK C project and the **Purgeable** attribute flag is enabled.

6.2. Writing Expert System Functions

The arguments and return values of ESFs depend on the actual call to the function within an expert system program. The ECSS only supports a subset of the C data types: booleans are represented as **unsigned char**s, integers as **signed long int**s, floats as **long double**s, strings as **char ***, and pointers as **char ***. Therefore, when writing ESFs, use these type representations for the function arguments and return value. As with all code resources, the link to the function is main(). As with DIFs, the function calls to RememberA0(), SetUpA4(), and RestoreA4() must be made if global or static variables are used; however, place the **return** statement after the call to RestoreA4(). Look at the example ESF project 'ESF_Template.π' and the corresponding 'ESF_Template.c'. Except for these few restrictions, ESFs can be as general as is needed.

6.3. Building ESF Code Resources

After having set the ESF project type and written the ESF program, the code resource has to be created. The ESF code resource is created in exactly the same way as a DIF code resource. In the THINK C application, perform the following steps:

1) Select the menu item: **Project** ; **Build Code Resource...**
2) Navigate to the desired folder and save the code resource file with a file name of your choice. This is the name of the code resource file, not the name of the code resource you enter in Section 6.1.

6.4. Importing ESF Code Resources

After a ESF code resource has been created, it has to be imported within the controller file. To import a ESF code resource into a controller file from the ECSS, perform the following steps:

1) Select the menu item: **Controller** ; **ES Functions** ; **Add...**
2) Navigate to and open the resource file containing the ESF code resource
3) Select the ESF code resource in the dialog
4) Double-click on the name, click on the **OK** button, or press the **return** key
5) Specify the ESF's return value type by selecting the appropriate radio button. If the function does not return a value (*i.e.*, its a procedure) then select the **Void** radio button. In this case, the ECSS will not assign a value to a "function-value" variable in an expert system program. Make sure the ESF return value and expert system program types correspond.

CHAPTER 7
OPERATING CONTROLLERS

This chapter covers the details on how to operate a controller. In particular, it covers how to 1) start and stop a controller, 2) log data to a (binary) file, set the sampling period used to sample devices, to log data, and by expert system programs, 3) convert binary log files to text files, and 4) play-back a log-file "as if the controller was running" in real-time.

7.1. Running a Controller

This section covers the essentials of starting and stopping controllers, logging data, and setting the sample period.

7.1.1. Starting and Stopping a Controller

The running of a controller is started by selecting the **Operate** ; **Run** menu item or by entering **command-R**. Upon selecting this menu item, the ECSS performs a start-up sequence involving the 1) compiling of any newly added or modified expert system programs, 2) initializing of all devices, 3) resetting of the controls, 4) clearing of any user-interface events in the controls, and 5) changing the active cursor tool to the **Finger** tool. The start-up sequence may fail on steps 1 or 2. In this event, the controller is not run until the problem has been corrected.

While running a controller, the ECSS essentially loops through the following five steps: 1) sampling the devices, 2) updating external displays (*i.e.*, those that read device channels), 3) updating internal displays (*i.e.*, those that use an input function), 4) drawing the new display values, and 5) executing expert system programs. Steps 1-4 are performed only every sample period (see sub-section 7.1.3) while step 5 is performed as often as possible.

The running of a controller is stopped by selecting the **Operate** ; **Stop** menu item or by entering **command-R**. Note that this menu item is in the same location as the **Run** menu item. While running, this menu item reads **Stop**. Upon selecting this menu item, the ECSS requests the operator to confirm the shut-down. If confirmed, the shut-down sequence involves the 1) closing and possibly the saving of the console window if it was opened (using the **printf** ESF), 2) closing of the log file if data logging was occurring (see sub-section 7.1.2), and 3) terminating all devices.

7.1.2. Logging Data

A powerful feature of the ECSS is the capability to save control (both actuators and displays) values to a (binary) data file during the running of a controller. The menu-item for controlling the logging is generically the **Operate ; Log Data** menu-item (**command-L**); however, this menu-item changes with different conditions. If the controller is not running and data-logging is not enabled, the menu-item reads **Log Data**. The actual data-logging occurs only while the controller is running. If the controller is not running and data-logging is enabled, the menu-item reads **Don't Log Data**. If the controller is running and data-logging is not enabled, the menu-item reads **Start Logging Data** and if data-logging is enabled (i.e., the ECSS is currently writing data to a file) the menu-item reads **Stop Logging Data**. Data-logging occurs only on sample intervals specified by the sampling period. The log file is a temporary one and is saved to disk only after stopping the controller, at which time a file-save dialog requests the name and folder of the log file.

Data logging can also be controlled within an expert system program by using the built-in ESF **log_data**. The statement **log_data(TRUE)** begins the logging of data if the ECSS is not currently doing so and **log_data(FALSE)** stops data-logging if it is currently on. Data-logging can still be controlled from the user-interface menu-item; however, the data-logging status can change if manipulated within an expert system program.

7.1.3. Setting the Sampling Period

An important consideration when running a controller is selecting the sampling period. In the ECSS, the sampling period 1) defines when display controls are updated (either by reading a device channel or using an input function), 2) dictates when control values are written to the log file, and 3) is typically used in expert system programs to properly synchronize control times. The sampling period is set by first selecting the **Operate ; Set Sample Period...** menu item and then by entering the sampling period in the dialog. The sampling period is specified in seconds and has a granularity of 1/60th of a second.

The ECSS will do it's best to keep up with very short sampling periods. The ECSS's ability to keep up depends on the type of machine its running on, the complexity of the controller (i.e., the number of controls), the number and complexity of device drivers, and so on. You might have to do some experimentation to find a suitable sampling period for your configuration. A reasonably short sampling period might be 0.5 seconds or 0.25 if you're fortunate.

7.2. Converting a Log File to Text

The purpose of this facility is to permit the saving of data, which is stored in binary format, to be saved as an ASCII text file. The data can be saved in tab, space, or comma-separated format. Optionally, the names of the controls can be included as the first row of the output text file.

First of all, to convert a log file, no controller file can be open; therefore, if a controller file is open, close it. Then, select the **Operate** ; **Convert...** menu item. Doing so will initiate an open file dialog with which you select the (binary) log file created previously. Subsequently, a dialog will appear requesting the type of column delimiter to use and whether or not to include the control names. Upon selecting the **OK** button in this dialog, the ECSS will prompt you for the name and folder of the text file to be created. Upon selecting the **Save** button in the save file dialog, a message will appear indicating the progress of the conversion in terms of how many "samples" have been so far converted and saved. When the conversion is completed, the message window will disappear, and the text log file will be in the specified folder. This text file can now be used by any text editor, spread-sheet, graphing, or mathematic application.

7.3. Playing-Back a Log File

The ability to play-back a log file is a very useful feature provided by the ECSS. Using a remote-control, a log file can be played-back in real-time, fast-forward, or frame-by-frame. The remote can also be used to fast-forward or rewind to any sample in the log file. Again, to play-back a log file, no controller file can be open; therefore, if a controller file is open, close it. To open a (binary) log file, select the **Operate** ; **Playback...** menu item and then select the log file you wish to open. The controller window can be moved and resized and the remote window can be moved.

The play button (>) is used to play-back the log file in real-time. Play-back will continue until either the stop button is pressed or the last sample has been displayed. If the fast-forward button (>>) is pressed while playing a log file, the samples will be displayed twice as fast (if possible). The frame button (**F**) is used to play the log file back frame-by-frame. The rewind button (<<) is used to return to a previously display sample. When one of the playback buttons is pressed after using the rewind button, the controls are cleared before redrawing.

APPENDIX A

Built-In Expert System Functions

Functions:

Function Name	Description	Return Type
sampling_period()	The sampling period in seconds	*real*
sampling_frequency()	The sampling frequency	*real*
sample_number()	The current sample number	*integer*
sample_time()	The current sample time	*real*
assert(*[items]*)	Assert fact to module fact-base *[items]* : *general*	*void*
retract(*fact_ptr*)	Retract the fact from the module fact-base *fact_ptr* : *pointer*	*void*
bb_assert(*[items]*)	Assert fact to black-board fact-base *[items]* : *general*	*void*
bb_retract(*fact_ptr*)	Retract the fact from the black-board fact-base *fact_ptr* : *pointer*	*void*
send_msg(*name [args]*)	Send a message to module *name* *name* : *string* *[args]* : Optional list of *general* arguments	*void*
get_msg()	Get next message in message port	*pointer*
msg_arg(*msg nbr*)	Get argument *nbr* from message *msg* *msg* : *pointer* *nbr* : *integer*	*general*
send_reply(*msg [args]*)	Send a reply to message *msg* *msg* : *pointer* *[args]* : Optional list of *general* arguments	*void*
get_reply()	Get next reply in reply port	*void*
free_msg(*msg*)	Free (deallocate memory) message *msg* *msg* : *pointer*	*void*
wait()	Put module to sleep pending message	*void*
terminate()	Remove module from processing cue	*void*
halt()	Stop running controller	*void*

Functions:

Function Name	Description	Return Type
rand()	Return a floating-point random number	*real*
reset_rand(*seed*)	Reset random number generator seed *seed* : *integer*	*void*
set_rand_type(*type*)	Set random number generator type *type* : *string* ("uniform" or "normal")	*void*
get_value(*name*)	Get the control value *name* : *string*	*general*
set_value(*name value*)	Get the control value *name* : *string* *value* : *general*	*void*
activate(*name YorN*)	Make control accessible to user-interface *name* : *string* *YorN* : *boolean*	*void*
event_occurred(*name*)	Did user access this control? *name* : *string*	*boolean*
clear_event(*name*)	Clear oldest user-interface "event" *name* : *string*	*void*
clear_events(*name*)	Clear all user-interface "events" *name* : *string*	*void*
acknowledge(*prompt*)	Prompt user to acknowledge a message *prompt* : *string*	*void*
confirm(*prompt*)	Prompt user to confirm a message *prompt* : *string*	*boolean*
dev_write(*dev chnl val*)	Write *val* to device *dev* at channel *chnl* *dev* : *string* *chnl* : *integer* *val* : *general*	*void*
dev_read(*dev chnl*)	Read from device *dev* at channel *chnl* *dev* : *string* *chnl* : *integer*	*general*

Functions:

Function Name	Description	Return Type
log_data(*YorN*)	Start or stop logging data *YorN* : *boolean*	*void*
save_log_file(*fName*)	Save log file to file name *fName* *fName* : *string*	*void*
delete_log_data()	Delete current log file and stop logging data	*void*
sys_beep()	System Beep	*void*
ctime()	Time in string format	*string*
clock()	Clock ticks in integer format	*integer*
fclock()	Clock in seconds	*real*
fopen(*name mode*)	Open file *name* with mode *mode* *name* : *string* *mode* : *string*	*pointer*
fclose(*fp*)	Close file whose file pointer is *fp* *fp* : *pointer*	*void*
fprintf(*fp [args]*)	Print arguments to file *fp* *fp* : *pointer* *[args]* : Optional list of *general* arguments	*integer*
fscanf(*fp [args]*)	Read arguments from file *fp* *fp* : *pointer* *[args]* : Optional list of *general* arguments	*integer*
printf(*[args]*)	Print arguments to console window *[args]* : Optional list of *general* arguments	*void*
strcat(*s1 s2*)	Concatenate *s2* to *s1* and return result *s1* : *string* *s2* : *string*	*string*
strcpy(*s1 s2*)	Copy *s2* to *s1* and return *s1* *s1* : *string* *s2* : *string*	*string*

Functions:

Function Name	Description	Return Type
abs(*x*)	The absolute value of x	*real*
acos(*x*)	The arc-cosine of x	*real*
asin(*x*)	The arc-sine of x	*real*
atan(*x*)	The arc-tangent of x	*real*
atan2(*x y*)	The arc-tangent of x/y	*real*
ceil(*x*)	Rounds up to the next highest integer	*real*
cos(*x*)	The cosine of x	*real*
cosh(*x*)	The hyperbolic-cosine of x	*real*
exp(*x*)	The exponential of x	*real*
floor(*x*)	Rounds down to the next lowest integer	*real*
fmod(*x y*)	Floating-point remainder of x / y	*real*
log(*x*)	The natural logarithm of x	*real*
log10(*x*)	The base-10 logarithm of x	*real*
pow(*x y*)	x to the y	*real*
sin(*x*)	The sine of x	*real*
sinh(*x*)	The hyperbolic-sine of x	*real*
sqr(*x*)	The square of x	*real*
sqrt(*x*)	The square-root of x	*real*
tan(*x*)	The tangent of x	*real*
tanh(*x*)	The hyperbolic-tangent of x	*real*
+(*x y*)	x + y	*number*
-(*x y*)	x - y	*number*
*(*x y*)	x * y	*number*
/(*x y*)	x / y	*number*
itoa(*i*)	Convert integer *i* to string format	*string*
atoi(*s*)	Convert string *s* to integer	*integer*
ftoa(*x*)	Convert real *x* to string format	*string*
atof(*s*)	Convert string *s* to real	*real*

Functions:

Function Name	Description	Return Type
gt(*x y*)	Is *x* greater-than *y*? *x* : *real or integer* *y* : *real or integer*	*boolean*
gte(*x y*)	Is *x* greater-than or equal to *y*?	*boolean*
lt(*x y*)	Is *x* less-than *y*?	*boolean*
lte(*x y*)	Is *x* less-than or equal to *y*?	*boolean*
eq(*x y*)	Is *x* equal to *y*?	*boolean*
neq(*x y*)	Is *x* not equal to *y*?	*boolean*
and(*a b*)	Logical *a* and *b* *a* : *boolean* *b* : *boolean*	*boolean*
or(*a b*)	Logical *a* or *b*	*boolean*
not(*a*)	Logical negation of *a*	*boolean*

APPENDIX B

Built-In Display Constants, Variables, and Functions

Constants:

PI	The constant π

Variables:

T	The sampling period
Fs	The sampling frequency (*i.e.*, Fs = 1/T)
n	The current sample number
t	The current sample time (*i.e.*, t = nT)

Functions:

abs(x)	The absolute value of x
acos(x)	The arc-cosine of x
asin(x)	The arc-sine of x
atan(x)	The arc-tangent of x
atan2(x y)	The arc-tangent of x/y
ceil(x)	Rounds up to the next highest integer
cos(x)	The cosine of x
cosh(x)	The hyperbolic-cosine of x
exp(x)	The exponential of x
floor(x)	Rounds down to the next lowest integer
fmod(x y)	Floating-point remainder of x / y
ln(x)	The natural logarithm of x
log10(x)	The base-10 logarithm of x
pow(x y)	x to the y
sin(x)	The sine of x
sinh(x)	The hyperbolic-sine of x
sqr(x)	The square of x
sqrt(x)	The square-root of x
tan(x)	The tangent of x
tanh(x)	The hyperbolic-tangent of x
rand()	Uniformly distributed pseudo-random numbers
randn()	Normally distributed pseudo-random numbers