OMRON

eV+ Language

User's Guide

Copyright Notice

The information contained herein is the property of Omron Adept Technologies, Inc., and shall not be reproduced in whole or in part without prior written approval of Omron Adept Technologies, Inc. The information herein is subject to change without notice and should not be construed as a commitment by Omron Adept Technologies, Inc. The documentation is periodically reviewed and revised.

Omron Adept Technologies, Inc., assumes no responsibility for any errors or omissions in the documentation. Critical evaluation of the documentation by the user is welcomed. Your comments assist us in preparation of future documentation. Please submit your comments to: techpubs@adept.com.

Table Of Contents

eV+ Language User's Guide Introduction

The following topics are described in this chapter:

Introduction to the eV+ Language User's Guide

eV+ is a computer-based control system and programming language designed specifically for use with Omron Adept Technologies, Inc. industrial robots, vision systems, and motion-control systems.

As a real-time system, continuous trajectory computation by eV+ permits complex motions to be executed quickly, with efficient use of system memory and reduction in overall system complexity. The eV+ system continuously generates robot-control commands and can concurrently interact with an operator, permitting on-line program generation and modification.

eV+ provides all the functionality of modern high-level programming languages, including:

- Callable subroutines
- Control structures
- Multitasking environment
- Recursive, reentrant program execution

Compatibility

This manual is for use with eV+ v2.x and later. This manual covers the basic eV+ system. If your system is equipped with optional vision, see the *ACE Sight User's Guide* and the *ACE Sight Reference Guide*, for details on your vision system.

Manual Overview

This manual details the concepts and strategies of programming in eV+. Material covered includes:

- Functional overview of eV+
- A description of the data types used in eV+
- A description of the system parameters and switches
- Basic programming of eV+ systems
- Editing and debugging eV+ programs
- Communication with peripheral devices
- Communication with the manual control pendant ("the pendant")
- Conveyor tracking feature
- Example programs

- Using tool transformations

- Accessing external encoders

Many eV+ keywords are shown in abbreviated form in this user guide. See the *eV+ Language Reference Guide* for complete details on all eV+ keywords.

eV+ Release Notes

For information on new features or enhanced keywords listed by eV+ software release, select a link below:

eV+ v2.x Release Notes

Related Publications

In addition to this manual, have the following publications handy as you set up and program your Omron Adept automation system.

Related Publications

Manual	Material Covered
eV+ Language Reference Guide	This manual provides a complete description of the keywords used in the basic eV+ system.
eV+ Operating System User's Guide	This manual provides a description of the eV+ operating system. Loading, storing, and executing programs are covered in this manual.
eV+ Operating System Reference Guide	This manual provides descriptions of the eV+ operating system commands (known as monitor commands).
ACE User's Guide	This manual describes the ACE graphical user interface, which is used to program your Adept motion system.
ACE Reference Guide	This manual provides descriptions of the commands available with systems that include the optional ACE Sight vision system.
Adept SmartController EX User's Guide	This manual detailsthe installation, configuration, and maintenance of your controller. The controller must be set up and configured before control programs will execute properly.

Manual	Material Covered
Adept SmartMotion Developer's Guide *Adept SmartMotion Installation Guide*	These manuals describe the installation, configuration, and tuning of an Adept motion system.
Adept T20 Pendant User's Guide	This manual describes the basic use of the T20 manual control pendant.

Dangers, Warnings, Cautions, and Notes

There are six levels of special alert notation that may be used in this manual. In descending order of importance, they are:

DANGER: This indicates an imminently hazardous electrical situation which, if not avoided, will result in death or serious injury.

DANGER: This indicates an imminently hazardous situation which, if not avoided, will result in death or serious injury.

WARNING: This indicates a potentially hazardous electrical situation which, if not avoided, could result in serious injury or major damage to the equipment.

WARNING: This indicates a potentially hazardous situation which, if not avoided, could result in serious injury or major damage to the equipment.

⚠️ **CAUTION:** This indicates a situation which, if not avoided, could result in minor injury or damage to the equipment.

NOTE: This provides supplementary information, emphasizes a point or procedure, or gives a tip for easier operation.

Safety

The following sections discuss the safety measures you must take while operating an Omron Adept robot.

Reading and Training for System Users

Omron Adept robot systems include computer-controlled mechanisms that are capable of moving at high speeds and exerting considerable force. Like all robot systems and industrial equipment, they must be treated with respect by the system user.
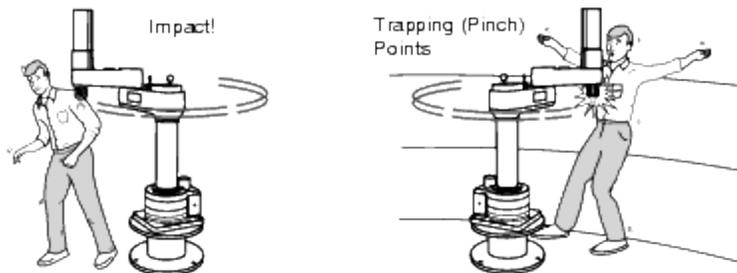
Impacts and Trapping Points

Omron Adept recommends that you read the *American National Standard for Industrial Robot Systems-Safety Requirements,* published by the Robotic Industries Association in conjunction with the American National Standards Institute. The publication, ANSI/RIA R15.06-1986, contains guidelines for robot system installation, safeguarding, maintenance, testing, startup, and operator training. The document is available from the American National Standards Institute, 1430 Broadway, New York, NY 10018.

System Safeguards

Safeguards should be an integral part of robot workcell design, installation, operator training, and operating procedures. Omron Adept robot systems have various communication features to aid you in constructing system safeguards. These include remote emergency stop circuitry and digital input and output lines.

Computer-Controlled Robots

Omron Adept robots are computer controlled, and the program that is running the robot may cause it to move at times or along paths you may not anticipate. Your system should be equipped with indicator lights that tell operators when the system is active. The Front Panel (FP) provides these lights. When the White HIGH POWER enable light on the FP or T20 Pendant is illuminated, do not enter the workcell because the robot may move unexpectedly.
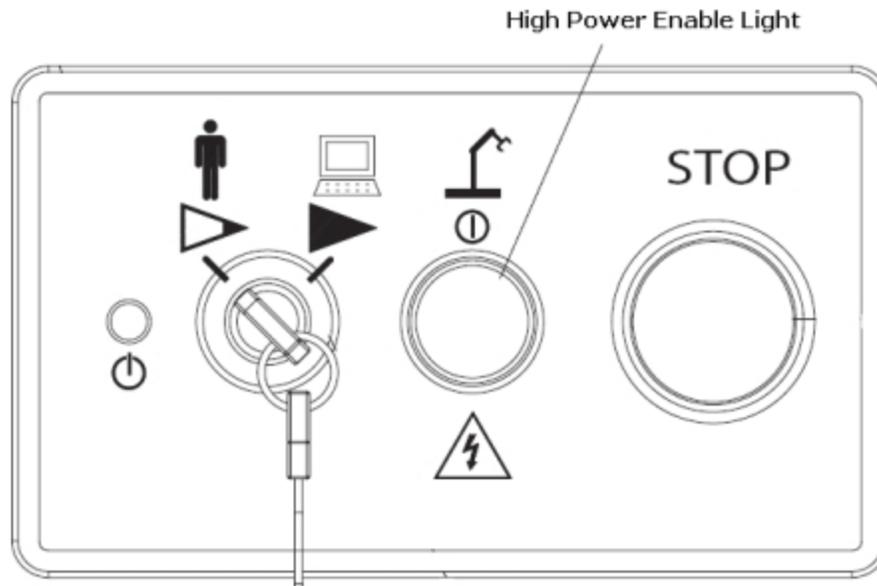
High Power Enable Light

Manually Controlled Robots

Omron Adept robots can also be controlled manually when the white HIGH POWER enable light on the front of the controller is illuminated. When this light is lit, robot motion can be initiated from the terminal or the pendant (see the *T20 Pendant User's Guide* for more information). Before you enter the workspace, turn the keyswitch to manual mode and take the key (or the T20 pendant) with you. This will prevent anyone else from initiating unexpected robot motions from the terminal keyboard.

Other Computer-Controlled Devices

In addition, these systems can be programmed to control equipment or devices other than the robot. As with the robot, the program controlling these devices may cause them to operate at times not anticipated by personnel. Make sure that safeguards are in place to prevent personnel from entering the workcell.

> **WARNING:** Entering the robot workcell when the white HIGH POWER enable light is illuminated can result in severe injury.

Omron Adept Technologies, Inc. recommends the use of additional safety features such as light curtains, safety gates, or safety floor mats to prevent entry to the workcell while HIGH POWER is enabled. These devices may be connected using the robot's remote emergency stop circuitry.

Notations and Conventions

This section describes various notations used throughout this manual and conventions observed by the eV+ system.

Uppercase and Lowercase Letters

You will notice that a mixture of uppercase (capital) and lowercase letters is used throughout this manual when eV+ operations are presented. eV+ keywords are shown in uppercase letters. Parameters to keywords are shown in lowercase. Many eV+ keywords have optional parameters and/or elements. Required keyword elements and parameters are shown in boldface type. Optional keyword elements and parameters are shown in normal type. If there is a comma following an optional parameter, the comma must be retained if the parameter is omitted, unless nothing follows.

Note that the commas preceding the number 300 must be present to correctly relate the number with a Z-direction change.

Numeric Arguments

All numbers in this manual are decimal unless otherwise noted. Binary numbers are shown as ^B, octal numbers as ^, and hexadecimal numbers as ^H.

Several types of numeric arguments can appear in commands and instructions. For each type of argument, the value can generally be specified by a numeric constant, a variable name, or a mathematical expression.

There are some restrictions on the numeric values that are accepted by eV+. The following rules determine how a value will be interpreted in the various situations described.

1. **Distances** are used to define locations to which the robot is to move. The unit of measure for distances is the millimeter, although units are never explicitly entered for any value. Values entered for distances can be positive or negative.[1]

2. **Angles** in degrees are entered to define and modify orientations the robot is to assume at named locations, and to describe angular positions of robot joints. Angle values can be positive or negative, with their magnitudes limited by 180 degrees or 360 degrees depending on the usage.

3. **Joint numbers** are integers from one up to the number of joints in the robot, including the hand if a servo-controlled hand is operational. For Omron Adept SCARA robots, joint numbering starts with the rotation about the base, referred to as joint 1. For mechanisms controlled by AdeptMotion, see the device module documentation for joint numbering.

4. **Signal numbers** are used to identify digital (on/off) signals. They are always considered as integer values with magnitudes in the ranges 1 to 8, 33 to 512, 1001 to 1012, 1032 to 1512, 2001 to 2512, or 3001 to 3004. A negative signal number indicates an off state.

5. **Integer** arguments can be satisfied with real values (that is, values with integer and fractional parts). When an integer is required, the value is rounded and the resulting integer is used.

6. **Arguments** indicated as being **scalar variables** can be satisfied with a real value (that is, one with integer and fractional parts) except where noted. Scalars can range from $-9.22*10^{18}$ to $9.22*10^{18}$ in value (displayed as -9.22E18 and 9.22E18).[2]

---

[1]See the IPS instruction for a special case of specifying robot speed in inches per second.

[2]Numbers specifically declared to be double-precision values can range from $-1.8*10^{-307}$ to $1.8*10^{-307}$.

Programming eV+

The following topics are described in this chapter:

Creating a Program

Beginning with eV+ version v2.x, eV+ programs are created (and debugged) through the ACE user interface. The eV+ Editor and Debugger tools provide a full-featured environment for creating, editing and debugging eV+ programs. For more details, see the chapter Programming ACE in the *ACE User's Guide*.

Program and Variable Name Requirements

Program and variable names can have up to 15 characters. Names must begin with a letter and can be followed by any sequence of letters, numbers, periods, and underline characters. Letters used in program names can be entered in either lowercase or uppercase. eV+ always displays program and variable names in lowercase.

eV+ Program Types

There are two types of eV+ programs:

- Executable Programs
- Command Programs

Executable programs are described in this section. Command programs are similar to MS-DOS batch programs or UNIX scripts, and they are described in the *eV+ Operating System User's Guide*.

Executable Programs

There are two classes of executable programs: robot control programs and general programs.

Robot Control Programs

A robot control program is an eV+ program that directly controls a robot or motion device. It can contain any of the eV+ program instructions.

Robot control programs are usually executed by program task #0, but they can be executed by any of the program tasks available in the eV+ system. Task #0 automatically attaches the robot when program execution begins. If a robot control program is executed by a task other than #0, however, the program must explicitly attach the robot (program tasks are described in detail later in this chapter).

For normal execution of a robot control program, the system switch DRY.RUN must be disabled and the robot must be attached by the robot control program. Then, any robot-related error will stop execution of the program (unless an error-recovery program has been established [see REACTE in the *eV+ Language Reference Guide*]).[1]

Exclusive Control of a Robot

- Whenever a robot is attached by an active task, no other task can attach that robot or execute instructions that affect it, except for the REACTI and BRAKE instructions. For details, see Program Interrupt Instructions on page 62.

- When the robot control task stops execution for any reason, the robot is detached until the task resumes, at which time the task automatically attempts to reattach the robot. If another task has attached the robot in the meantime, the first task cannot be resumed.

- Task #0 always attempts to attach robot #1 when program execution begins. No other tasks can successfully attach any robot unless an explicit ATTACH instruction is executed.

- Since task #0 attempts to attach robot #1, that task cannot be executed **after** another task has attached that robot. If you want another task to control the robot **and** you want to execute task #0, you must follow this sequence of events:

- Start task #0.

- Have task #0 DETACH the robot.

- Start the task that will control the robot. (The program executing as task #0 can start up another task.)

- Have that task ATTACH the robot.

For more information on the ATTACH and DETACH instructions, see Creating Windows on page 162.

- Note that robots are attached even in DRY.RUN mode. In this case, motion commands issued by the task are ignored, and no other task can access the robot.

General Programs

A general program is any program that does not control a robot. With a robot system, there can be one or more programs executing concurrently with the robot control program. For example, an additional program might monitor and control external processes via the external digital signal lines and analog signal lines.

General programs can also communicate with the robot control program (and each other) through global variables and software signals. (General programs can also have a direct effect on the robot motion with the BRAKE instruction, although that practice is not recommended.)

With the exception of the BRAKE instruction, a general program cannot execute any instruction that affects the robot motion. Also, the TOOL settings cannot be changed by general programs.

Except for the robot, general-purpose control programs can access all the other features of the system, including ACE Sight (if it is present in the system), the (internal and external) digital signal lines, the USER serial lines, the system terminal, the disk drives, and the manual control pendant.

Note that except for the exclusion of certain instructions, general-purpose control programs are just like robot control programs. Thus, the term program is used in the remainder of this chapter when the material applies to **either** type of control program.

---

[1]If the system is in DRY.RUN mode while a robot control program is executing, robot motion instructions are ignored. Also, if the robot is detached from the program, robot-related errors do not affect program execution.

Format of Programs

This section presents the format that eV+ programs must follow. The format of the individual lines is described, followed by the overall organization of programs. This information applies to all programs regardless of their type or intended use.

Program Lines

Each line or **step** of a program is interpreted by the eV+ system as a program instruction. The general format of a eV+ program step is:

```
step_number step_label operation        ;Comment
```

Each item is optional and is described in detail below.

Step Number	Each step within a program is automatically assigned a step number. Steps are numbered consecutively, and the numbers are automatically adjusted whenever steps are inserted or deleted. Although you will never enter step numbers into programs, you will see them displayed by the eV+ system in several situations. Step numbers are also often referenced as line numbers.
Step Label	Because step numbers change as a program evolves, they are not useful for identifying steps for program-controlled branching. Therefore, program steps can contain a step label. A step label is a programmer-specified integer (0 to 65535) that is placed at the start of a program line to be referenced elsewhere in the program (used with GOTO statements).
Operation	The operation portion of each step must be a valid eV+ language keyword and may contain parameters and additional keywords. The *eV+ Language Reference Guide* gives detailed descriptions of all the keywords recognized by eV+. Other instructions may be recognized if your system includes optional features.
Comment	The semicolon character is used to indicate that the remainder of a program line is comment information to be ignored by eV+.
	When all the elements of a program step are omitted, a **blank line** results. Blank program lines are acceptable in eV+ programs. Blank lines are often useful to space out program steps to make them easier to read.
	When only the comment element of a program step is present, the step is called a **comment line**. Comments are useful to describe what the program does and how it interacts with other programs. Use comments to describe and explain the intent of the sections of the programs. Such internal documentation will make it easier to modify and debug programs.

The example programs in this manual, and the utility programs provided by Omron Adept with your system, provide examples of programming format and style. Notice that Omron Adept programs contain numerous comments and blank lines.

When program lines are entered, extra spaces can be entered between any elements in the line. The eV+ editors add or delete spaces in program lines to make them conform with the standard spacing. The editors also automatically format the lines to uppercase for all keywords and lowercase for all user-defined names.

When you complete a program line (by entering a carriage return, moving off a line, or exiting the editor), the editor checks the syntax of the line. If the line cannot be executed, it is displayed in red in the ACE editor.

Certain control structure errors are displayed in the status bar of the editor and the program is marked as not executable. (Error checking stops at that point in the program. Thus, only one control structure error at a time can be detected.)

Program Organization

The first step of every eV+ program must be a .PROGRAM instruction. This instruction names the program, defines any arguments it receives or returns, and has the format:

```
.PROGRAM program_name(parameter_list)    ;Comment
```

The program name is required, but the parameter list and comment are optional.

After the .PROGRAM line, there are only two restrictions on the order of other instructions in a program.

- AUTO, LOCAL, or GLOBAL instructions must precede any executable program instructions. Only comment lines, blank lines, and other AUTO, LOCAL, or GLOBAL instructions are permitted between the .PROGRAM step and an AUTO, LOCAL, or GLOBAL instruction.

- The end of a program is marked by a line beginning with .END. The eV+ editors automatically add this line at the end of a program.[1]

Program Variables

eV+ uses three classes of variables: GLOBAL, LOCAL, and AUTO. These are described in detail in Variable Classes on page 47.

---

[1]The .PROGRAM and .END lines are automatically entered by the Omron Adept-supplied eV+ program editors. If you use another text editor for transfer to a eV+ system, you MUST enter these two lines. In general, any editor that produces unformatted ASCII files can be used for programming.

Executing Programs

When eV+ is actively following the instructions in a program, it is said to be executing that program.

The standard eV+ system provides for simultaneous execution of up to seven different programs-for example, a robot control program and up to six additional programs. The optional eV+ extensions software provides for simultaneous execution of up to 28 programs. Execution of each program is administered as a separate program task by the system.

The way program execution is started depends upon the program task to be used and the type of program to be executed. The following sections describe program execution in detail.

Selecting a Program Task

Task # 0 has the highest priority in the (standard) task configuration. Thus, this task is normally used for the primary application program. For example, with a robot system, task #0 is normally used to execute the robot control program.

> **NOTE:** As a convenience, when execution of task #0 begins, the task always automatically selects robot #1 and attaches the robot as soon as a motion related keyword is encountered.

Execution of task #0 is normally started by using the EXECUTE monitor command.

The ABORT monitor command or program instruction stops task #0 after the current robot motion completes. The CYCLE.END monitor command or program instruction can be used to stop the program at the end of its current execution cycle.

If program execution stops because of an error, a PAUSE instruction, an ABORT command or instruction, or the monitor commands PROCEED or RETRY can be used to resume execution (see the *eV+ Operating System Reference Guide* for information on monitor commands). While execution is stopped, the DO monitor command can be used to execute a single program instruction (entered from the keyboard) as though it were the next instruction in the program that is stopped.

For debugging purposes, the ACE eV+ Debugger tool can be used to execute a program one step at a time, and to follow the flow of program execution. For details, see the *ACE User's Guide*.

Execution of program tasks other than #0 is generally the same as for task #0. The following points highlight the differences:

- The task number must be explicitly included in all the monitor commands and program instructions that affect program execution, including EXECUTE, ABORT, PROCEED and RETRY.
- If the program is going to control the robot, it must explicitly ATTACH the robot before executing any instructions that control the robot.

See the section Scheduling of Program Execution Tasks for details on task scheduling.

Program Stacks

When subroutine calls are made, eV+ uses aeV+n internal storage area called a stack to save information required by the executing program. This information includes:

- The name and step number of the calling program.

- Data necessary to access subroutine arguments.

- The values of any automatic variables specified in the called program.

The eV+ system allows you to explicitly allocate storage to the stack for each program task. Thus, the amount of stack space can be tuned for a particular application to optimize the use of system memory. Stacks can be made arbitrarily large, limited only by the amount of memory available on your system.

Stack Requirements

When a eV+ program is executed in a given task, each program stack is allocated 32 kilobytes of memory. This value can be adjusted, once the desired stack requirements are determined, by using the STACK monitor command (for example, in a start-up monitor command program). See the *eV+ Operating System Reference Guide* for information on monitor commands.

One method of determining the stack requirements of a program task is simply to execute its program. If the program runs out of stack space, it stops with the error message:

```
*Too many subroutine calls*
```

or

```
*Not enough stack space*
```

If this happens, use the STACK monitor command to increase the stack size and then issue the RETRY monitor command to continue program execution. In this case, you do not need to restart the program from the beginning. (The STATUS command will tell you how much stack space a failed task requested.)

Alternatively, you can start by setting a large stack size before running your program. After the program has been run, and all the execution paths have been followed, use the STATUS monitor command to look at the stack statistics for the program task. The stack MAX value shows how much stack space your program task needs in order to execute. The stack size can then be set to the maximum shown, with a little extra for safety.

If it is impossible to invoke all the possible execution paths, the theoretical stack limits can be calculated from the figures provided in the following table. You can calculate the worst-case stack size by adding up the overhead for all the program calls that can be active at one time. Divide the total by 1024 to get the size in kilobytes. Use this number in the STACK monitor command to set the size.

Stack Space Required by a Subroutine

Bytes	Required For	Notes
20	The actual subroutine call	
32	Each subroutine argument (plus one of the following):	
4	Each real subroutine argument or automatic variable	1
8	Each double-precision real subroutine argument or automatic variable	1
48	Each transformation subroutine argument or automatic variable	1, 2
varies	Each precision-point subroutine argument or automatic variable	1, 2, 3
84	Each belt variable argument or automatic variable	1, 2
132	Each string variable argument or automatic variable	1, 2

NOTES:

1. If any subroutine argument or automatic variable is an array, the size shown must be multiplied by the size of the array. (Remember that array indexes start at zero.)
2. If a subroutine argument is always called by reference, this value can be omitted for that argument.
3. Requires four bytes for each joint of the robot (on multiple robot systems, use the robot with the most joints).

Flow of Program Execution

Program instructions are normally executed sequentially from the beginning of a program to its end. This sequential flow may be changed when a GOTO or IF...GOTO instruction, or a control structure, is encountered. The CALLinstruction causes another program to be executed, but it does not change the sequential flow through the calling program because execution of the calling program resumes where it left off when a RETURN instruction is executed by the CALLed program.

The WAIT instruction suspends execution of the current program until a condition is satisfied. The WAIT.EVENT instruction suspends execution of the current program until a specified event occurs or until a specified time elapses.

The PAUSE and HALT instructions both terminate execution of the current program. After a PAUSE, program execution can be resumed with a PROCEED monitor command (see the *eV+ Operating System Reference Guide* for information on monitor commands). Execution cannot be resumed after a HALT.

The STOP instruction may or may not terminate program execution. If there are more program execution cycles to perform, the STOP instruction causes the **main** program to be restarted at its first step (even if the STOP instruction occurs in a subroutine). If no execution loops remain, STOP terminates the current program.

For more details on these instructions, see Program Interrupt Instructions on page 62.

Subroutines

There are three methods of exchanging information between programs:

- global variables
- soft-signals
- program argument list

When using global variables, simply use the same variable names in the different programs. Unless used carefully, this method can make program execution unpredictable and hard to debug. It also makes it difficult to write generalized subroutines because the variable names in the main program and subroutine must always be the same.

Soft-signals are internal program signals. These are digital software switches whose state can be read and set by all tasks and programs (including across CPUs in multiple CPU systems). See "Soft Signals" for details.

Exchanging information through the program argument list gives you better control over changes made to variables. It also eliminates the requirement that the variable names in the calling program be the same as the names in the subroutine. The following sections describe exchanging data through the program parameter list.

Argument Passing

There are two important considerations when passing an argument list from a calling program to a subroutine. The first is making sure the calling program passes arguments in the way the subroutine expects to receive them (mapping). The second is determining how you want the subroutine to be able to alter the variables (passing by value or reference).

Mapping the Argument List

An argument list is a list of variables or values separated by commas. The argument list passed to a calling program must match the subroutine's argument list in number of arguments and data type of each argument (see Undefined Arguments on page 31). The variable names do not have to match.

When a calling program passes an argument list to a subroutine, the subroutine does not look at the variable names in the list but the position of the arguments in the list. The argument list in the CALL statement is mapped item for item to the argument list of the subroutine. It is this mapping feature that allows you to write generalized subroutines that can be called by any number of different programs, regardless of the actual values or variable names the calling program uses.

The following figure shows the mapping of an argument list in a CALL statement to the argument list in a subroutine. The arrows indicate that each item in the list must match in position and data type but not necessarily in name. (The CALL statement argument list can include values and expressions as well as variable names.)

instruction in main program:

CALL a_routine(loc_var_a, real_var_a, 43.654, $string_var_a)

subroutine
program header

.PROGRAM a_routine(any_loc, any_real_x, any_real_y, $any_string)

instruction in main program:

CALL a_routine(loc_var_a:NULL, (real_var_a), real_var_b, $string_var_a+"")

subroutine
program header

.PROGRAM a_routine(any_loc, any_real_x, any_real_y, $any_string)

Argument Mapping

When the main program reaches the CALL instruction shown at the top of the figure, the subroutine a_routine is called and the argument list is passed as shown.

See the description of the CALL instruction in the *eV+ Language Reference Guide* for additional details on passing arrays.

Argument Passing by Value or Reference

An important principle to grasp in using subroutine calls is the way that the passed variables are affected. Variables can be changed by a subroutine, and the changed value can be passed back to the calling program. If a calling program passes a variable to a subroutine, and the subroutine can change the variable and pass the changed variable back to the calling program, the variable is said to be passed by reference. If a calling program passes a variable to a subroutine but the subroutine cannot pass the variable back in an altered form, the variable is said to be passed by value.

Variables you want changed by a subroutine should be passed by reference. In the previous figure, all the variables passed in the CALL statement are being passed by reference. Changes made by the subroutine are reflected in the state of the variables in the calling

program. Any argument that is to be changed by a subroutine and passed back to the calling routine must be specified as a variable (not an expression or value).

In addition to passing variables whose value you want changed, you will also pass variables that are required for the subroutine to perform its task but whose value you do not want changed after the subroutine completes execution. Pass these variables by value. When a variable is passed by value, a copy of the variable, rather than the actual variable, is passed to the subroutine. The subroutine can make changes to the variable, but the changes are not returned to the calling program (the variable in the calling program has the same value it had when the subroutine was called).

The following figure shows how to pass the different types of variables by value. Real numbers and integers are surrounded by parentheses, :NULL is appended to location variables, and +"" is appended to string variables.

In the following figure, real_var_b is still being passed by reference, and any changes made in the subroutine will be reflected in the calling program. The subroutine cannot change any of the other variables: it can make changes only to the copies of those variables that have been passed to it. (It is considered poor programming practice for a subroutine to change any arguments except those that are being passed back to the calling routine. If an input argument must be changed, Omron Adept suggests you make an AUTOmatic copy of the argument and work with the copy.)

```
instruction in main program:
  CALL a_routine(loc_var_a, real_var_a, 43.654, $string_var_a)




subroutine
program header
  .PROGRAM a_routine(any_loc, any_real_x, any_real_y, $any_string)
```

```
instruction in main program:
  CALL a_routine(loc_var_a:NULL, (real_var_a), real_var_b, $string_var_a+"")




subroutine
program header
  .PROGRAM a_routine(any_loc, any_real_x, any_real_y, $any_string)
```

Call by Value

Values, as well as variables, can be passed by a CALL statement. The instruction:

```
CALL a_routine(loc_1, 17.5, 121, "some string")
```

is an acceptable call to a_routine.

Undefined Arguments

If the calling program omits an argument, either by leaving a blank in the argument list (e.g., arg_1, , arg_3) or by omitting arguments at the end of a list (e.g., arg_1, arg_2), the argument are passed as undefined. The subroutine receiving the argument list can test for this value using the DEFINED function and take appropriate action.

Program Files

Since linking and compiling are not required by eV+, main programs and subroutines always exist as separate programs. The eV+ file structure allows you to keep a main program and all the subroutines it CALLs or EXECUTEs together in a single file so that when a main program is loaded, all the subroutines it calls are also loaded. (If a program calls a subroutine that is not resident in system memory, the error *Undefined program or variable name* will result.)

See the descriptions of the STORE_ commands and the MODULE command in the eV+ Operating System User's Guide for details. For an example of creating a program file, see "Sample Editing Session" on page 85.

Reentrant Programs

The eV+ system allows the same program to be executed concurrently by multiple program tasks. That is, the program can be reentered while it is already executing.

This allows different tasks that are running concurrently to use the same general-purpose subroutine.

To make a program reentrant, you must observe a few general guidelines when writing the program:

- Global variables can be read but must not be modified.

- Local variables should not be used.

- Only automatic variables and subroutine arguments can be modified.

In special situations, local variables can be used, and global variables can be modified, but then the program must explicitly provide program logic to interlock access to these variables. The TAS real-valued function (defined in Table 6-4, "System Control Functions") may be helpful in these situations. (See the *eV+ Language Reference Guide* for details.)

Recursive Programs

Recursive programs are subroutines that call themselves, either directly or indirectly. A direct call occurs when a program actually calls itself, which is useful for some special programming situations. Indirect calls are more common. They occur when program A calls program B, which eventually leads to another call to program A before program B returns. For example, an output routine may detect an error and call an error-handling routine, which in turn calls the original output routine to report the error.

If recursive subroutine calls are used, the program must observe the same guidelines as for reentrant programs (see Reentrant Programs on page 31). In addition, you must guarantee that the recursive calls do not continue indefinitely. Otherwise, the program task will run out of stack space.

Asynchronous Processing

A particularly powerful feature of eV+ is the ability to respond to an event (such as an external signal or error condition) when it occurs, without the programmer's having to include instructions to test repeatedly for the event. If event handling is properly enabled, eV+ will react to an event by invoking a specified program just as if a CALL instruction had been executed. Such a program is said to be called asynchronously, since its execution is not synchronized with the normal program flow.

Asynchronous processing is enabled by the REACT, REACTE, and REACTI program instructions. Each program task can use these instructions to prepare for independent processing of events. In addition, the optional eV+ Extensions software uses the WINDOW instruction to enable asynchronous processing of window violations when the robot is tracking a conveyor belt.

Sometimes a reaction must be delayed until a critical program section has completed. Also, since some events are more important than others, a program should be able to react to some events but not others. eV+ allows the relative importance of a reaction to be specified by a program priority value from 1 to 127. The higher the program priority setting, the more important is the reaction.

A reaction subroutine is called only if the main program priority is less than that of the reaction program priority. If the main program priority is greater than or equal to the reaction program priority, execution of the reaction subroutine is deferred until the main program priority drops. Since the main program (for example, the robot control program) normally runs at program priority zero and the minimum reaction program priority is one, any reaction can normally interrupt the main program.

The main program priority can be raised or lowered with the LOCK program instruction, and its current value can be determined with the PRIORITY real-valued function. When the main program priority is raised to a certain value, all reactions of equal or lower priority are locked out.

When a reaction subroutine is called, the main program priority is automatically set to the reaction program priority, thus preventing any reactions of equal or lower program priority

from interrupting it. When a RETURN instruction is executed in the reaction program, the main program priority is automatically reset to the level it had before the reaction subroutine was called.

For further information on reactions and program priority, see the following keywords: LOCK, PRIORITY, REACT, and REACTI in the *eV+ Language Reference Guide*.

Error Trapping

Normally, when an error occurs during execution of a program, the program is terminated and an error message is displayed on the system terminal. However, if the REACTE instruction has been used to enable an error-trapping program, the eV+ system invokes that program as a subroutine instead of terminating the program that encountered the error. (Each program task can have its own error trap enabled.)

Before invoking the error-trapping subroutine, eV+ locks out all other reactions by raising the main program priority to 254 (see Asynchronous Processing on page 32). See the description of the REACTE instruction in the *eV+ Language Reference Guide* for further information on error trapping.

The SEE Editor and Debugger

Beginning with eV+ version v2.x, eV+ programs are created (and debugged) through the ACE user interface. The eV+ Editor and Debugger tools provide a full-featured environment for creating, editing and debugging eV+ programs. For more details, see the chapter Programming ACE in the *ACE User's Guide*.

Data Types and Operators

The following topics are described in this chapter:

Introduction

This chapter describes the data typeseV+ used by eV+.

Dynamic Data Typing and Allocation

eV+ does not require you to declare variables or their data types. The first use of a variable determines its data type and allocates spaeV+ce for that variable. You can create variables and assign them a type as needed. The program instruction:

```
real_var = 13.65
```

creates the variable real_var as a real variable and assigns it the value 13.65 (if the real_var had already been created, the instruction will merely change its value).

Numeric, string, and transformation arrays up to three dimensions can be declared dynamically.

Variable Name Requirements

The requirements for a valid variable name are:

1. Keywords reserved by Omron Adept cannot be used. The *eV+ Language Reference Guide* lists the basic keywords reserved by Omron Adept. If you have ACE Sight, The *ACE Sight Reference Guide* lists the additional reserved words used by the vision system.

2. The first character of a variable name must be a letter.

3. Allowable characters after the first character are letters, numbers, periods, and the underline character.

4. Only the first 15 characters in a variable name are significant.

The following are all valid variable names:

```
x
count
dist.to.part.33
ref_frame
```

The following names are invalid for the reasons indicated:

```
3x                  (first character not a letter)
one&two             (& is an invalid name character)
pi                  (reserved word)
this_is_a_long_name (too many characters)
```

All but the last of these invalid names are rejected by eV+ with an error message. The extra-long name is truncated (without warning) to:

```
this_is_a_long_.
```

String Data Type

Variable names are preceded with a dollar ($) sign to indicate that they contain string data.[1] The program instruction:

```
$string_name = "Omron Adept eV+"
```

allocates the string variable string_name (if it had not previously been allocated) and assigns it the value Omron Adept eV+. Numbers can be used as strings with a program instruction such as:

```
$numeric_string = "13.5"
```

where numeric _string is assigned the value 13.5. The program instruction:

```
$numeric_string = 13.5
```

results in an error since you are attempting to assign a real value to a string variable.

The following restrictions apply to string constants (e.g., "a string"):

- ASCII values 32 (space) to 126 (7e) are acceptable
- ASCII 34 (") cannot be used in a string

Strings can contain from 0 to 128 characters. String variables can contain values from 0 to 255. For the interpretation of the full character set, see the section Character Sets on page 223.

The following are all valid names for string variables:

```
$x $process $prototype.names $part_1
```

The following names are invalid for strings for the reasons indicated:

```
$3x                  (first character not a letter)
$one-two             (- is an invalid name character)
factor               ($ prefix missing)
$this_is_a_long_name (too many characters)
```

All but the last of these invalid names are rejected by eV+ with an error message. The extra long name is truncated (without warning) to $this_is_a_long_.

ASCII Values

An ASCII value is the numeric representation of a **single** ASCII character. (For a complete list of the ASCII character set, see the section Character Sets on page 223.) An ASCII value is specified by prefixing a character with an apostrophe ('). Any ASCII character from the space character (decimal value 32) to the tilde character (7e, decimal value 126) can be used as an ASCII constant. Thus, the following are valid ASCII constants:

```
'A        '1        'v        '%
```

Note that the ASCII value '1 (decimal value 49) is not the same as the integer value 1 (decimal value 1.0). Also, it is not the same as the string value "1".

Functions That Operate on String Data

For a summary of eV+ functions that operate on string data, see the section String-Related Functions on page 82.

---

[1]The dollar sign is not considered in the character count of the variable name.

Real and Integer Data Types

Numbers that have a whole number and a fractional part (or mantissa and exponent if the value is expressed in scientific notation) belong to the data type real. Numeric values having only a whole number belong to the data type integer. In general, eV+ does not require you to differentiate between these two data types. If an integer is required and you supply a real, eV+ promotes the real to an integer by rounding (not truncation). Where real values are required, eV+ considers an integer a special case of a real that does not have a fractional part. The default real type is a signed, 32-bit IEEE single-precision number. Real values can also be stored as 64-bit IEEE double-precision numbers if they are specifically typed using the DOUBLE keyword (for details, see Variable Classes on page 47).

The range of integer values is:

-16,777,216 to 16,777,215

Single-precision real values have 24 bits of precision, and have the approximate range:

-1E+38 to 1E+38

Double-precision real values have 52 bits of precision, and have the approximate range:

-1E+307 to 1E+307

Numeric Representation

Numeric values can be represented in the standard decimal notation or in scientific notation, as described in the previous section.

Numeric values can also be represented in octal, binary, and hexadecimal form. The following table shows the required form for each integer representation.

Integer Value Representation

Prefi-x	Example	Representatio-n
none	-193	decimal
^B	^B1001	binary (maximum of 8 bits)
^	^346	octal
^H	^H23FF	hexadecimal
^D	^D2000000-0	double-precision

Numeric Expressions

In almost all situations where a numeric value of a variable can be used, a numeric expression can also be used. The following examples all result in x having the same value.

```
x = 3
x = 6/2
x = SQRT(9)
x = SQR(2) - 1
x = 9 MOD 6
```

Logical Expressions

eV+ does not have a specific logical (Boolean) data type. Any numeric value, variable, or expression can be used as a logical data type. eV+ considers 0 to be false and any other value to be true.

Logical Constants

There are four logical constants, TRUE and ON that will resolve to -1, and FALSE and OFF that will resolve to 0. These constants can be used anywhere that a Boolean expression is expected.

A logical value, variable, or expression can be used anywhere that a decision is required. In this example, an input signal is tested. If the signal is on (high) the variable dio.sample is given the value true, and the IF clause executes. Otherwise, the ELSE clause executes:

```
dio.sample = SIG(1001)
IF dio.sample THEN
    ; Steps to take when signal is on (high)
ELSE
    ; Steps to take when signal is off (low)
END
```

Since a logical expression can be used in place of a logical variable, the first two lines of this example could be combined to

```
IF SIG(1001) THEN
```

Functions That Operate on Numeric Data

For a summary of eV+ functions that operate on numeric data, see the section Numeric Value Functions on page 85.

Location Data Types

This section gives a brief explanation of location data. Motion Control Operations on page 99 covers locations and their use in detail.

Transformations

A data type particular to eV+ is the transformation data type. This data type is a collection of several values that uniquely identify a location in Cartesian space.

The creation and modification of location variables are discussed in Motion Control Operations on page 99.

Precision Points

Precision points are a second data type particular to eV+. A precision point is a collection of joint angles and translational values that uniquely identify the position and orientation of a robot. The difference between transformation variables and precision-point variables will become more apparent when robot motion instructions are discussed in Motion Control Operations on page 99.

Arrays

eV+ supports arrays of up to three dimensions. Any eV+ data type can be stored in an array. Like simple variables, array allocation (and typing) is dynamic. Unless they are declared to be AUTOmatic, array sizes do not have to be declared.

For example:

```
array.one[2] = 36
```

allocates space for a one-dimensional array named array.one and places the value 36 in element two of the array. (The numbers inside the brackets ([ ]) are referred to as indices. An array index can also be a variable or an expression.)

```
$array.two[4,5] = "row 4, col 5"
```

allocates space for a two-dimensional array named array.two and places row 4, col 5 in row four, column five of the array.

```
array.three[2,2,4] = 10.5
```

allocates space for a three-dimensional array named array.three and places the value 10.5 in row two, column two, range four.

If any of the above instructions were executed and the array had already been declared, the instruction would merely place the value in the appropriate location. If a data type different from the one the array was originally created with is specified, an error will result.

Arrays are allocated in blocks of 16. Thus, the instruction:

```
any_array[2] = 50
```

results in allocation of array elements 0 - 15. The instructions:

```
any_array[2] = 50
any_array[20] = 75
```

results in the allocation of array elements 0 - 31.

Array allocation is most efficient when the highest range index exceeds the highest column index, and the highest column index exceeds the highest row index. (Row is the first element, column is the second element, and range is the third element.)

Global Array Access Restriction

eV+ has a feature where global and LOCAL arrays are automatically extended as they are used. For efficiency, there is no interlocking of the array extension process between multiple tasks. A crash can occur if one task is extending or deleting an array while another is trying to access it. The AIM software application has built-in protection to prevent this problem and the resulting crash. However, custom eV+ programs must be coded to avoid this problem using one of the following methods:

Method 1

If there is a known reasonable upper-bound on the array dimensions, define (by assigning an arbitrary value to it) the highest element of the array. For multi-dimensional arrays, assign the highest element of each possible sub-array. This assignment prevents the arrays from extending.

Method 2

Use the TAS function to interlock access to the array. In this case, access to the array is handled exclusively from one or two subroutines that include the TAS to control access to the array. For details, see the TAS program instruction in the *eV+ Language Reference Guide*.

Variable Classes

In addition to having a data type, variables belong to one of three classes, GLOBAL, LOCAL, or AUTOMATIC. These classes determine how a variable can be altered by different calling instances of a program.

Global Variables

This is the default class. Unless a variable has been specifically declared to be LOCAL or AUTO, a newly created variable is considered global. Once a global variable has been initialized, it is available to any executing program[1] until the variable is deleted or all programs that reference it are removed from system memory (with a DELETE or ZERO instruction). Global variables can be explicitly declared with the GLOBAL program instruction.

```
GLOBAL DOUBLE dbl_real_var
```

Global variables are very powerful and should be used carefully and consciously. If you cannot think of a good reason to make a variable global, good programming practice dictates that you declare it to be LOCAL or AUTO.

Local Variables

Local variables are created by a program instruction similar to:

```
LOCAL the_local_var
```

where the variable the_local_var is created as a local variable. Local variables can be changed only by the program in which they are declared.

An important difference between local variables in eV+ and local variables in most other high-level languages is that eV+ local variables are local to all copies (calling instances) of a program, not just a particular calling instance of that program. This distinction is critical if you write recursive programs. In recursive programs you will generally want to use the next variable class, AUTO.

Automatic Variables

Automatic variables are created by a program instruction similar to:

```
AUTO the_auto_var
```

where the_auto_var is created as an automatic variable. Automatic variables can be changed only by a particular calling instance of a program.

AUTO statements cannot be added or deleted when the program is on the stack. See "Special Editing Situations."

```
AUTO DOUBLE dbl_auto_var
```

Automatic variables are more like the local variables of other high-level languages. If you are writing programs using a recursive algorithm, you will most likely want to use variables in the automatic class.

Scope of Variables

The scope of a variable refers to the range of programs that can see that variable. The following figureshows the scope of the different variable classes. A variable can be altered by the program(s) indicated in the shaded area of the box it is in plus any programs that are in smaller boxes. When a program declares an AUTO or LOCAL variable, any GLOBAL variables of the same name created in other programs are not accessible.


```
Any Program

    Global
   Variables

  Program Declared in

      Local
    Variables

   Copy of Program
   Declared in

     Automatic
     Variables
```
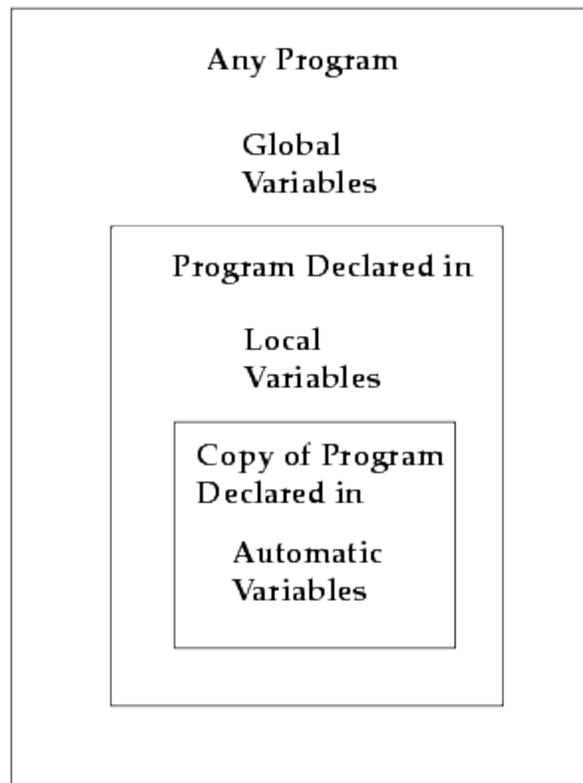
Variable Scoping

Variable Scope Example shows an example of using the various variable classes. Notice that:

- prog_1 declares a to be GLOBAL. Thus, it is available to all programs not having an AUTO or LOCAL a.

- prog_2 creates an undeclared variable b. By default, b is GLOBAL and available to other

programs not having a LOCAL or AUTO b.

- prog_3 declares an AUTO a and cannot use GLOBAL a. After prog_3 completes, the value of AUTO a is deleted.

- prog_4 declares a LOCAL a and, therefore, cannot use GLOBAL a. Unlike the AUTO a in prog_3, however, the value of LOCAL a is stored and is available for any future CALLs to prog_4.

```
.PROGRAM prog_1()                          a (GLOBAL to
   GLOBAL a                                    all programs
   a = 4 ─────────────────────                 not having AUTO
   CALL prog_2()                               or LOCAL a)


      .PROGRAM prog_2()
         GLOBAL a
         a = 7 ─────────────────
         b = 12
         CALL prog_3()


            .PROGRAM prog_3()
               AUTO a
               a = 23 ──────────────── a (AUTO defined
               b = 56                     for each time
               CALL prog_4()             program is called)

 b (GLOBAL to
    all programs
    not having AUTO        .PROGRAM prog_4()
    or LOCAL b)               LOCAL a
                             a = 2
                             CALL prog_3()


                               .PROGRAM prog_3()
                                  AUTO a
 a (LOCAL defined                 a = 23 ────── a (AUTO defined
    once for all calls            b = 56           for each time
    to "prog_4")                  CALL prog_4()   program is called)


                                    .PROGRAM prog_4()
                                       LOCAL a
                                      a = 2
                                      CALL prog_3()
```

Variable Scope Example

Variable Initialization

Before a variable can be used it must be initialized. String and numeric variables can be initialized by placing them on the left side of an assignment statement. The statements:

```
var_one = 36
$var_two = "two"
```

initializes the variables var_one and $var_two.

```
var_one = var_two
```

initializes var_one **if** var_two has already been initialized. Otherwise, an undefined value error is returned. A variable can never be initialized on the right side of an assignment statement (var_two could never be initialized by the above statement).

The statement:

```
var_one = var_one + 10
```

is valid only if var_one has been initialized in a previous statement.

Strings, numeric variables, and location variables can be initialized by being loaded from a disk file.

Strings and numeric variables can be initialized with the PROMPT instruction.

Transformations and precision points can be initialized with the SET or HERE program instructions. They can also be initialized with the HERE monitor command or with the T20 pendant. See the *eV+ Operating System Reference Guide* for information on monitor commands.

---

[1]Unless the program has declared a LOCAL or AUTO variable with the same name.

Operators

The following sections discuss the valid operators.

Assignment Operator

The equal sign (=) is used to assign a value to a numeric or string variable. The variable being assigned a value must appear by itself on the left side of the equal sign. The right side of the equal sign can contain any variable or value of the same data type as the left side, or any expression that resolves to the same data type as the left side. Any variables used on the right side of an assignment operator must have been previously initialized.

Location variables require the use of the SET instruction for a valid assignment statement. The instruction:

```
loc_var1 = loc_var2
```

is unacceptable for location and precision-point variables.

Mathematical Operators

eV+ uses the standard mathematical operators shown in the following table.

Mathematical Operators

Symbol	Function
+	addition
-	subtraction or unary minus
*	multiplication
/	division
MOD	modular (remainder) division

Relational Operators

Relational operators are used in expressions that yield a Boolean value. The resolution of an expression containing a relational operator is always -1 (true) or 0 (false) and tells you if the specific relation stated in the expression is true or false. The most common use of relational expressions is with the control structures.

eV+ uses the standard relational operators shown in the following table.

Relational Operators

Symbol	Function
==	equal to
<	less than
>	greater than
<= or =<	less than or equal to
>= or =>	greater than or equal to
<>	not equal to

If x has a value of 6 and y has a value of 10, the following Boolean expressions resolve to -1 (true):

```
x < y
y >= x
y <> x
```

and these expressions resolve to 0 (false):

```
x > y
x <> 6
x == y
```

Note the difference between the assignment operator = and the relational operator ==:

```
z = x == y
```

In this example, z is assigned a value of 0 since the Boolean expression x == y is false and would therefore resolve to 0. A relational operator never changes the value of the variables on either side of the relational operator.

Logical Operators

Logical operators affect the resolution of a Boolean variable or expression, and combine several Boolean expressions so they resolve to a single Boolean value.

eV+ uses the standard logical operators shown in the following table.

Logical Operators

Symbol	Effect
NOT	Complement the expression

	or value; makes a true expression or value false and vice versa.
AND	Both expressions must be true before the entire expression is true.
OR	Either expression must be true before the entire expression is true.
XOR	One expression must be true and one must be false before the entire expression is true.

If x = 6 and y = 10, the following expressions resolves to -1 (true):

```
NOT(x == 7)
(x > 2) AND (y =< 10)
```

And these expressions resolves to 0 (false):

```
NOT(x == 6)
(x < 2) OR (y > 10)
```

Bitwise Logical Operators

Bitwise logical operators operate on pairs of integers. The corresponding bits of each integer are compared and the result is stored in the same bit position in a third binary number.The following table lists the eV+ bitwise logical operators.

Bitwise Logical Operators

Operator	Effect
BAND	Each bit is compared using and logic. If both bits are 1, then the corresponding bit is set to 1. Otherwise, the bit is set to 0.
BOR	Each bit is compared using or logic. If either bit is 1, then the corresponding bit is set to 1. If both bits are 0, the corresponding bit is set to 0.
BXOR	Each bit is compared using exclusive or logic. If both bits are 1 or both bits are 0, the corresponding bit is set to 0. When one bit is 1 and the other is 0, the corresponding bit is set to 1.

COM	This operator works on only one number. Each bit is complemented: 1s become 0s and 0s become 1s.

Examples:

```
x = ^B1001001 BAND ^B1110011
```

results in x having a value of ^B1000001.

```
x = COM ^B100001
```

results in x having a value of ^B11110.

String Operator

Strings can be concatenated (joined) using the plus sign. For example:

```
$name = "Omron Adept "
$incorp = ", Inc."
$coname = $name + "Technologies" + $incorp
```

results in the variable $coname having the value "Omron Adept Technologies, Inc".

Order of Evaluation

Expressions containing more than one operator are not evaluated in a simple left to right manner. The following table lists the order in which operators are evaluated. Within an expression, functions are evaluated first, with expressions within the function evaluated according to the table.

The order of evaluation can be changed using parentheses. Operators within each pair of parentheses, starting with the most deeply nested pair, are completely evaluated according to the rules in the following table before any operators outside the parentheses are evaluated.

Operators on the same level in the table are evaluated strictly left to right.

Order of Operator Evaluation

Operator
NOT, COM
- (Unary minus)
*, /, MOD, AND, BAND
+, -, OR, BOR, XOR, BXOR
==, <=, >=, <, >, <>

Program Control

The following topics are described in this chapter:

Introduction

This chapter introduces the structures available in eV+ to control program execution. These structures include the looping and branching instructions common to most high-level languages as well as some instructions specific to eV+.

Unconditional Branch Instructions

There are three unconditional branching instructions in eV+:

- GOTO
- CALL
- CALLS

GOTO

The GOTO instruction causes program execution to branch immediately to a program label instruction somewhere else in the program. The syntax for GOTO is:

```
GOTO label
```

label   is an integer entered at the beginning of a line of program code. **label** is not the same as the program step numbers: Step numbers are assigned by the system; labels are entered by the programmer as the opening to a line of code. In the next code example, the numbers in the first column are program step numbers. The numbers in the second column are program labels.

```
61      .
62      GOTO 100
63      .
64      .
65 100 TYPE "The instruction GOTO 100 got me he re."
66      .
```

A GOTO instruction can branch to a label before or after the GOTO instruction.

GOTO instructions can make program logic difficult to follow and debug, especially in a long, complicated program with many subroutine calls. Use GOTO instructions with care. A common use of GOTO is as an exit routine or exit on error instruction.

CALL

The CALL and CALLS instructions are used in eV+ to implement subroutine calls. The CALL instruction causes program execution to be suspended and execution of a new program to begin. When the new program has completed execution, execution of the original program resumes at the instruction after the CALL instruction. The details of subroutine creation, execution, and parameter passing are covered in Subroutines on page 28. The simplified syntax for a CALL instruction is:

```
CALL program(arg_list)
```

program   is the name of the program to be called. The program name must be specified exactly, and the program being CALLed must be resident in system memory.

    arg_list   is the list of arguments being passed to the subroutine. These arguments can be passed either by value or by reference and must agree with the arguments expected by the program being called. Subroutines and argument lists are described in "Subroutines."

The code:

```
48       .
49       CALL check_data(locx, locy, length)
50       .
```

suspends execution of the calling program, passes the arguments locx, locy, and length to program check_data, executes check_data, and (after check_data has completed execution) resumes execution of the calling program at step 50.

CALLS

The CALLS instruction is identical to the CALL instruction except for the specification of **program**. For a CALLS instruction, **program** is a string value, variable, or expression. This allows you to call different subroutines under different conditions using the same line of code. (These different subroutines must have the same arg_list.) You can use this technique to create "virtual functions" in object oriented languages like C++, C#, Java or Python.

The code:

```
47       .
48       $program_name = $program_list[program_select]
49       CALLS $program_name(length, width)
50       .
```

suspends execution of the calling program, passes the parameters length and width to the program specified by array index program_select from the array $program_list, executes the specified program, and resume execution of the calling program at step 50.

Program Interrupt Instructions

eV+ provides several ways of suspending or terminating program execution. A program can be put on hold until a specific condition becomes TRUE using the WAIT instruction. A program can be put on hold for a specified time period or until an event is generated in another task by the WAIT.EVENT instruction. A program can be interrupted based on a state transition of a digital input signal with the REACT and REACTI instructions. Program errors can be intercepted and handled with a REACTE instruction. Program execution can be terminated with the HALT, STOP, and PAUSE commands. These instructions interrupt the program in which they are contained. Any programs running as other tasks are not affected. Robot motion can be controlled with the BRAKE, BREAK, and DELAY instructions. (The ABORT and PROCEED monitor commands can also be used to suspend and proceed programs, see the *eV+ Operating System Reference Guide* for details.)

WAIT

WAIT suspends program execution until a condition (or conditions) becomes true.

```
WAIT SIG(1032, -1028)
```

delays execution until digital input signal 1032 is on and 1028 is off.

```
WAIT TIMER(1) > 10
```

suspends execution until timer 1 returns a value greater than 10.

WAIT.EVENT

The instruction:

```
WAIT.EVENT , 3.7
```

suspends execution for 3.7 seconds. This wait is more efficient than waiting for a timer (as in the previous example) because the task does not have to loop continually to check the timer value.

The instruction:

```
WAIT.EVENT
```

suspends execution until another task issues a SET.EVENT instruction to the waiting task. If the SET.EVENT does not occur, the task waits indefinitely. For more details on SET.EVENT, see the *eV+ Language Reference Guide*.

REACT and REACTI

When a REACT or REACTI instruction is encountered, the program begins monitoring a digital input signal specified in the REACT instruction. This signal is monitored in the background with program execution continuing normally until the specified signal transitions. When (and

if) a transition is detected, the program suspends execution at the currently executing step. REACT and REACTI suspend execution of the current program and call a specified subroutine. Additionally, REACTI issues a BRAKE instruction to immediately stop the current robot motion.

Both instructions specify a subroutine to be run when the digital transition is detected. After the specified subroutine has completed, program execution resumes at the step executing when the digital transition was detected.

Digital signals 1001 - 1012 and 2001 - 2008 can be used for REACT instructions.

The signal monitoring initiated by REACT/REACTI is in effect until another REACT/REACTI or IGNORE instruction is encountered. If the specified signal transition is not detected before an IGNORE or second REACT/REACTI instruction is encountered, the REACT/REACTI instruction has no effect on program execution.

The syntax for a REACT or REACTI instruction is:

```
REACT signal_number, program, priority
```

signal_ number	digital input signal in the range 1001 to 1012 or 2001 to 2008.
program	the subroutine (and its argument list) that is to be executed when a react is initiated.
priority	number from 1 to 127 that indicates the relative importance of the reaction.

The following code implements a REACT routine:

```
35      ; Look for a change in signal 1001 from "on" to "off".
36      ;       Call subroutine "alarm if a change is detected.
37      ;       Set priority of "alarm" to 10 (default would be 1).
38      ;       The main program has default priority of 0.
39
40              REACT -1001, alarm, 10
41
42      ; REACT will be in effect for the following code
43
44              MOVE a
45              MOVE b
46              LOCK 20        ;Defer any REACTions to "alarm"
47              MOVE c
48              MOVE d
49              LOCK 0         ;Allow REACTions
50              MOVE e
51
52      ; Disable monitoring of signal 1001
53
54              IGNORE -1001
55 .
```

If signal 1001 transitions during execution of step 43, step 43 completes, the subroutine alarm is called, and execution resumes at step 44.

If signal 1001 transitions during execution of step 47, steps 47, 48, and 49 completes (since the program had been given a higher priority than REACT), the subroutine alarm is called, and execution resumes at step 50.[1]

REACTE

REACTE enables a reaction program that is run whenever a system error that causes program execution to terminate is encountered. This includes all robot errors, hardware errors, and most system errors (it does NOT include I/O errors).

Unlike REACT and REACTI, REACTE cannot be deferred based on priority considerations. The instruction:

```
REACTE trouble
```

enables monitoring of system errors and execute the program **trouble** whenever a system error is generated.

HALT, STOP, and PAUSE

When a HALT instruction is encountered, program execution is terminated, and any open serial or disk units are DETACHED and FCLOSEd. PROCEED or RETRY will not resume execution.

When a STOP instruction is encountered, execution of the current program cycle is terminated and the next execution cycle resumes at the first step of the program. If the STOP instruction is encountered on the last execution cycle, program execution is terminated, and any open serial or disk units are DETACHED and FCLOSEd. PROCEED or RETRY will not resume execution. (See EXECUTE for details on execution cycles.) When a PAUSE instruction is encountered, execution is suspended. After a PAUSE, the system prompt appears and Monitor Commands can be executed. This allows you to verify the values of program variables and set system parameters. This is useful during program debugging. The monitor command PROCEED resumes execution of a program interrupted with the PAUSE command.

BRAKE, BREAK, and DELAY

BRAKE aborts the current robot motion. This instruction can be issued from any task. Program execution is not suspended and the program (executing as task 0) continues to execute at the next instruction. BREAK suspends program execution (defeats forward processing) until the current robot motion is completed. This instruction can be executed only from a robot control program and is used when completion of the current robot motion must occur before execution of the next instruction. A DELAY instruction specifies the minimum delay between robot motions (not program instructions).

Additional Program Interrupt Instructions

You can specify a parameter in the instruction line for the I/O instructions ATTACH, READ, GETC, and WRITE that causes the program to suspend until the I/O request has been successfully completed.

Program Interrupt Example

The following figure shows how the task and program priority scheme works. It also shows how the asynchronous and program interrupt instructions work within the priority scheme. The example makes the following assumptions:

- Task 1 runs in all time slices at priority 30

- Task 2 runs in all time slices at priority 20

- All system tasks are ignored

- All system interrupts are ignored

The illustration shows the time lines of executing programs. A solid line indicates a program is running, and a dotted line indicates a program is waiting. The Y axis shows the program priority. The X axis is divided into 16-millisecond major cycles. The example shows two tasks executing concurrently with REACT routines enabled for each task. Note how the LOCK instructions and triggering of the REACT routines change the program priority.

The sequence of events for the example is:

1. Task 1 is running program prog_a at program priority 0. A reaction program based on signal 1003 is enabled at priority 5.

2. Signal 1003 is asserted externally. The signal transition is not detected until the next major cycle.

3. The signal 1003 transition is detected. The task 1 reaction program begins execution, interrupting prog_a.

4. The task 1 reaction program reenables itself and completes by issuing a RETURN instruction. prog_a resumes execution in task 1.

5. Task 1 prog_a issues a CLEAR.EVENT instruction followed by a WAIT.EVENT instruction to wait for its event flag to be set. Task 1 is suspended, and task 2 resumes execution of prog_b. Task 2 has a reaction program based on signal 1010 enabled at priority 5.

6. Task 2 prog_b issues a LOCK 10 instruction to raise its program priority to level 10.

7. Signal 1010 is asserted externally. The signal transition is not detected until the next major cycle.

8. The signal 1010 transition is detected, and the task 2 reaction is triggered. However,

since the reaction is at level 5 and the current program priority is 10, the reaction execution is deferred.

9. Task 2 prog_b issues a LOCK 0 instruction to lower its program priority to level 0. Since a level 5 reaction program is pending, it begins execution immediately and sets the program priority to 5.

10. Signal 1003 is asserted externally. The signal transition is not detected until the next major cycle.

11. The signal 1003 transition is detected which triggers the task 1 reaction routine and also sets the task 1 event flag. Since task 1 has a higher priority (30) than task 2 (20), task 1 begins executing its reaction routine and task 2 is suspended.

12. The task 1 reaction routine completes by issuing a RETURN instruction. Control returns to prog_a in task 1.

13. Task 1 prog_a issues a CLEAR.EVENT instruction followed by a WAIT.EVENT instruction to wait for its event flag to be set. Task 1 is suspended and task 2 resumes execution of its reaction routine.
The task 2 reaction routine completes by issuing a RETURN instruction. Control returns to prog_b in task 2.

14. Task 2 prog_b issues a SET.EVENT 1 instruction, setting the event flag for task 1. Task 2 now issues a RELEASE program instruction to yield control of the CPU.

15. Since the task 1 event flag is now set, and its priority is higher than task 2, task 1 resumes execution, and task 2 is suspended.
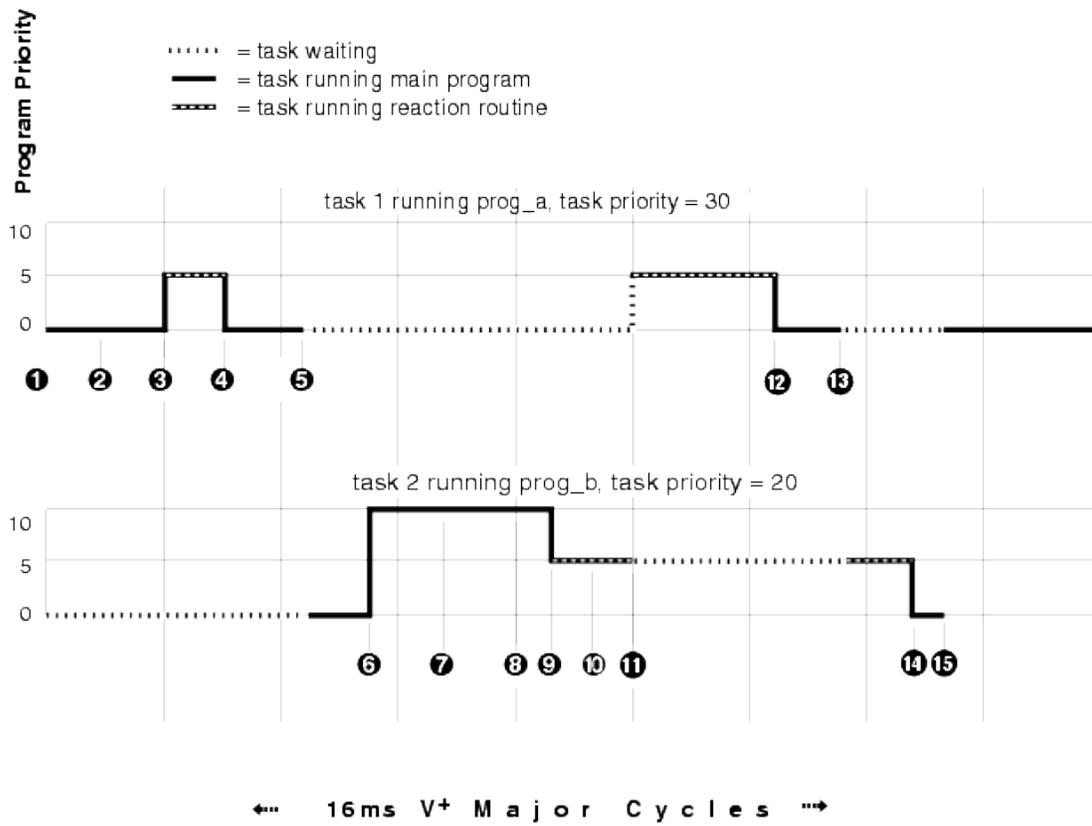
Priority Example 2

---

[1]The LOCK instruction can be used to control execution of a program after a REACT or REACTI subroutine has completed.

Logical (Boolean) Expressions

The next two sections discuss program control structures whose execution depends on an expression or variable that takes on a Boolean value (a variable that is either true or false, or an expression that resolves to true or false). An expression can take into account any number of variables or digital input signals as long as the final resolution of the expression is a Boolean value. In eV+, any number (real or integer) can satisfy this requirement. Zero is considered false; any nonzero number is considered true. There are four system constants, TRUE and ON that resolve to -1, and FALSE and OFF, that resolve to 0.

Examples of valid Boolean expressions:

```
y > 32
NOT(y > 32)
x == 56
x AND y
(x AND y) OR (var1 < var2)
-1
```

For details on eV+ relational operators, see Relational Operators on page 51.

Conditional Branching Instructions

Conditional branching instructions allow you to execute blocks of code based on the current values of program variables or expressions. eV+ has three conditional branch instructions:

- IF...GOTO
- IF...THEN...ELSE
- CASE value OF

IF...GOTO

IF...GOTO behaves similarly to GOTO, but a condition can be attached to the branch. If the instruction:

```
IF logical_expression GOTO 100
```

is encountered, the branch to label 100 occurs only if logical_expression has a value of true.

IF...THEN...ELSE

The basic conditional instruction is the IF...THEN...ELSE clause. This instruction has two forms:

```
IF expression THEN
    code block (executed when expression is true)
END

IF expression THEN
    code block (executed when expression is true)
ELSE
    code block (executed when expression is false)
END
```

expression is any well-formed Boolean expression (described above).

In the following example, if program execution reaches step 59 and num_parts is greater than 75, step 60 is executed. Otherwise, execution resumes at step 62.

```
56      .
57      ;CALL "check_num" if "num_parts" is greater than 75
58
59      IF num_parts > 75 THEN
60              CALL check_num(num_parts)
61      END
62      .
```

In the following example, if program execution reaches step 37 with input signal 1033 on and need_part true, the program executes steps 38 to 40 and resumes at step 44. Otherwise, it executes step 42 and resumes at step 44.

```
32     .
33     ; If I/O signal 1033 is on and Boolean "need_part" is
34     ;        true, then pick up the part
35     ; else alert the operator.
36
37     IF SIG(1033) AND need_part THEN
38             MOVE loc1
39             CLOSEI
40             DEPART 50
41     ELSE
42             TYPE "Part not picked up."
43     END
44     .
```

CASE value OF

The IF...THEN...ELSE structure allows a program to take one of two different actions. The CASE structure will allow a program to take one of many different actions based on the value of a variable. The variable used must be a real or an integer. The form of the CASE structure is:

```
CASE target OF
    VALUE list_of_values:
         code block (executed when target is in list_of_values)
    VALUE list_of_values:
         code block (executed when target is in list_of_values)
         ...
  ANY
   code block (executed when target not in any list_of_values)
END
```

real value to match.

list_of_valueslist (separated by commas) of real values. If one of the values in the list equals target, the code following that value statement is executed.

Example
```
5     ; Create a menu structure using a CASE statement
66
67 50  TYPE "1. Execute the program."
68     TYPE "2. Execute the programmer."
69     TYPE "3. Execute the computer."
70     PROMPT "Enter menu selection.", select
71
72     CASE select OF
73       VALUE 1:
74             CALL exec_program()
75       VALUE 2:
76             CALL exec_programmer()
77       VALUE 3:
78             CALL exec_computer()
79       ANY
```

```
80              PROMPT "Entry must be from 1 to 3", select
81              GOTO 50
82      END
83      .
```

If the above code is rewritten without an ANY statement, and a value other than 1, 2, or 3 is entered, the program continues to execute at step 83 without executing any program.

Looping Structures

In many cases, you will want the program to execute a block of code more than once. eV+ has three looping structures that allow you to execute blocks of code a variable number of times. The three instructions are:

- FOR

- DO...UNTIL

- WHILE...DO

FOR

A FOR instruction creates an execution loop that will execute a given block of code a specified number of times. The basic form of a FOR loop is:

```
FOR index = start_val TO end_val STEP incr
    .
    code block
    .
END
```

index	is a real variable that keeps track of the number of times the FOR loop has been executed. This variable is available for use within the loop.
start_val	is a real expression for the starting value of the **index**.
end_val	is a real expression for the ending value of the **index**. Execution of the loop terminates when **index** reaches this value.
incr	is a real expression indicating the amount **index** is to be incremented after each execution of the loop. The default value is 1.

Examples

```
88      .
89      ; Output even elements of array "$names" (up to index 32)
90
91        FOR i = 2 TO 32 STEP 2
92            TYPE $names[i]
93        END
94      .
        .
102     .
103     ; Output the values of the 2 dimensional array "values" in
104     ; column and row form (10 rows by 10 columns)
105     .
106       FOR i = 1 TO 10
107         FOR j = 1 to 10
108             TYPE values[i,j], /S
109         END
```

```
110        TYPE " ", /C1
111      END
112        .
```

A FOR loop can be made to count backward by entering a negative value for the step increment.

```
13      .
14    ; Count backward from 10 to 1
15
16      FOR i = 10 TO 1 STEP -1
17          TYPE i
18      END
19      .
```

Changing the value of **index** inside a FOR loop will cause the loop to behave improperly. To avoid problems with the **index**, make the **index** variable an auto variable and do not change the **index** from inside the FOR loop. Changes to the starting and ending variables do not affect the FOR loop once it is executing.

DO...UNTIL

DO...UNTIL is a looping structure that executes a given block of code an indeterminate number of times. Termination of the loop occurs when the Boolean expression or variable that controls the loop becomes true. The Boolean is tested after each execution of the code block-if the expression evaluates to true, the loop is not executed again. Since the expression is not evaluated until after the code block has been executed, the code block will always execute at least once. The form for this looping structure is:

```
DO
    .
    code block
    .
UNTIL expression
```

expressionis any well-formed Boolean expression. This **expression** must eventually evaluate to true, or the loop executes indefinitely.

```
20      .
21    ; Output the numbers 1 to 100 to the screen
22
23      x = 1
24      DO
25          TYPE x
26          x = x + 1
27      UNTIL x > 100
28      .
```

Step 26 ensures that x will reach a high enough value so that the expression x > 100 becomes true.

```
43      .
44    ; Echo up to 15 characters to the screen. Stop when 15
45    ; characters or the character "#" have been entered.
```

```
46
47       x = 1
48       DO
49           PROMPT "Enter a character: ", $ans
50           TYPE $ans
51           x = x + 1
52        UNTIL (x > 15) OR ($ans == "#")
53           .
```

In this code, either x reaching 15 or # being entered at the PROMPT instruction terminates the loop. As long as the operator enters enough characters, the loop terminates.

WHILE...DO

WHILE...DO is a looping structure similar to DO...UNTIL except the Boolean expression is evaluated at the beginning of the loop instead of at the end. This means that if the condition indicated by the expression is true when the WHILE...DO instruction is encountered, the code within the loop will be executed.

WHILE...DO loops are susceptible to infinite looping just as DO...UNTIL loops are. The expression controlling the loop must eventually evaluate to true for the loop to terminate. The form of the WHILE...DO looping structure is:

```
WHILE expression DO
    code block
END
```

expressionis any well-formed Boolean expression as described at the beginning of this section.

The following code shows a WHILE...DO loop being used to validate input. Since the Boolean expression is tested before the loop is executed, the code within the loop will be executed only when the operator inputs an unacceptable value at step 23.

```
20       .
21     ; Loop until operator inputs value in the range 32-64
22
23       PROMPT "Enter a number in the range 32 to 64.", ans
24       WHILE (ans < 32) OR (ans > 64) DO
25           PROMPT "Number must be in the range 32-64.", ans
26       END
27       .
```

In the above code, an operator could enter a nonnumeric value, in which case the program execution would stop. A more robust strategy would be to use a string variable in the PROMPT instruction and then use the $DECODE and VAL functions to evaluate the input.

In the following code, if digital signal 1033 is on when step 69 is reached, the loop does not execute, and the program continues at step 73. If digital signal 1033 is off, the loop executes continually until the signal comes on.

```
65                      .
66     ; Create a busy loop waiting for signal
```

```
67      ; 1033 to turn "on"
68        WHILE NOT SIG(1033) DO
69
70          ;Wait for signal
71
72        END
73        .
```

Summary of Program Control Keywords

The following table summarizes the program control instructions. See the *eV+ Language Reference Guide* for details on these commands.

Program Control Operations

Keyword	Type	Function
ABORT	Program Instruction	Terminate execution of a control program.
CALL	Program Instruction	Suspend execution of the current program and continue execution with a new program (that is, a subroutine).
CALLS	Program Instruction	Suspend execution of the current program and continue execution with a new program (that is, a subroutine) specified with a string value.
CASE	Program Instruction	Initiate processing of a CASE structure by defining the value of interest.
CLEAR.EVENT	Program Instruction	Clear an event associated with the specified task.
CYCLE.END	Program Instruction	Terminate the specified control program the next time it executes a STOP program instruction (or its equivalent). Suspend processing of an application program or command program until a program completes execution.
DO	Program Instruction	Introduce a DO program structure.
EXECUTE	Program Instruction	Begin execution of a control program.
EXIT	Program Instruction	Exit a FOR, DO, or WHILE control structure.
FOR	Program Instruction	Execute a group of program instructions a certain number of times.
GET.EVENT	Real-Valued	Return events that are set for the specified task.

Keyword	Type	Function
	Function	
GOTO	Program Instruction	Perform an unconditional branch to the program step identified by the given label.
HALT	Program Instruction	Stop program execution and do not allow the program to be resumed.
IF...GOTO	Program Instruction	Branch to the specified label if the value of a logical expression is TRUE (nonzero).
IF...THEN	Program Instruction	Conditionally execute a group of instructions (or one of two groups) depending on the result of a logical expression.
LOCK	Program Instruction	Set the program reaction lock-out priority to the value given.
MCS	Program Instruction	Invoke a monitor command from a control program.
NEXT	Program Instruction	Break a FOR, DO, or WHILE structure and start the next iteration of the control structure.
PAUSE	Program Instruction	Stop program execution but allow the program to be resumed.
PRIORITY	Real-Valued Function	Return the current reaction lock-out priority for the program.
REACT REACTI	Program Instruction	Initiate continuous monitoring of a specified digital signal and automatically trigger a subroutine call if the signal transitions properly.
REACTE	Program Instruction	Initiate the monitoring of errors that occur during execution of the current program task.
RELEASE	Program Instruction	Allow the next available program task to run.
RETURN	Program	Terminate execution of the current subroutine and

Keyword	Type	Function
	Instruction	resume execution of the last-suspended program at the step following the CALL or CALLS instruction that caused the subroutine to be invoked.
RETURNE	Program Instruction	Terminate execution of an error reaction subroutine and resume execution of the last-suspended program at the step following the instruction that caused the subroutine to be invoked.
RUNSIG	Program Instruction	Turn on (or off) the specified digital signal as long as execution of the invoking program task continues.
SET.EVENT	Program Instruction	Set an event associated with the specified task.
STOP	Program Instruction	Terminate execution of the current program cycle.
WAIT	Program Instruction	Put the program into a wait loop until the condition is TRUE.
WAIT.EVENT	Program Instruction	Suspend program execution until a specified event has occurred, or until a specified amount of time has elapsed.
WHILE	Program Instruction	Initiate processing of a WHILE structure if the condition is TRUE or skipping of the WHILE structure if the condition is initially FALSE.

Functions

The following topics are described in this chapter:

Using Functions

eV+ provides you with a wide variety of predefined functions for performing string, mathematical, and general system parameter manipulation. In most cases, you must provide the data that is input to a function. The function then returns a value based on a specific operation on that data. Functions can be used anywhere a value or expression would be used.

Variable Assignment Using Functions

The instruction:

```
$curr_time = $TIME()
```

puts the current system time into the variable $curr_time. This is an example of a function that does not require any input data. The instruction:

```
var_root = SQRT(x)
```

puts the square root of the value x into var_root. X is not changed by the function.

Functions Used in Expressions

A function can be used wherever an expression can be used (as long as the data type returned by the function is the correct type). The instruction:

```
IF LEN($some_string) > 12 THEN
```

results in the Boolean expression being true if the string $some_string has more than 12 characters. The instruction:

```
array_var = some_array[VAL($x)]
```

results in array_var having the same value as the array cell $x. (VAL converts a string to a real.)

Functions as Arguments to a Function

In most cases, the values passed to a function are not changed. This not only protects the variables you use as arguments to a function, but also allows you to use a function as an argument to a function (so long as the data type returned is the type expected by the function). The following example results in i having the absolute value of x. (i = D(-22) = 2).

```
i = SQRT(SQR(x))
```

String-Related Functions

The value returned from a string function may be another string or a numeric value.

String-Related Functions

Keyword	Function
ASC	Return a single character value from within a string.
$CHR	Return a one-character string having a given value.
DBLB	Return the value of eight bytes of a string interpreted as an IEEE double-precision floating-point number.
$DBLB	Return an 8-byte string containing the binary representation of a real value in double-precision IEEE floating-point format.
$DECODE	Extract part of a string as delimited by given break characters.
$ENCODE	Return a string created from output specifications. The string produced is similar to the output of a TYPE instruction.
FLTB	Return the value of four bytes of a string interpreted as an IEEE single-precision floating-point number.
$FLTB	Return a 4-byte string containing the binary representation of a real value in single-precision IEEE floating-point format.
$INTB	Return a 2-byte string containing the binary representation of a 16-bit integer.
LEN	Return the number of characters in the given string.
LNGB	Return the value of four bytes of a string interpreted as a signed 32-bit binary integer.
$LNGB	Return a 4-byte string containing the binary representation of a 32-bit integer.
$MID	Return a substring of the specified string.
PACK	Replace a substring within an array of (128-character) string variables or within a (non-array) string variable.

Keyword	Function
POS	Return the starting character position of a substring in a string.
$TRANSB	Return a 48-byte string containing the binary representation of a transformation value.
$TRUNCATE	Return all characters in the input string until an ASCII NUL (or the end of the string) is encountered.
$UNPACK	Return a substring from an array of 128-character string variables.
VAL	Return the real value represented by the characters in the input string.

Examples of String Functions

The instruction:

```
TYPE $ERROR(-504)
```

outputs the text *Unexpected end of file* to the screen.

The instructions:

```
$message = "The length of this line is: "
TYPE $ENCODE($message, /I0, LEN($message)+14), "
characters."
```

output the message:

```
The length of this line is: 42 characters.
```

Location, Motion, and External Encoder Functions

eV+ provides numerous functions for manipulating and converting location variables. See Motion Control Operations for details on motion processing and a table that includes all location-related functions. For details on the external encoders, see Reading Device Data on page 220.

Examples of Location Functions

The instruction:

```
rotation = RZ(HERE)
```

places the value of the current rotation about the Z axis in the variable rotation.

The instruction:

```
dist = DISTANCE(HERE, DEST)
```

places the distance between the motion device's current location and its destination (the value of the next motion instruction) in the variable dist.

The instructions:

```
IF INRANGE(loc_1) == 0 THEN
    IF SPEED(2) > 50 THEN
        SPEED 50
    END
    MOVE(loc_1)
END
```

ensures that loc_1 is reachable and moves the motion device to that location at a program speed not exceeding 50.

Numeric Value Functions

The functions listed in the following table provide trigonometric, statistical, and data type conversion operations. For additional details on arithmetic processing, see Data Types and Operators on page 37.

Numeric Value Functions

Keyword	Function
ABS	Return absolute value.
ATAN2	Return the size of the angle (in degrees) that has its trigonometric tangent equal to value_1/value_2.
BCD	Convert a real value to Binary Coded Decimal (BCD) format.
COS	Return the trigonometric cosine of a given angle.
DCB	Convert BCD digits into an equivalent integer value.
FRACT	Return the fractional part of the argument.
INT	Return the integer part of the value.
MAX	Return the maximum value contained in the list of values.
MIN	Return the minimum value contained in the list of values.
OUTSIDE	Test a value to see if it is outside a specified range.
PI	Return the value of the mathematical constant pi (3.141593).
RANDOM	Return a pseudorandom number.
SIGN	Return the value 1 with the sign of the value parameter.
SIN	Return the trigonometric sine of a given angle.
SQR	Return the square of the parameter.
SQRT	Return the square root of the parameter.

Examples of Arithmetic Functions

The instructions:

```
$a = "16"
x = SQRT(VAL($a))
```

results in x having a value of 4.

The instruction:

```
x = INT(RANDOM*10)
```

creates a pseudorandom number between 0 and 10.

Logical Functions

The following table lists the functions that return Boolean values. These functions require no arguments and essentially operate as system constants.

Logical Functions

Keyword	Function
FALSE	Return the value used by eV+ to represent a logical false result.
OFF	Return the value used by eV+ to represent a logical false result.
ON	Return the value used by eV+ to represent a logical true result.
TRUE	Return the value used by eV+ to represent a logical true result.

System Control Functions

The functions listed in the following table return information about the system and system parameters.

System Control Functions

Keyword	Function
DEFINED	Determine whether a variable has been defined.
ERROR	Return the error number of a recent error that caused program execution to stop or caused a REACTE reaction.
$ERROR	Return the error message associated with the given error code.
FREE	Return the amount of unused free memory storage space.
GET.EVENT	Return events that are set for the specified task.
ID	Return values that identify the configuration of the current system.
$ID	Return the system creation date and edit/revision information.
INTB	Return the value of two bytes of a string interpreted as a signed 16-bit binary integer.
LAST	Return the highest index used for an array (dimension).
PARAMETER	Return the current setting of the named system parameter.
PRIORITY	Return the current reaction lock-out priority for the program.
SELECT	Return the unit number that is currently selected by the current task for the device named.
STATUS	Return status information for an application program.
SWITCH	Return an indication of the setting of a system switch.
TAS	Return the current value of a real-valued variable and assign it a new value. The two actions are done indivisibly so no other program task can modify the variable at the same time.

Keyword	Function
TASK	Return information about a program execution task.
TIME	Return an integer value representing either the date or the time specified in the given string parameter.
$TIME	Return a string value containing either the current system date and time or the specified date and time.
TIMER	Return the current time value of the specified system timer.
TPS	Return the number of ticks of the system clock that occur per second (Ticks Per Second).

Example of System Control Functions

The instruction:

```
IF (TIMER(2) > 100) AND (DEFINED(loc_1)) THEN
     MOVE loc_1
END
```

executes the MOVE instruction only if timer (2) had a value greater than 100 and the variable loc_1 had been defined.

Switches and Parameters

The following topics are described in this chapter:

Introduction

System parameters determine certain operating characteristics of the eV+ system. These parameters have numeric values that can be changed from the command line or from within a program to suit particular system configurations and needs. The various parameters are described in this chapter along with the operations for displaying and changing their values.

System switches are similar to system parameters in that they control the operating behavior of the eV+ system. Switches differ from parameters, however, in that they do not have numeric values. Switches can be set to either enabled or disabled, which can be thought of as on and off, respectively.

All the basic system switches are described in this chapter. The monitor commands and program instructions that can be used to display and change their settings are also presented.

Parameters

See the *eV+ Language Reference Guide* for more detailed descriptions of the keywords discussed here.

Whenever a system parameter name is used, it can be abbreviated to the minimum length required to identify the parameter. For example, the HAND.TIME parameter can be abbreviated to H, since no other parameter name begins with H.

Viewing Parameters

To see the state of a single parameter, use the PARAMETER monitor command:

```
PARAMETER parameter_name
```

If parameter_name is omitted, the value of all parameters is displayed.

To retrieve the value of a parameter from within a program, use the PARAMETER function. The instruction:

```
TYPE "HAND.TIME parameter =", PARAMETER(HAND.TIME)
```

will display the current setting of the hand-delay parameter in the monitor window.

The PARAMETER function can be used in any expression to include the value of a parameter. For example, the following program statement increases the delay for hand actuation:

```
PARAMETER HAND.TIME = PARAMETER(HAND.TIME) + 0.15
```

Note that the left-hand occurrence of PARAMETER is the instruction name and the right-hand occurrence is the function name.

Setting Parameters

To set a parameter from the command line, use the PARAMETER monitor command. The command:

```
PARAMETER HAND.TIME = 0.5
```

sets the hand operation delay time to 0.5 seconds.

To set a parameter in a program, use the PARAMETER program instruction. The instruction:

```
PARAMETER NOT.CALIBRATED = 1
```

asserts the not calibrated state for robot 1.

Some parameters are organized as arrays and must be accessed by specifying an array index.

Summary of Basic System Parameters

System parameters are set to defaults when the eV+ system is initialized. The default values are indicated with each parameter description below. The settings of the parameter values are not affected by the ZERO command.

If your robot system includes optional enhancements (such as vision), you will have other system parameters available. Consult the documentation for the options for details. The basic system parameters are shown in the following table.

Basic System Parameters

Parameter	Use	De-fault	Min	Max
BELT.MODE	Controls the operation of the conveyor tracking feature of the eV+ system.	0	0	14
HAND.TIME	Determines the duration of the motion delay that occurs during processing of OPENI, CLOSEI, and RELAXI instructions. The value for this parameter is interpreted as the number of seconds to delay. Due to the way in which eV+ generates its time delays, the HAND.TIME parameter is internally rounded to the nearest multiple of 0.016 seconds.	0.05	0	1E18
NOT.CALIBRATED	Represents the calibration status of the robot(s) controlled by the eV+ system.	7	0	7

Switches

System switches govern various features of the eV+ system. The switches are described below. See the *eV+ Language Reference Guide* and the *eV+ Operating System Reference Guide* for more detailed descriptions of the keywords discussed here.

As with system parameters, the names of system switches can be abbreviated to the minimum length required to identify the switch.

Viewing Switch Settings

The SWITCH monitor command displays the setting of one or more system switches:

```
SWITCH switch_name, ..., switch_name
```

If no switches are specified, the settings of all switches are displayed.

Within programs, the SWITCH real-valued function returns the status of a switch. The instruction:

```
SWITCH(switch_name)
```

returns TRUE (-1.0) if the switch is enabled, FALSE (0.0) if the switch is disabled.

Some switches are organized as arrays and may be accessed by specifying the array index.

Setting Switches

The ENABLE and DISABLE monitor commands/program instructions control the setting of system switches. The instruction:

```
ENABLE BELT
```

enables the BELT switch. The instruction:

```
DISABLE BELT, CP
```

disables the CP and BELT switches. Multiple switches can be specified for either instruction.

Switches can also be set with the SWITCH program instruction. Its syntax is:

```
SWITCH switch_name = value
```

This instruction differs from the ENABLE and DISABLE instructions in that the SWITCH instruction enables or disables a switch depending on the value on the right-hand side of the equal sign. This allows you to set switches based on a variable or expression. The switch is enabled if the value is TRUE (nonzero) and disabled if the value is FALSE (zero). The instruction:

```
SWITCH CP = SIG(1001)
```

enables the continuous path (CP) switch if input signal 1001 is on.

Summary of Basic System Switches

The default switch settings at system power-up are given in the following table. (The switch settings are not affected by the ZERO command.)

Optional enhancements to your eV+ system may include additional system switches. If so, they are described in the documentation for the options.

Basic System Switches

Switch	Use
AUTO.POWER.OFF	When this switch is enabled eV+ will treat software errors as hard errors and disable HIGH POWER. Normally these errors stop the robot and signal the eV+ program, but DO NOT cause HIGH POWER to be turned off. The soft errors are: (-624) *force protect limit exceeded (-1003) *Time-out nulling errors* Mtr (-1006) *Soft envelope error* Mtr
BELT	Used to turn on the conveyor tracking features of eV+ (if the option is installed).This switch must be enabled before any of the special conveyor tracking instructions can be executed. When BELT is disabled, the conveyor tracking software has a minimal impact on the overall performance of the system. Default is disabled.
CP	Enable/disable continuous-path motion processing (see "Continuous-Path Trajectories"). Default is enabled.
DECEL.100	When DECEL.100 is enabled for a robot, the maximum deceleration percentage defined by SPEC is ignored and a maximum deceleration of 100% is used instead. This maximum deceleration value is used to limit the value specified by the ACCEL program instruction. For backwards compatibility, by default, DECEL.100 is disabled for all robots.
DRY.RUN	Enable/disable sending of motion commands to the robot. Enable this switch to test programs for proper logical flow and correct external communication without having to worry about the robot running into something. (Also see the TRACE switch, which is useful during program checkout.) The manual control pendant can still be used to move the robot when DRY.RUN is enabled.

Switch	Use
	Default is disabled.
FORCE	Controls whether the (optional) stop-on-force feature of the eV+ system is active. Default is disabled.
MESSAGES	Controls whether output from TYPE instructions will be displayed on the terminal. Default is enabled.
POWER	Tracks the status of Robot Power. This switch is automatically enabled whenever Robot Power is turned on. This switch can be used to turn Robot Power on or off-enabling the switch turns on Robot Power and disabling the switch turns off Robot Power. Default is disabled.
ROBOT	This is an array of switches that control whether or not the system should access robots normally controlled by the system.Default is disabled.
UPPER	Determines whether comparisons of string values will consider lowercase letters the same as uppercase letters. When this switch is enabled, all lowercase letters are considered as though they are uppercase. Default is enabled.

Motion Control Operations

The following topics are described in this chapter:

Introduction

A primary focus of the eV+ language is to drive motion devices. This chapter discusses the language elements that generate controller output to move a motion device from one location to another. Before we introduce the eV+ motion instructions, we should examine the eV+ location variables and see how they relate to the space in which the motion device operates.

Location Variables

Locations can be specified in two ways in eV+: transformations and precision points.

A transformation is a set of six components that uniquely identifies a location in Cartesian space and the orientation of the motion device end-of-arm tooling at that location. A transformation can also represent the location of an arbitrary local reference frame.

A precision point includes an element for each joint in the motion device. Rotational joint values are measured in degrees; translational joint values are measured in millimeters. These values are absolute with respect to the motion device's home sensors and cannot be made relative to other locations or coordinate frames.

Coordinate Systems

The following figure shows the world coordinate system for an Omron Adept SCARA robot and an Omron Adept Cartesian robot. Ultimately, all transformations are based on a world coordinate system. The eV+ language contains several instructions for creating local reference frames, building relative transformations, and changing the origin of the base (world) coordinate frame. Therefore, an individual transformation may be relative to another transformation, a local reference frame, or an altered base reference frame.

Different robots and motion devices designates different locations as the origin of the world coordinate system. See the user's guide for Omron Adept robots to determine the origin and orientation of the world coordinate frame.
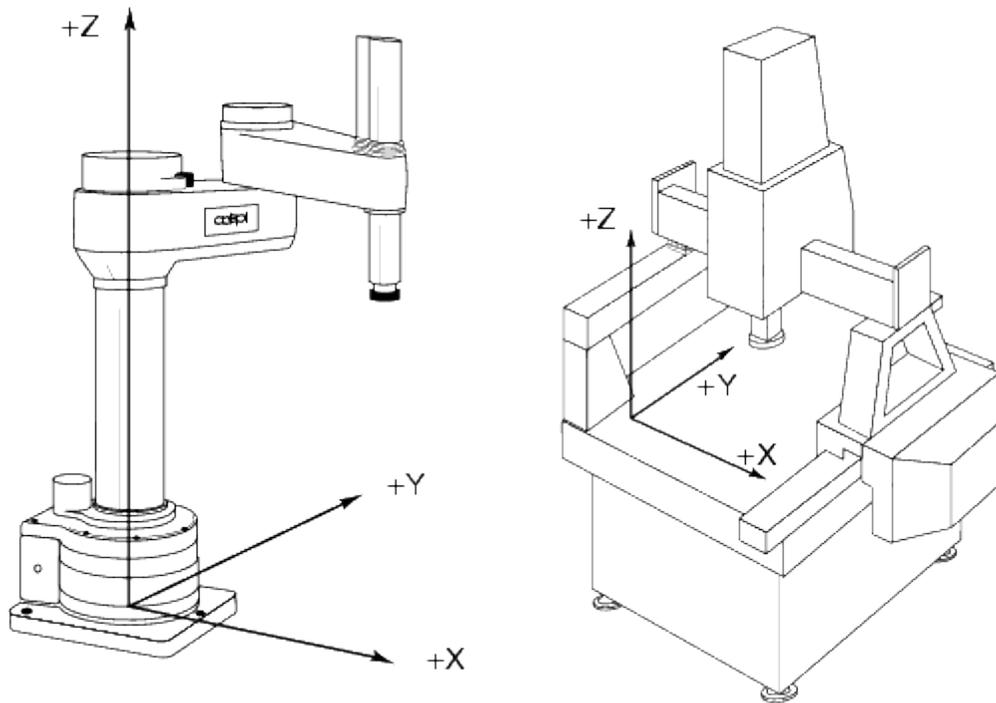
Omron Adept Robot Cartesian Space

Transformations

The first three components of a transformation variable are the values for the points on the X, Y, and Z axes. In an Omron Adept SCARA robot, the origin of this Cartesian space is the base of the robot. The Z axis points straight up through the middle of the robot column. The X axis points straight out, and the Y axis runs left to right as you face the robot. The first robot in the figure Omron Adept Robot Cartesian Space shows the orientation of the Cartesian space for an Omron Adept SCARA robot. The location of the world coordinate system for other robots and motion devices depends on the kinematic model of the motion device. For example, the second robot in the figure Omron Adept Robot Cartesian Space shows the world coordinate frame for a robot built on the Cartesian coordinate model. See the kinematic device module documents for your particular motion device.

When a transformation is defined, a local reference frame is created at the X, Y, Z location with all three local frame axes parallel to the world coordinate frame. The figure XYZ Elements of a Transformation shows the first part of a transformation. This transformation has the value X = 30, Y = 100, Z = 125, yaw = 0, pitch = 0, and roll = 0.
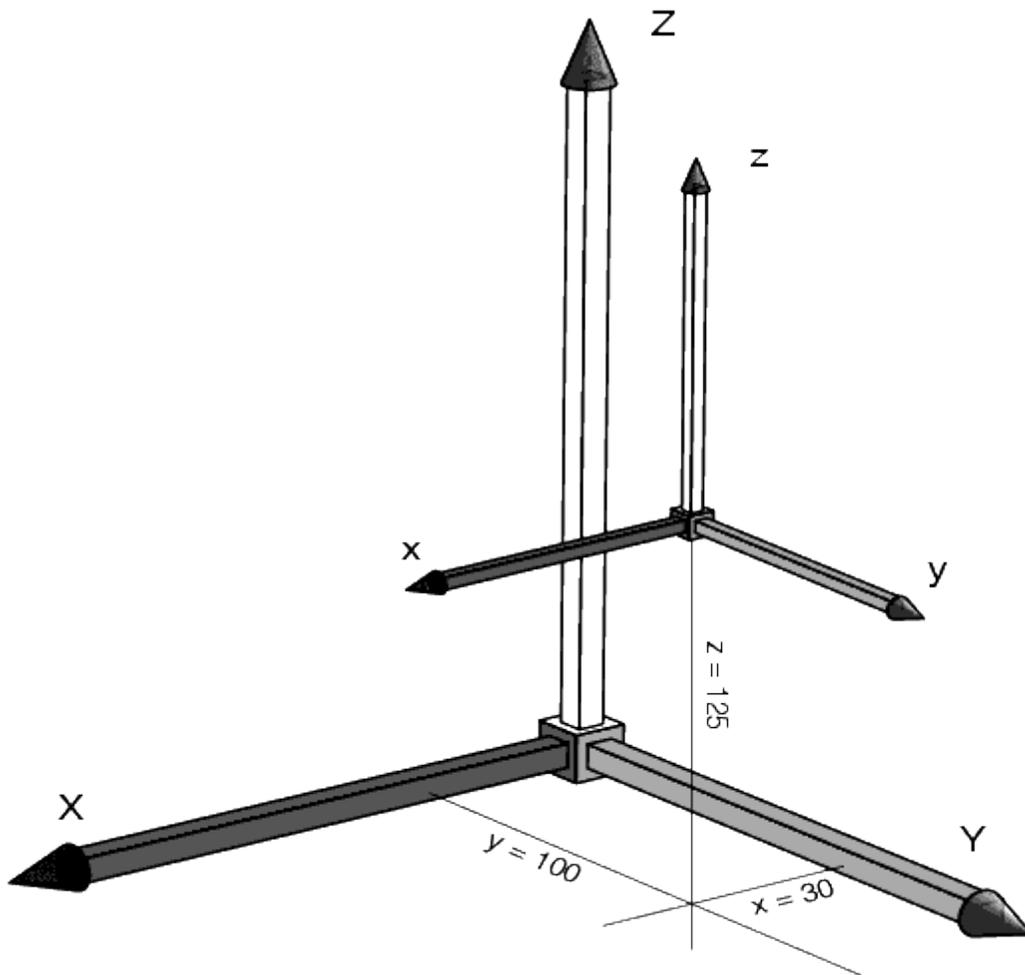
XYZ Elements of a Transformation

The second three components of a transformation variable specify the orientation of the end-of-arm tooling. These three components are yaw, pitch, and roll. These elements are figured as ZYZ' Euler values. The following figures demonstrate how these values are interpreted.

Yaw

Yaw is a rotation about the local reference frame Z axis. This rotation is not about the primary reference frame Z axis, but is centered at the origin of the local frame of reference. The figure Yaw shows the yaw axis with a rotation of 30 degrees. Note that it is parallel to the primary reference frame Z axis but may be centered at any point in that space. In this example, the yaw value is 30 degrees, resulting in a transformation with the value (X = 30, Y = 100, Z = 125, yaw = 30, pitch = 0, and roll = 0).

When you are using a robot, the local frame of reference defined by the XYZ components is located at the end of the robot tool flange. (This local reference frame is referred to as the tool coordinate system.) In the figure Yaw, the large Cartesian space represents a world coordinate system. The small Cartesian space represents a local tool coordinate system (which is centered at the motion device tooling flange).

Yaw
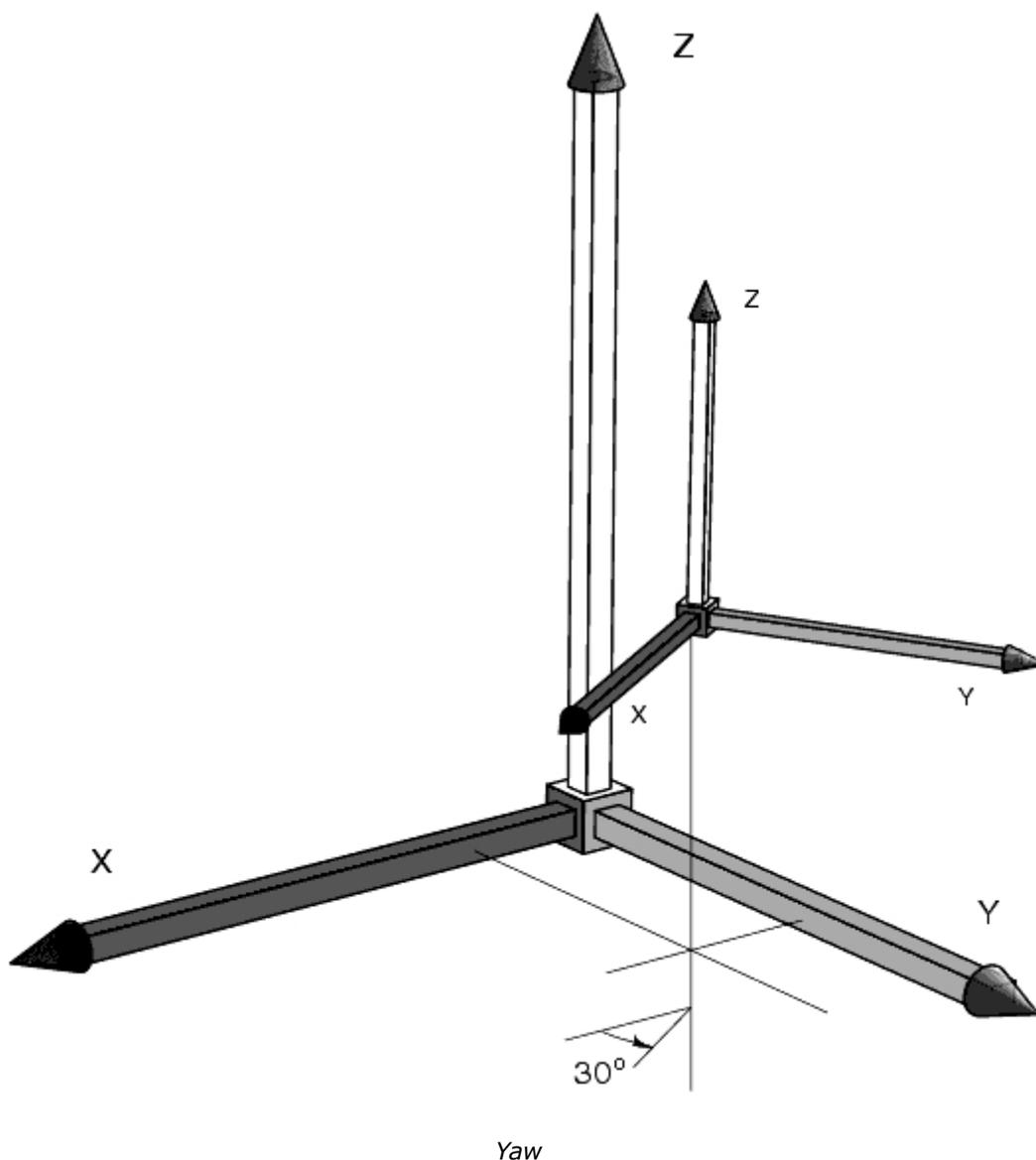
Pitch

Pitch is defined as a rotation about the local reference frame Y axis, after yaw has been applied. The figure Pitch shows the local reference frame with a yaw of 30 degrees and a pitch of 40 degrees.

For example, deflection of a wrist joint is reflected in the pitch component. The movement of a fifth axis on a SCARA robot is reflected in the pitch component. In this example, the motion device end-of-arm tooling has a pitch of 40 degrees, resulting in a transformation with the value (X = 30, Y = 100, Z = 125, yaw = 30, pitch = 40, and roll = 0). This location can be reached only by a mechanism with a fifth axis. Pitch is represented as ±180°, not as 360° of rotation. Thus, a positive rotation of 190° is shown as -170 degrees.
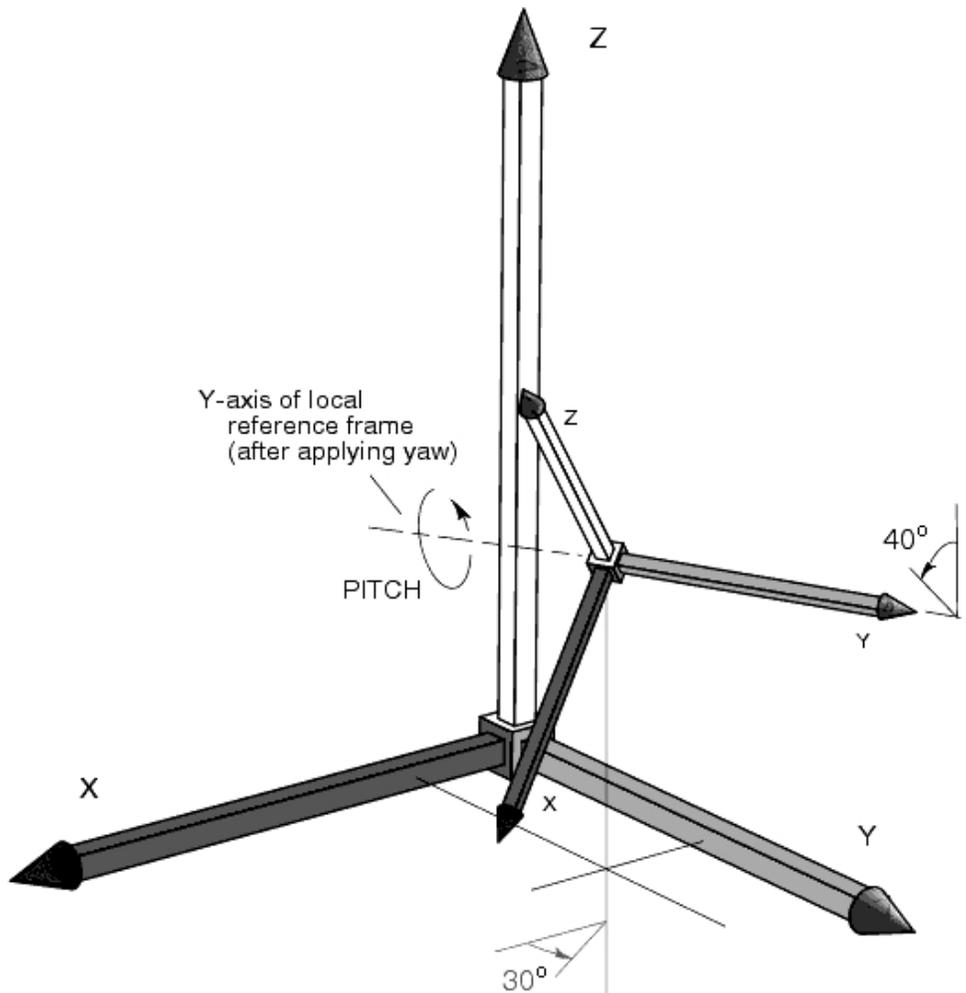
Pitch

Roll

Roll is defined as a rotation about the Z axis of the local reference frame after yaw and pitch have been applied. The figure Roll shows a local reference frame in the primary robot Cartesian space and the direction roll would take within that space. In this example the transformation has a value of X = 30, Y = 100, Z = 125, yaw = 30, pitch = 40, and roll = 20. This location can be reached only by a mechanism with fifth and sixth axes.
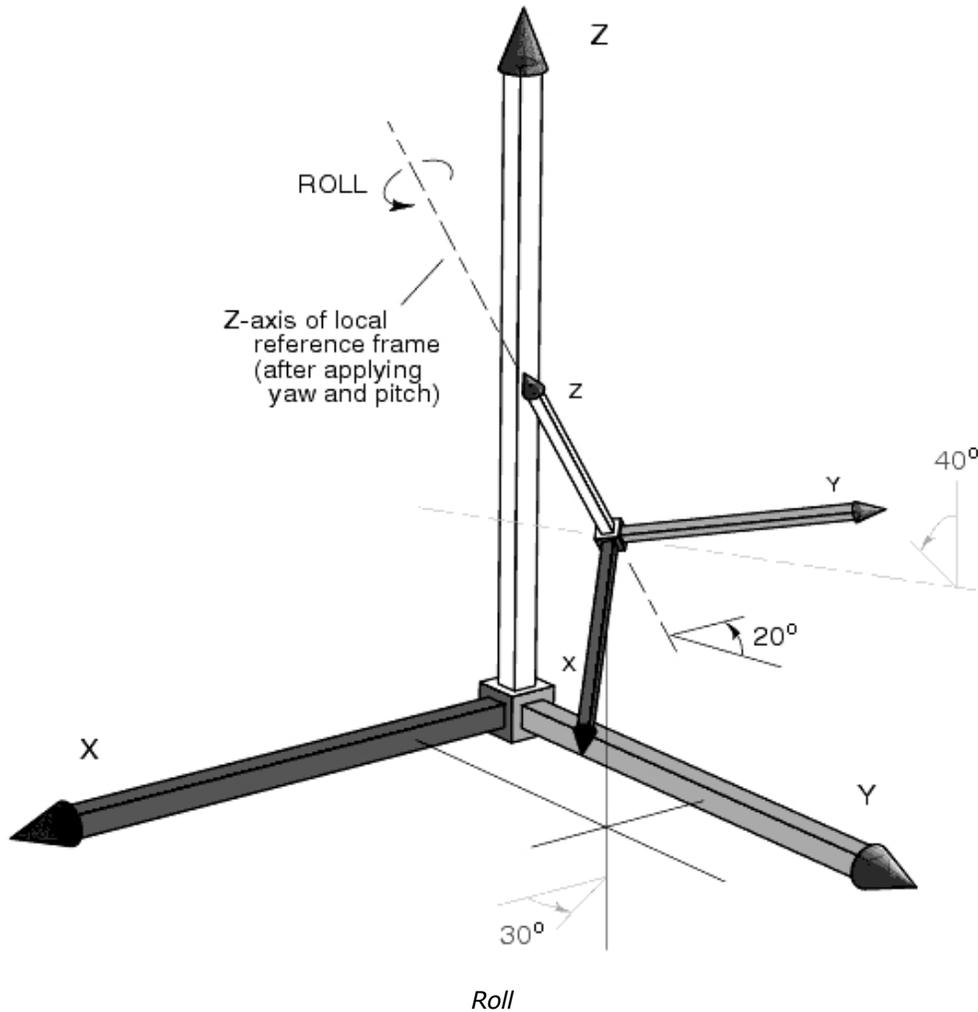
Roll

Special Situations

When the Z axes of the local and primary reference frames are parallel, roll and yaw produce the same motion in the same plane, although the two motions may be in different directions. This is always the case with a four-axis SCARA robot. The system automatically reflects rotation of the quill in the roll component of a transformation variable, and the yaw component is forced to 0 degrees. In a SCARA robot equipped with a fifth axis, rotation of the quill is reflected in the yaw component and motion of a rotating end-effector (sixth axis) is reflected in the roll component.

Notice in the figure XYZ Elements of a Transformation that the local reference frame points straight up. This corresponds to a situation where the end of arm tooling points straight back along the third axis. In a mechanism not equipped with a 360 degree wrist, this is an impossible position. For a four-axis SCARA, this component must point straight down (pitch = 180 degrees). For a mechanism with a fifth axis, this component must be within the range of motion of the fifth axis.

NOTE: When thinking about a transformation, remember that the rules of ZYZ' Euler angles require that the orientation components be applied in order after the local reference frame has been defined. After calculating the Cartesian components and placing a local reference frame with x, y, and z axes parallel to the primary reference frame X, Y, and Z axes, the orientation components are applied in a strict order—yaw is applied first, then pitch, and, finally, roll.

Creating and Altering Location Variables

Creating Location Variables

The most straightforward method of creating a location variable is to place the robot or motion device at a location and enter the monitor command:

```
HERE loc_name
```

Transformations vs. Precision Points

A location can be specified using either the six components described in the previous section, or by specifying the state the robot joints would be in when a location is reached. The former method results in a transformation variable. Transformations are the most flexible and efficient location variables.

Precision points record the joint values of each joint in the motion device. Precision points may be more accurate, and they are the only way of extracting joint information that will allow you to move an individual joint. Precision points are identified by a leading pound sign (#). The command:

```
HERE #pick
```

will create the precision point #pick equal to the current robot joint values.

Modifying Location Variables

A location variable can be duplicated using the SET program instruction. The program instruction:

```
SET loc_name = loc_value
```

results in the variable loc_name being given the value of loc_value.

The following functions return transformation values:

TRANS Create a location by specifying individual components of a transformation. A value can be specified for each component.
SHIFT Alter the Cartesian components of an existing transformation.

The SET operation can be used in conjunction with the transformation functions SHIFT and TRANS to create location variables based on specific modifications of existing variables.

```
SET loc_name = SHIFT(loc_value BY 5, 5, 5)
```

will create the location variable loc_name. The location of loc_name are shifted 5 mm in the positive X, Y, and Z directions from loc_value.

Relative Transformations

Relative transformations allow you to make one location relative to another and to build local reference frames to which that transformations can be relative. For example, you may be building an assembly whose location in the workcell changes periodically. If all the locations on the assembly are taught relative to the world coordinate frame, each time the assembly is located differently in the workcell, all the locations must be retaught. If, however, you create a frame based on identifiable features of the assembly, you will have to reteach only the points that define the frame.

Examples of Modifying Location Variables

The figure Relative Transformation shows how relative transformations work. The magnitude and direction elements (x, y, z), but not the orientation elements (y, p, r), of an Omron Adept transformation can be represented as a 3-D vector, as shown by the dashed lines and arrows in the figure Relative Transformation. The following code generates the locations shown in that figure.

```
; Define a simple transformation
    SET loc_a = TRANS(300,50,350,0,180,0)
; Move to the location
    MOVE loc_a
    BREAK
; Move to a location offset -50mm in X, 20mm in Y,
; and 30mm in Z relative to "loc_a"
    MOVE loc_a:TRANS(-50, 20, 30)
    BREAK
; Define "loc_b" to be the current location relative
; to "loc_a"
    HERE loc_a:loc_b          ;loc_b = -50, 20, 30, 0, 0, 0
    BREAK
; Define "loc_c" as the vector sum of "loc_a" and "loc_b"
    SET loc_c = loc_a:loc_b   ;loc_c = 350, 70, 320, 0, 180, 0
```

Once this code has run, loc_b exists as a transformation that is completely independent of loc_a. The following instruction moves the robot another -50 mm in the x, 20 mm in the y, and 30 mm in the z direction (relative to loc_c):

```
    MOVE loc_c:loc_b
```

Multiple relative transformations can be chained together. If we define loc_d to have the value 0, 50, 0, 0, 0, 0:

```
    SET loc_d = TRANS(0,50)
```

and then issue the following MOVE instruction:

```
    MOVE loc_a:loc_b:loc_d
```

the robot moves to a position x = -50 mm, y =70 mm, and z = 30 mm relative to loc_a.

In the figure Relative Transformation, the transformation loc_b defines the transformation needed to get from the local reference frame defined by loc_a to the local reference frame defined by loc_c.

The transformation needed to go in the opposite direction (from loc_c to loc_a) can be calculated by:

```
INVERSE(loc_b)
```

Thus, the instruction:

```
MOVE loc_c:INVERSE(loc_b)
```

effectively moves the robot back to loc_a.
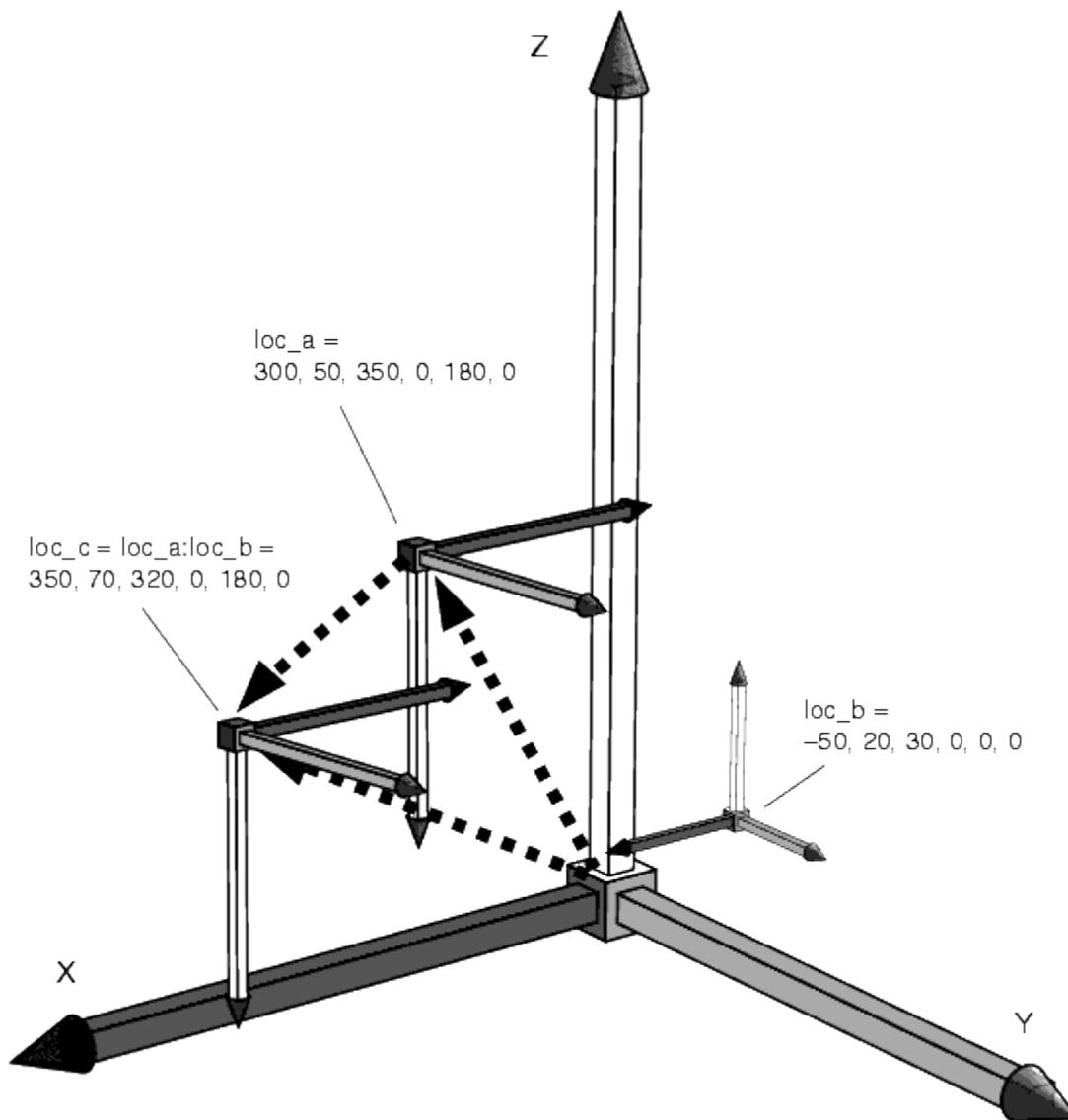
Relative Transformation

This figure shows the first three locations from the previous code examples.

Defining a Reference Frame

In the example shown in the figure Relative Locations, a pallet is brought into the workcell on a conveyor. The program that follows teaches three locations that define the pallet reference frame (pallet.frame) and then removes the parts from the pallet. The program that follows

runs regardless of where the pallet is placed in the workcell as long as it is within the robot's working envelope.
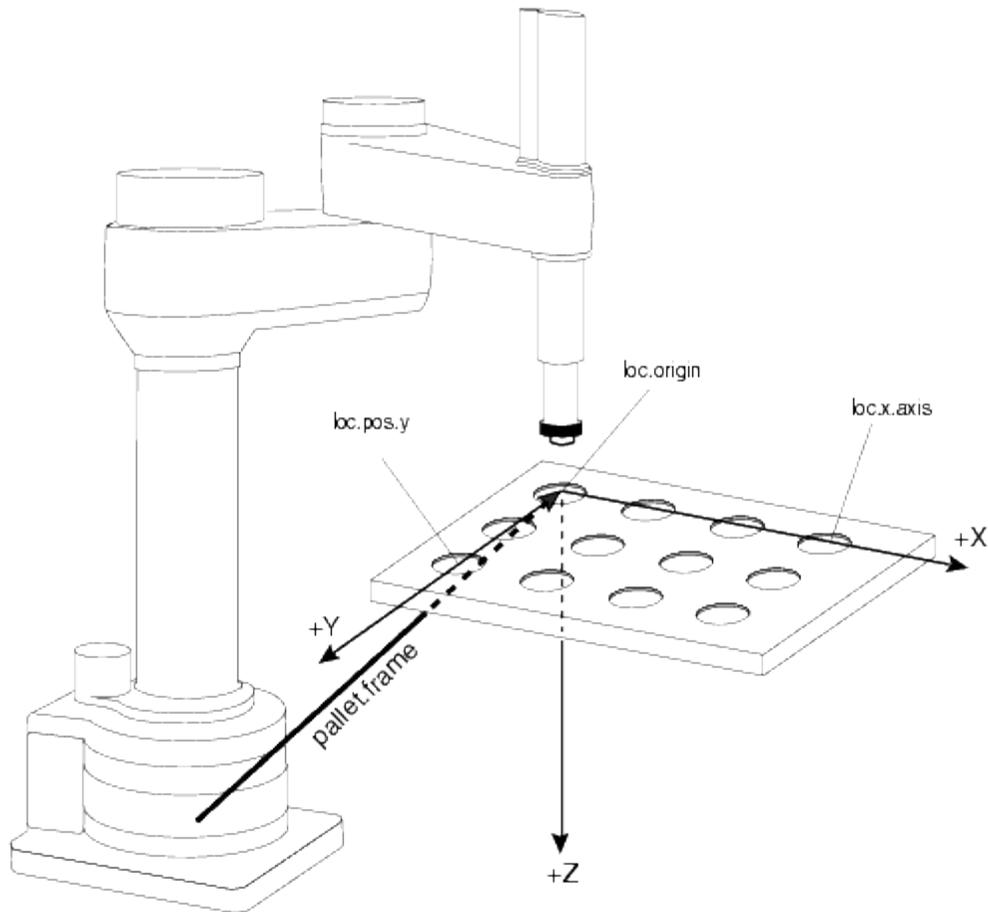
Relative Locations

```
; Get the locations to define the pallet

DETACH ()                                    ;Release robot for use by the MCP
PROMPT "Place robot at pallet origin. ", $ans
HERE loc.origin                              ;Record the frame origin

PROMPT "Place robot at point on the pallet x-axis. ", $ans
HERE loc.x.axis                              ;Record point on x-axis

PROMPT "Place robot at point in positive y direction. ", $ans
HERE loc.pos.y                               ;Record positive y direction

ATTACH ()                                    ;Reattach the robot
```

```
; Create the local reference frame "pallet.frame"

SET pallet.frame = FRAME(loc.origin, loc.x.axis,loc.pos.y, loc.origin)

cell.space = 50                              ;Spacing of cells on pallet

; Remove the palletized items

FOR i = 0 TO 3
     FOR J = 0 TO 2
          APPRO pallet.frame:TRANS(i*cell.space, j*cell.space), 25
          MOVE pallet.frame:TRANS(i*cell.space, j*cell.space)
          BREAK                              ;Settle robot
          CLOSEI                             ;Grab the part
          DEPART 25                          ;MOVE to the drop off location
     END
END
```

In the above example, the code that teaches the pallet frame must run only when the pallet location changes.

If you are building an assembly that does not have regularly spaced locations like the above example, the following code teaches individual locations relative to the frame:

```
; Get the locations to define the pallet frame

DETACH ()                                    ;Release robot for use by the MCP
PROMPT "Place robot at assembly origin. ", $ans
HERE loc.origin                              ;Record the frame origin

PROMPT "Place robot at point on the assm. x-axis. ", $ans
HERE loc.x.axis                              ;Record point on x-axis

PROMPT "Place robot at point in positive y direction. ", $ans
HERE loc.pos.y          ;Record positive y direction

; Create the local reference frame "assm.frame"

SET assm.frame = FRAME(loc.origin, loc.x.axis, loc.pos.y, loc.origin)

; Teach the locations on the assembly

PROMPT "Place the robot in the first location. ", $ans
HERE assm.frame:loc.1                        ;Record the first location

PROMPT "Place the robot in the second location. ", $ans
HERE assm.frame:loc.2                        ;Record the second location

; etc.

; Move to the locations on the assembly

ATTACH ()                                    ;Reattach the robot
```

```
APPRO assm.frame:loc.1, 25
MOVE assm.frame:loc.1
;Activate gripper
DEPART 25

APPRO assm.frame:loc.1, 25
MOVE assm.frame:loc.2
;Activate gripper
DEPART 25

; etc.
```

In the above example, the frame must be taught each time the assembly moves-the locations on the assembly must be taught only once.

The instruction HERE assm.frame:loc.1 tells the system to record the location loc.1 relative to assm.frame rather than relative to the world coordinate frame. If a subassembly is being built relative to loc.1, the instruction:

```
HERE assm.frame:loc.1:sub.loc.1
```

creates a compound transformation where sub.loc.1 is relative to the transformation assm.frame:loc.1.

Miscellaneous Location Operations

The instruction:

```
DECOMPOSE array_name[] = #loc_name
```

places the joint values of #loc_name in the array array_name. DECOMPOSE works with transformations and precision points.

The command:

```
WHERE
```

displays the current robot location.

Motion Control Instructions

eV+ processes robot motion instructions differently from the way you might expect. With eV+, a motion instruction such as MOVE part is interpreted to mean start moving the robot to location 'part'. As soon as the robot starts moving to the specified destination, the eV+ program continues without waiting for the robot motion to complete. The instruction sequence:

```
MOVE part.1
SIGNAL 1
MOVE part.2
SIGNAL 2
```

causes external output signal #1 to be turned on immediately after the robot begins moving to part.1, rather than waiting for it to arrive at the location. When the second MOVE instruction is encountered, eV+ waits until the motion to part.1 is completed. External output signal #2 is turned on just after the motion to part.2 begins. This is known as forward processing. See Breaking Continuous-Path Operation for details on how to defeat forward processing.

This parallel operation of program execution and robot motion makes possible the procedural motions described later in this chapter.

Basic Motion Operations

Joint-Interpolated Motion vs. Straight-Line Motion

The path a motion device takes when moving from one location to another can be either a joint-interpolated motion or a straight-line motion. Joint-interpolated motions move each joint at a constant velocity (except during the acceleration/deceleration phases-see Robot Speed). Typically, the robot tool tip moves in a series of arcs that represents the least processing-intensive path the trajectory generator can formulate. Straight-line motions ensure that the robot tool tip traces a straight line, useful for cutting a straight line or laying a bead of sealant. The instruction:

```
MOVE pick
```

causes the robot to move to the location pick using joint-interpolated motion. The instruction:

```
MOVES pick
```

causes the robot to move the pick using a straight-line motion.

Safe Approaches and Departures

In many cases you will want to approach a location from a distance offset along the tool Z axis or depart from a location along the tool Z axis before moving to the next location. For example, if you were inserting components into a crowded circuit board, you would want the

robot arm to approach a location from directly above the board so nearby parts are not disturbed. Assuming you were using a four-axis Omron Adept robot, the instructions:

```
APPRO place, 50
MOVE place
DEPART 50
```

causes joint-interpolated motion to a point 50 mm above place, movement down to place, and movement straight up to 50 mm above place.

If the instructions APPROS, DEPARTS, and MOVES had been used, the motions would have been straight line instead of joint interpolated.

NOTE: Approaches and departs are based on the tool coordinate system, not the world coordinate system. Thus, if the location specifies a pitch of 135 degrees, the robot will approach at a 45 degree angle relative to the world coordinate system. For a description of the tool coordinate system, see Yaw on page 104.

Moving an Individual Joint

You can move an individual joint of a robot using the instruction DRIVE. The instructions:

```
DRIVE 2,50.0, 100
DRIVE 3,25, 100
```

moves joint 2 through 50 degrees of motion and then move joint 3 a distance of 25 mm at SPEED 100%.

End-Effector Operation Instructions

The instructions described in this section depend on the use of two digital signals. They are used to open, close, or relax a gripper. The utility program SPEC specifies which signals control the end effector. See the *Instructions for Adept Utility Programs*.

The instruction OPEN opens the gripper during the ensuing motion instruction. The instruction OPENI opens the gripper before any additional motion instructions are executed. CLOSE and CLOSEI are the complementary instructions.

When an OPEN(I) or CLOSE(I) instruction is issued, one solenoid is activated and the other is released. To completely relax both solenoids, use the instruction RELAX or RELAXI.

Use the system parameter HAND.TIME to set the duration of the motion delay that occurs during an OPENI, CLOSEI, or RELAXI instruction.

Use the function HAND to return the current state of the gripper.

Continuous-Path Trajectories

When a single motion instruction is processed, such as the instruction:

```
MOVE pick
```

the robot begins moving toward the location by accelerating smoothly to the commanded speed. Sometime later, when the robot is close to the destination location pick, the robot decelerates smoothly to a stop at location pick. This motion is referred to as a single motion segment, since it is produced by a single motion instruction.

When a sequence of motion instructions is executed, such as:

```
MOVE loc.1
MOVE loc.2
```

 the robot begins moving toward loc.1 by accelerating smoothly to the commanded speed[1] just as before. However, the robot does not decelerate to a stop when it gets close to loc.1. Instead, it smoothly changes its direction and begins moving toward loc.2. Finally, when the robot is close to loc.2, it decelerates smoothly to a stop at loc.2. This motion consists of two motion segments since it is generated by two motion instructions.

Making smooth transitions between motion segments without stopping the robot motion is called continuous-path operation. That is the normal method eV+ uses to perform robot motions. If desired, continuous-path operation can be disabled with the CP switch. When the CP switch is disabled, the robot decelerates and stops at the end of each motion segment before beginning to move to the next location.

NOTE: Disabling continuous-path operation does not affect forward processing (the parallel operation of robot motion and program execution).

Continuous-path transitions can occur between any combination of straight-line and joint-interpolated motions. For example, a continuous motion could consist of a straight-line motion (for example, DEPARTS) followed by a joint-interpolated motion (for example, APPRO) and a final straight-line motion (for example, MOVES). Any number of motion segments can be combined this way.

Breaking Continuous-Path Operation

Certain eV+ program instructions cause program execution to be suspended until the current robot motion reaches its destination location and comes to a stop. This is called breaking continuous path. Such instructions are useful when the robot must be stopped while some operation is performed (for example, closing the hand). Consider the instruction sequence:

```
MOVE loc.1
BREAK
SIGNAL 1
```

The MOVE instruction starts the robot moving to loc.1. Program execution then continues and the BREAK instruction is processed. BREAK causes the eV+ program to wait until the motion to loc.1 completes. The external signal is not turned on until the robot stops. (Recall that without the BREAK instruction the signal would be turned on immediately after the motion to loc.1 **starts**.)

The following instructions always cause eV+ to suspend program execution until the robot stops (see the *eV+ Language Reference Guide* for detailed information on these instructions):

> BREAK CLOSEI CPOFF DETACH (0)
>
> HALT OPENI PAUSE RELAXI TOOL

Also, the robot decelerates to a stop when the BRAKE (not to be confused with BREAK) instruction is executed (by any program task), and when the reaction associated with a REACTI instruction is triggered. These events could happen at any point within a motion segment. (Note that these events can be initiated from a different program task.)

The robot also decelerates and comes to a stop if no new motion instruction is encountered before the current motion completes. This situation can occur for a variety of reasons:

- A WAIT or WAIT.EVENT instruction is executed and the wait condition is not satisfied before the robot motion completes.

- A PROMPT instruction is executed and no response is entered before the robot motion completes.

- The eV+ program instructions between motion instructions take longer to execute than the robot takes to perform its motion.

Procedural Motion

The ability to move in straight lines and joint-interpolated arcs is built into the basic operation of eV+. The robot tool can also move along a path that is prerecorded, or described by a mathematical formula. Such motions are performed by programming the robot trajectory as the robot is moving. Such a program is said to perform a procedural motion.

A procedural motion is a program loop that computes many short motions and issues the appropriate motion requests. The parallel execution of robot motions and non-motion instructions allows each successive motion to be defined without stopping the robot. The continuous-path feature of eV+ automatically smoothes the transitions between the computed motion segments.

Procedural Motion Examples

Two simple examples of procedural motions are described below. In the first example, the robot tool is moved along a trajectory described by locations stored in the array path. (The LAST function is used to determine the size of the array.)

```
SPEED 0.75 IPS ALWAYS
FOR index = 0 TO LAST(path[])
    MOVES path[index]
END
```

The robot tool moves at the constant speed of 0.75 inch per second through each location defined in the array path[].

In the next example, the robot tool is to be moved along a circular arc. However, the path is not prerecorded-it is described mathematically, based on the radius and center of the arc to be followed.

The program segment below assumes that a real variable radius has already been assigned the radius of the desired arc, and x.center and y.center have been assigned the respective coordinates of the center of curvature. The variables start and last are assumed to have been defined to describe the portion of the circle to be traced. Finally, the variable angle.step is assumed to have been defined to specify the (angular) increment to be traversed in each incremental motion. (Because the DURATION instruction is used, the program moves the robot tool angle.step degrees around the arc every 0.5 second.)

When this program segment is executed, the X and Y coordinates of points on the arc are repeatedly computed. They are then used to create a transformation that defines the destination for the next robot motion segment.

```
DURATION 0.5 ALWAYS
FOR angle = start TO last STEP angle.step
    x = radius*COS(angle)+x.center
    y = radius*SIN(angle)+y.center
    MOVE TRANS(x, y, 0, 0, 180, 0)
END
```

Timing Considerations

Because of the computation time required by eV+ to perform the transitions between motion segments, there is a limit on how closely spaced commanded locations can be. When locations are too close together, there is not enough time for eV+ to compute and perform the transition from one motion to the next, and there will be a break in the continuous-path motion. This means that the robot stops momentarily at intermediate locations.

The minimum spacing that can be used between locations before this effect occurs is determined by the time required to complete the motion from one location to the next. Straight-line motions can be used if the motion segments take more than about 32 milliseconds each. Joint-interpolated motions can be used with motion segments as short as about 16 milliseconds each.

NOTE: The standard trajectory generation frequency is 62.5 Hz. With an optional software license, trajectory frequencies of 125 Hz, 250 Hz, and 500 Hz are possible.

The minimum motion times for joint and straight-line motions must be greater than or equal to the configured trajectory cycle time. As a convenience, if they are set to be less than the configured trajectory cycle time (for example 0), the trajectory cycle time is used as the minimum motion time.

Robot Speed

A robot move has three phases: an acceleration phase where the robot accelerates to the maximum speed specified for the move, a velocity phase where the robot moves at a rate not exceeding the specified maximum speed, and a deceleration phase where the robot decelerates to a stop (or transitions to the next motion).

Robot speed can mean two things: how fast the robot moves between the acceleration and deceleration phases of a motion (referred to in this manual as robot speed), or how fast the robot gets from one place to another (referred to in this manual as robot performance).

The robot speed between the acceleration and deceleration phases is specified as either a percentage of normal speed or an absolute rate of travel of the robot tool tip. Speed set as a percentage of normal speed is the default. The speed of a robot move based on normal speed is determined by the following factors:

- The program speed (set with the SPEED program instruction). This speed is set to 100 when program execution begins.

- The monitor speed (set with the SPEED monitor command or a SPEED program instruction that specifies MONITOR). This speed is normally set to 50 at system startup(start-up SPEED can be set with the ACE Controller Config Tools). (The effects of the two SPEED operations are slightly different. See the SPEED program instruction for further details.)

    Robot speed is the product of these two speeds. With monitor speed and program speed set to 100, the robot moves at its normal speed. With monitor speed set to 50 and program speed set to 50, the robot moves at 25% of its normal speed.

To move the robot tool tip at an absolute rate of speed, a speed rate in inches per second or millimeters per second is specified in the SPEED program instruction. The instruction:

```
SPEED 25 MMPS ALWAYS
```

specifies an absolute tool tip speed of 25 millimeters per second for all robot motions until the next SPEED instruction. In order for the tool tip to actually move at the specified speed:

- The monitor speed must be 100.

- The locations must be far enough apart so that the robot can accelerate to the desired speed and decelerate to a stop at the end of the motion.

Robot performance is a function of the SPEED settings and the following factors:

- The robot acceleration profile and ACCEL settings. The default acceleration profile is based on a normal maximum rate of acceleration and deceleration. The ACCEL command can scale down these maximum rates so that the robot acceleration and/or deceleration takes more time.

    You can also define optional acceleration profiles that alter the maximum rate of change for acceleration and deceleration

- The location tolerance settings (COARSE/FINE, NULL/NONULL) for the move. The more accurately a robot must get to the actual location, the more time the move will take.

- Any DURATION setting. DURATION forces a robot move to take a minimum time to complete regardless of the SPEED settings.

- The maximum allowable velocity. For Omron Adept robots, maximum velocity is factory set.

- The inertial loading of the robot and the tuning of the robot.

- Straight-line vs. joint-interpolated motions-for complex geometries, straight-line and joint-interpolated paths produce different dynamic responses and, therefore, different motion times.

Robot performance for a given application can be greatly enhanced or severely degraded by these settings. For example:

- A heavily loaded robot may actually show better performance with slower SPEED and ACCEL settings, which lessens overshoot at the end of a move and allows the robot to settle more quickly.

- Applications such as picking up bags of product with a vacuum gripper do not require high accuracy and can generally run faster with a COARSE tolerance.

Motion Modifiers

The following instructions modify the characteristics of individual motions. These instructions are summarized in Motion Control Operations.

> **NOTE:**The instructions listed below with an asterisk (*) can take ALWAYS as an argument.

- ABOVE/BELOW
- ACCEL
- BREAK
- COARSE/FINE*
- CPON/CPOFF
- DURATION*
- FLIP/NOFLIP
- LEFTY/RIGHTY
- NOOVERLAP/OVERLAP*
- NULL/NONULL*BRAKE
- SINGLE/MULTIPLE*
- SPEED*

Customizing the Calibration Routine

The following information is required only if you need to customize the calibration sequence. Most AdeptMotion users do not need to do this.

When a CALIBRATE command or instruction is processed, the eV+ system loads the file CAL_UTIL.V2 (see the dictionary page for the CALIBRATE command for details) and executes a program contained in that file. The main calibration program then examines the SPEC data for the robot to determine the name of the disk file that contains the specific calibration program for the current robot, and the name of that program.

The standard routine used for AdeptMotion devices is stored on the system disk in \CALIB\STANDARD.CAL (and the routine is named **.standard.cal**). That file is protected and thus cannot be viewed. However, a read-only copy of the file is provided, in \CALIB\STANDARD.V2, as a basis for developing a custom calibration routine that can then be substituted for the standard file. (The name of the robot-specific calibration file and program can be changed using the ACE Controller Config Tools..)

---

[1]See the SPEED monitor command and SPEED program instructions.

Tool Transformations

A tool transformation is a special transformation that is used to account for robot grippers (or parts held in grippers) that are offset from the center of the robot tool flange. If a location is taught using a part secured by an offset gripper, the actual location recorded is not the part location, but the center of the tool flange to which the offset gripper is attached, as shown in the following figure. If the same location is taught with a tool transformation in place, the location recorded is the center of the gripper, not the center of the tool flange. This allows you to change grippers and still have the robot reach the correct location. The following figure shows the location of the robot when a location is taught and the actual location that is recorded when no tool transformation is in effect. If the proper tool transformation is in effect when the location is taught, the location recorded will be the part location and not the center of the tool flange.
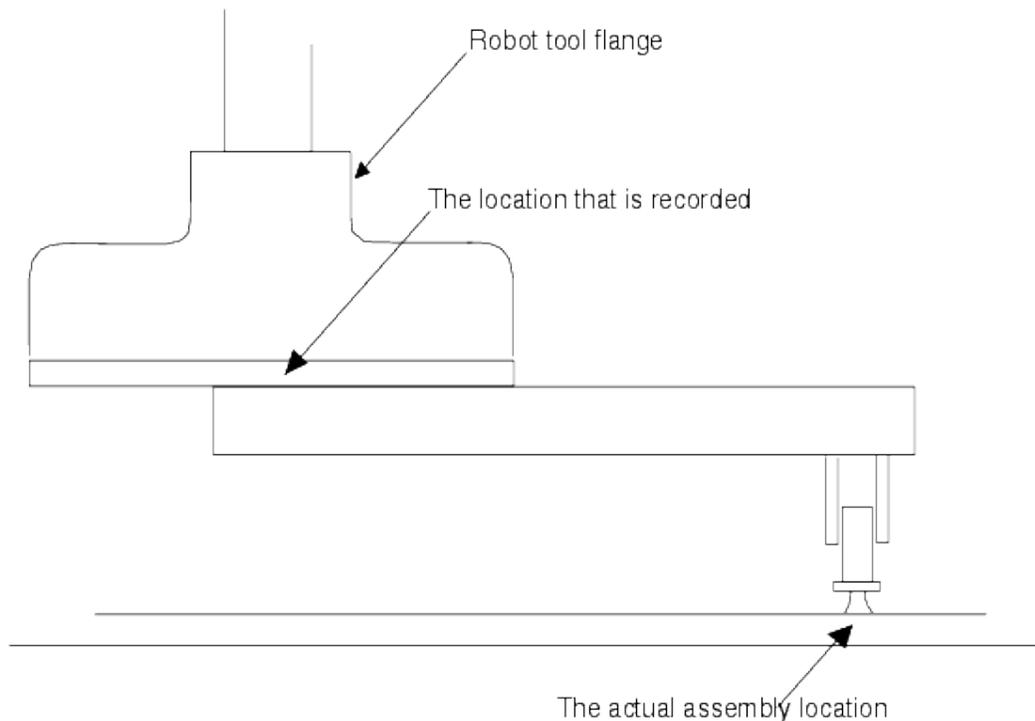
Recording Locations

Tool transformations are most important when:

- Grippers are changed frequently
- The robot is vision guided
- Robot locations are loaded directly from CAD data

Defining a Tool Transformation

A tool transformation can be defined using the Tool Offset wizard, which is available in the Gripper object editor in the ACE software. The Tool Offset wizard will ask you questions about the application, and then calculate the proper tool offset, based on your responses. For more details, see the Gripper Editor topic in the *ACE User's Guide*.

Summary of Motion Keywords

The following table summarizes the keywords associated with motion in eV+. For complete details on any keyword, click on the keyword name in the table, or refer to the keyword documentation available in the *eV+ Language Reference Guide*.

Motion Control Operations

Keyword	Type	Function
ABOVE	PI	Request a change in the robot configuration during the next motion so that the elbow is above the line from the shoulder to the wrist.
ACCEL	PI	Set acceleration and deceleration for robot motions.
ACCEL	RF	Return the current robot acceleration or deceleration setting.
ALIGN	PI	Align the robot tool Z axis with the nearest world axis.
ALTER	PI	Specify the magnitude of the real-time path modification that is to be applied to the robot path during the next trajectory computation.
ALTOFF	PI	Terminate real-time path-modification mode (alter mode).
ALTON	PI	Enable real-time path-modification mode (alter mode), and specify the way in which ALTER coordinate information will be interpreted.
AMOVE	PI	Position an extra robot axis during the next joint-interpolated or straight-line motion.
APPRO	PI	Start joint-interpolated robot motion toward a location defined relative to specified location.
APPROS	PI	Start straight-line robot motion toward a location defined relative to specified location.
BASE	TF	Return the transformation value that represents the translation and rotation set by the last BASE command or instruction.

Keyword	Type	Function
BELOW	PI	Request a change in the robot configuration during the next motion so that the elbow is below the line from the shoulder to the wrist.
BRAKE	PI	Abort the current robot motion.
BREAK	PI	Suspend program execution until the current motion completes.
CALIBRATE	PI	Initialize the robot positioning system.
CLOSE	PI	Close the robot gripper immediately.
CLOSEI	PI	Close the robot gripper.
COARSE	PI	Enable a low-precision feature of the robot hardware servo (see FINE).
CONFIG	RF	Return a value that provides information about the robot's geometric configuration, or the status of the motion servo-control features.
CP	S	Control the continuous-path feature.
CPOFF	PI	Instruct the eV+ system to stop the robot at the completion of the next motion instruction (for all subsequent motion instructions) and null position errors.
CPON	PI	Instruct the eV+ system to execute the next motion instruction (or all subsequent motion instructions) as part of a continuous path.
DECOMPOSE	PI	Extract the (real) values of individual components of a location value.
DELAY	PI	Cause robot motion to stop for the specified period of time.
DEPART	PI	Start a joint-interpolated robot motion away from the current location.

Keyword	Type	Function
DEPARTS	PI	Start a straight-line robot motion away from the current location.
DEST	TF	Return a transformation value representing the planned destination location for the current robot motion.
DISTANCE	RF	Determine the distance between the points defined by two location values.
DRIVE	PI	Move an individual joint of the robot.
DRY.RUN	S	Control whether or not eV+ communicates with the robot.
DURATION	PI	Set the minimum execution time for subsequent robot motions.
DURATION	RF	Return the current setting of one of the motion DURATION specifications.
DX	RF	Return the X displacement component of a given transformation value.
DY	RF	Return the Y displacement component of a given transformation value.
DZ	RF	Return the Z displacement component of a given transformation value.
FINE	PI	Enable a high-precision feature of the robot hardware servo (see COARSE).
FLIP	PI	Request a change in the robot configuration during the next motion so that the pitch angle of the robot wrist has a negative value (see NOFLIP).
FORCE	S	Control whether or not the (optional) stop-on-force feature of the eV+ system is active.
FRAME	TF	Return a transformation value defined by four positions.

Keyword	Type	Function
HAND	RF	Return the current hand opening.
HAND.TIME	P	Establish the duration of the motion delay that occurs during OPENI, CLOSEI, and RELAXI instructions.
HERE	PI	Set the value of a transformation or precision-point variable equal to the current robot location.
HERE	TF	Return a transformation value that represents the current location of the robot tool point.
IDENTICAL	RF	Determine if two location values are exactly the same.
INRANGE	RF	Return a value that indicates if a location can be reached by the robot, and if not, why not.
INVERSE	TF	Return the transformation value that is the mathematical inverse of the given transformation value.
IPS	CF	Specify the units for a SPEED instruction as inches per second.
LATCH	TF	Return a transformation value representing the location of the robot at the occurrence of the last external trigger.
LATCHED	RF	Return the status of the external trigger and of the information it causes to be latched.
LEFTY	PI	Request a change in the robot configuration during the next motion so that the first two links of a SCARA robot resemble a human's left arm (see RIGHTY).
MMPS	CF	Specify the units for a SPEED instruction as millimeters per second.
MOVE	PI	Initiate a joint-interpolated robot motion to the position and orientation described by the given

Keyword	Type	Function
		location.
MOVES	PI	Initiate a straight-line robot motion to the position and orientation described by the given location.
MOVEF	PI	Initiate a three-segment pick-and-place joint-interpolated robot motion to the specified destination, moving the robot at the fastest allowable speed.
MOVESF	PI	Initiate a three-segment pick-and-place straight-line robot motion to the specified destination, moving the robot at the fastest allowable speed.
MOVET	PI	Initiate a joint-interpolated robot motion to the position and orientation described by the given location and simultaneously operate the hand.
MOVEST	PI	Initiate a straight-line robot motion to the position and orientation described by the given location and simultaneously operate the hand.
MULTIPLE	PI	Allow full rotations of the robot wrist joints (see SINGLE).
NOFLIP	PI	Request a change in the robot configuration during the next motion so that the pitch angle of the robot wrist has a positive value (see FLIP).
NONULL	PI	Instruct the eV+ system not to wait for position errors to be nulled at the end of continuous-path motions (see NULL).
NOOVERLAP	PI	Disable the NOOVERLAP limit-error checking (see OVERLAP.)
NORMAL	TF	Correct a transformation for any mathematical round-off errors.
NOT.CALIBRATED	P	Indicate (or assert) the calibration status of the robots connected to the system.

Keyword	Type	Function
NULL	TF	Return a null transformation value-one with all zero components.
NULL	PI	Enable nulling of joint position errors.
OPEN	PI	Open the robot gripper.
OPENI	PI	Open the robot gripper immediately.
OVERLAP	PI	Generate a program error if a subsequent motion is planned that causes a selected multi-turn axis to move more than ±180 degrees to avoid a limit stop (see NOOVERLAP).
#PDEST	PF	Return a precision-point value representing the planned destination location for the current robot motion.
#PHERE	PF	Return a precision-point value representing the current location of the currently selected robot.
#PLATCH	PF	Return a precision-point value representing the location of the robot at the occurrence of the last external trigger.
POWER	S	Control or monitor the status of Robot Power.
#PPOINT	PF	Return a precision-point value composed from the given components.
REACTI	PI	Initiate continuous monitoring of a specified digital signal. Automatically stop the current robot motion if the signal properly transitions and optionally trigger a subroutine call.
READY	PI	Move the robot to the READY location above the workspace, which forces the robot into a standard configuration.
RELAX	PI	Limp the pneumatic hand.

Keyword	Type	Function
RELAXI	PI	Limp the pneumatic hand immediately.
RIGHTY	PI	Request a change in the robot configuration during the next motion so that the first two links of the robot resemble a human's right arm (see LEFTY).
ROBOT	S	Enable or disable one robot or all robots.
RX	TF	Return a transformation describing a rotation about the x axis.
RY	TF	Return a transformation describing a rotation about the y axis.
RZ	TF	Return a transformation describing a rotation about the z axis.
SCALE	TF	Return a transformation value equal to the transformation parameter with the position scaled by the scale factor.
SCALE.ACCEL	S	Enable or disable the scaling of acceleration and deceleration as a function of program speed.
SCALE.ACCEL.ROT	S	Specify whether or not the SCALE.ACCEL switch takes into account the Cartesian rotational speed during straight-line motions.
SELECT	PI	Select the unit of the named device for access by the current task.
SELECT	RF	Return the number of the currently selected unit of the named device type.
SET	PI	Set the value of the location variable on the left equal to the location value on the right of the equal sign.
SHIFT	TF	Return a transformation value resulting from shifting the position of the transformation parameter by the given shift amounts.

Keyword	Type	Function
SINGLE	PI	Limit rotations of the robot wrist joint to the range -180 degrees to +180 degrees (see MULTIPLE).
SOLVE.ANGLES	PI	Compute the robot joint positions (for the current robot) that are equivalent to a specified transformation.
SOLVE.FLAGS	RF	Return bit flags representing the robot configuration specified by an array of joint positions.
SOLVE.TRANS	PI	Compute the transformation equivalent to a given set of joint positions for the current robot.
SPEED	PI	Set the nominal speed for subsequent robot motions.
SPEED	RF	Return one of the system motion speed factors.
STATE	RF	Return a value that provides information about the robot system state.
TOOL	PI	Set the internal transformation used to represent the location and orientation of the tool tip relative to the tool mounting flange of the robot.
TOOL	TF	Return the value of the transformation specified in the last TOOL command or instruction.
TRANS	TF	Return a transformation value computed from the given X, Y, Z position displacements and y, p, r orientation rotations.
PI: Program Instruction, RF: Real-Valued Function, TF: Transformation Function, S: Switch, P: Parameter, PF: Precision-Point Function, CF: Conversion Factor		

Input/Output Operations

The following topics are described in this chapter:

Digital I/O

Omron Adept controllers can communicate in a digital fashion with external devices using the Digital I/O capability. Digital input reads the status of a signal controlled by user-installed equipment. A typical digital input operation is to wait for a microswitch on a workcell conveyor to close, indicating that an assembly is in the proper place. The WAIT instruction and SIG function are used to halt program execution until a digital input channel signal achieves a specified state. The program line:

```
WAIT SIG(1001)
```

halts program execution until a switching device attached to digital input channel 1001 is closed. If signal 1002 is a sensor indicating a part feeder is empty, the code:

```
IF SIG(1002) THEN
        CALL service.feeder()
END
```

checks the sensor state and calls a routine to service the feeder if the sensor is on.

The SIGNAL instruction is used for digital output. In the above example, the conveyor belt may need to be stopped after digital input signal 1001 signals that a part is in place. The instruction:

```
SIGNAL(-33)
```

turns off digital output signal 33, causing the conveyor belt connected to signal 33 to stop. When processing on the part is finished and the part needs to be moved out of the work area, the instruction:

```
SIGNAL(33)
```

turns the conveyor belt back on. The digital I/O channels must be installed before they can be accessed by the SIG function or SIGNAL instruction. The SIG.INS function returns an indication of whether a given signal number is available. The code line:

```
IF SIG.INS(33) THEN
```

can be used to ensure that a digital signal is available before you attempt to access it. The monitor command IO displays the status of all digital I/O channels. For details on installing digital I/O hardware, see the *SmartController EX User's Guide*.

Digital output channels are numbered from 1 to 512. Input channels are in the range 1001 to 1512. Multiple signals can be turned ON or OFF with a single instruction.

```
SIGNAL(33),(-34),(35)
        or
SIGNAL(-33),(34),(-35)
```

High-Speed Interrupts

Normally, the digital I/O system is checked once every eV+ major cycle (every 16 ms). In some cases, the delay or uncertainty resulting may be unacceptable. Digital signals 1001 - 1004 can be configured as high-speed interrupts. When a signal configured as a high-speed interrupt transitions, its state is read at system interrupt level, resulting in a maximum delay of 1 ms. The ACE software Controller Configuration Tools are used to configure high-speed interrupts.

Soft Signals

Soft signals are used primarily as global flags. The soft signals are in the range 2001 - 2512 and can be used with SIG and SIGNAL. A typical use of soft signals is for intertask communication. See "REACT and REACTI" and the REACT_ instructions in the *eV+ Language Reference Guide*.

Digital I/O and Third Party Boards

When eV+ starts, default blocks of system memory are assigned to the digital I/O system. eV+ expects to find the digital I/O image at these locations. If you are using a third party digital I/O board, you must remap these memory locations to correspond to the actual memory location of the digital I/O image on your board. See the description of DEF.DIO in the *eV+ Language Reference Guide* for details.

Digital I/O and DeviceNet

When eV+ starts, default blocks of system memory are assigned to the DeviceNet system. eV+ expects to find the DeviceNet image at these locations. For additional information, see DeviceNet on page 156.

Serial and Disk I/O Basics

The following sections describe the basic procedures that are common to both serial and disk I/O operations. Disk I/O on page 143 covers disk I/O in detail. Serial Line I/O on page 152 covers serial I/O in detail.

Logical Units

All eV+ serial and disk I/O operations reference an integer value called a Logical Unit Number or LUN. The LUN provides a shorthand method of identifying which device or file is being referenced by an I/O operation. See the ATTACH command in the *eV+ Language Reference Guide* for the default device LUN numbers.

Disk devices are different from all the other devices in that they allow files to be opened. Each program task can have one file open on each disk LUN. That is, each program task can have multiple files open simultaneously (on the same or different disk units).

> **NOTE:** No more than 60 disk files can be open by the entire system at any time. That includes files opened by programs and by the system monitor (for example, for the FCOPY command). The error *Device not ready* results if an attempt is made to open a 61st file.

For details on accessing the graphics window LUNs, see Graphics Programming on page 161.

Error Status

Unlike most other eV+ instructions, I/O operations are expected to fail under certain circumstances. For example, when reading a file, an error status is returned to the program to indicate when the end of the file is reached. The program is expected to handle this error and continue execution. Similarly, a serial line may return an indication of a parity error, which should cause the program to retry a data transmission sequence.

For these reasons, eV+ I/O instructions normally do not stop program execution when an error occurs. Instead, the success or failure of the operation is saved internally for access by the IOSTAT real-valued function. For example, a reference to IOSTAT(5) returns a value indicating the status of the last I/O operation performed on LUN 5. The values returned by IOSTAT fall into one of following three categories:

IOSTAT Return Values

Value	Explanation
1	The I/O operation completed successfully.
0	The I/O operation has not yet completed. This value appears only if a pre-read or no-wait I/O is being performed.
<0	The I/O operation completed with an error. The error code indicates

Value	Explanation
	what type of error occurred.

The error message associated with a negative value from IOSTAT can be found in the *eV+ Language Reference Guide*. The $ERROR string function can be used in a program (or with the LISTS monitor command) to generate the text associated with most I/O errors.

It is good practice to use IOSTAT to check each I/O operation performed, even if you think it cannot fail (hardware problems can cause unexpected errors).

> **NOTE:** It is not necessary to use IOSTAT after use of a GETC function, since errors are returned directly by the GETC function.

Attaching/Detaching Logical Units

In general, an I/O device must be attached using the ATTACH instruction before it can be accessed by a program. Once a specific device (such as the manual control pendant) is attached by one program task, it cannot be used by another program task. Most I/O requests fail if the device associated with the referenced LUN is not attached.

Each program task has its own sets of disk and graphics logical units. Thus, more than one program task can attach the same logical unit number in those groups at the same time without interference.

A physical device type can be specified when the logical unit is attached. If a device type is specified, it supersedes the default, but only for the logical unit attached. The specified device type remains selected until the logical unit is detached.

An attach request can optionally specify immediate mode. Normally, an attach request is queued, and the calling program is suspended if another control program task is attached to the device. When the device is detached, the next attachment in the queue will be processed. In immediate mode, the ATTACH instruction completes immediately-with an error if the requested device is already attached by another control program task.

With eV+ systems, attach requests can also specify no-wait mode. This mode allows an attach request to be queued without forcing the program to wait for it to complete. The IOSTAT function must then be used to determine when the attach has completed.

If a task is already attached to a logical unit, it will get an error immediately if it attempts to attach again without detaching, regardless of the type of wait mode specified.

When a program is finished with a device, it detaches the device with the DETACH program instruction. This allows other programs to process any pending I/O operations.

When a control program completes execution normally, all I/O devices attached by it are automatically detached. If a program stops abnormally, however, most device attachments are preserved. If the control program task is resumed and attempts to reattach these logical

units, it may fail because of the attachments still in effect. The KILL monitor command forces a program to detach all the devices it has attached.

If attached by a program, the terminal and manual control pendant are detached whenever the program halts or pauses for any reason, including error conditions and single-step mode. If the program is resumed, the terminal and the manual control pendant are automatically reattached if they were attached before the termination.

> **NOTE:** It is possible that another program task could have attached the terminal or manual control pendant. That would result in an error message when the stopped task is restarted.

Reading

The READ instruction processes input from all devices. The basic READ instruction issues a request to the device attached on the indicated LUN and waits until a complete data record is received before program execution continues. (The length of the last record read can be obtained with the IOSTAT function with its second argument set to 2.)

The GETC real-valued function returns the next data byte from an I/O device without waiting for a complete data record. It is commonly used to read data from the serial lines or the system terminal. It also can be used to read disk files in a byte-by-byte manner.

Special mode bits to allow reading with no echo are supported for terminal read operations. Terminal input also can be performed using the PROMPT instruction.

The GETEVENT instruction can be used to read input from the system terminal. This may be useful in writing programs that operate on both graphics and nongraphics-based systems.

To read data from a disk device, a file must be open on the corresponding logical unit. The FOPEN_ instructions open disk files.

Writing

The WRITE instruction processes output to serial and disk devices and to the terminal. The basic WRITE instruction issues a request to the device attached on the indicated LUN, and waits until the complete data record is output before program execution continues.

WRITE instructions accept format control specifiers that determine how output data is formatted, and whether or not an end of record mark should be written at the end of the record.

Terminal output also can be performed using the PROMPT or TYPE instructions.

A file must be open using the FOPENW or FOPENA instructions before data can be written to a disk device. FOPENW opens a new file. FOPENA opens an existing file and appends data to that file.

Input Wait Modes

Normally, eV+ waits until the data from an input instruction is available before continuing with program execution. However, the READ instruction and GETC function accept an optional argument that specifies no-wait mode. In no-wait mode, these instructions return immediately with the error status -526 (No data received) if there is no data available. A program can loop and use these operations repeatedly until a successful read is completed or until some other error is received.

The disk devices do not recognize no-wait mode on input and treat such requests as normal input-with-wait requests.

Output Wait Modes

Normally, eV+ waits for each I/O operation to be completed before continuing to the next program instruction. For example, the instruction:

```
TYPE /X50
```

causes eV+ to wait for the entire record of 50 spaces to be transmitted (about 50 milliseconds with the terminal set to 9600 baud) before continuing to the next program instruction.

Similarly, WRITE instructions to serial lines or disk files will wait for any required physical output to complete before continuing.

This waiting is not performed if the /N (no wait) format control is specified in an output instruction. Instead, eV+ immediately executes the next instruction. The IOSTAT function checks whether or not the output has completed. It returns a value of zero if the previous I/O is not complete.

If a second output instruction for a particular LUN is encountered before the first no-wait operation has completed, the second instruction automatically waits until the first is done. This scheme means the no-wait output is effectively double-buffered. If an error occurs in the first operation, the second operation is canceled, and the IOSTAT value is correct for the first operation.

With eV+, the IOSTAT function can be used with a second argument of 3 to explicitly check for the completion of a no-wait write.

Disk I/O

> **NOTE:** The ACE software provides the following functionality through its graphical user interface. Therefore, Omron Adept strongly recommends that you use the ACE software.

The following sections discuss disk I/O.

Attaching Disk Devices

A disk LUN refers to a local disk device, such as the SDI card in a SmartController EX system. Also, a remote disk may be accessed via a network.

The type of device to be accessed is determined by the DEFAULT command or the ATTACH instruction. If the default device type set by the DEFAULT command is not appropriate at a particular time, the ATTACH instruction can be used to override the default. The syntax of the ATTACH instruction is:

```
ATTACH (lun, mode) $device
```

See the documentation for the ATTACH program instruction for the mode options and possible device names. The instruction:

```
ATTACH (dlun, 4) "DISK"
```

attaches to an available disk logical unit and returns the number of the logical unit in the variable dlun, which can then be used in other disk I/O instructions.

If the device name is omitted from the instruction, the default device for the specified LUN is used. Omron Adept recommends that you always specify a device name with the ATTACH instruction. (The device SYSTEM refers to the device specified with the DEFAULT monitor command.)

Once the attachment is made, the device cannot be changed until the logical unit is detached. However, any of the units available on the device can be specified when opening a file. For example, the eV+ DISK units are A, C and D. After attaching a DISK device LUN, a program can open and close files on either of these disk units before detaching the LUN.

Disk I/O and the Network File System (NFS)

In addition to local disk devices, an Omron Adept system equipped with Ethernet hardware and the TCP/IP license can access remote disk drives in the same fashion as local disks.

The following sections describe accessing a disk drive regardless of whether it is a local drive or a remotely-accessed drive.

Disk Directories

The FOPEN_ instructions, which open disk files for reading and writing, use directory paths in the same fashion as the monitor commands LOAD, STORE, etc. Files on a disk are grouped in

directories. If a disk is thought of as a file cabinet, then a directory can be thought of as a drawer in that cabinet. Directories allow files (the file folders in our file cabinet analogy) that have some relationship to each other to be grouped together and separated from other files. See the chapter Using Files in the *eV+ Operating System User's Guide* for more details on the directory structure.

Disk File Operations

All I/O requests to a disk device are made to a file on that device. A disk file is a logical collection of data records[1] on a disk. Each disk file has a name, and all the names on a disk are stored in a directory on the disk. The FDIRECTORY monitor command displays the names of the files on a disk.

A disk file can be accessed either sequentially, where data records are accessed from the beginning of the file to its end, or randomly, where data records are accessed in any order. Sequential access is simplest and is assumed in this section. Random access is described later in this chapter.

Opening a Disk File

Before a disk file can be opened, the disk the file is on must be ATTACHed.

The FOPEN_ instructions open disk files (and file directories). These instructions associate a LUN with a disk file. Once a file is open, the READ, GETC, and WRITE instructions access the file. These instructions use the assigned LUN to access the file so that multiple files may be open on the same disk and the I/O operations for the different disk files will not affect each other.[2]

The simplified syntax for FOPEN_ is:

```
FOPEN_ (lun) file_spec
```

where:

lun	logical unit number used in the ATTACH instruction
file_spec	file specification in the form, unit:path\filename.ext

	unit	is an optional disk unit name. The standard local disk units are A, C, and D. If no unit is specified, the colon also must be omitted. Then the default unit (as determined by the DEFAULT command) is assumed.
	path\	is an optional directory path string. The directory path is defined by one or more directory names, each followed by a \ character. The actual directory path is determined by combining any specified path with the path set by the DEFAULT command. If path is

preceded with a \, the path is absolute. Otherwise, the path is relative and is added to the current DEFAULT path specification. (If unit is specified and is different from the default unit, the path is always absolute.)

filename   is a name with 1 to 8 characters, which is used as the name of the file on the disk.

ext   is the filename extension-a string with 0 to 3 characters, which is used to identify the file type.

The four open commands are:

1. Open for read only (FOPENR). If the disk file does not exist, an error is returned. No write operations are allowed, so data in the file cannot be modified.

2. Open for write (FOPENW). If the disk file already exists, an error is returned. Otherwise, a new file is created. Both read and write operations are allowed.

3. Open for append (FOPENA). If the disk file does not exist, a new file is created. Otherwise, an existing file is opened. No error is returned in either case. A sequential write or a random write with a zero record number appends data to the end of the file.

4. Open for directory read (FOPEND). The last directory in the specified directory path is opened. Only read operations are allowed. Each record read returns an ASCII string containing directory information. Directories should be opened using variable-length sequential-access mode.

While a file is open for write or append access, another control program task cannot access that file. However, multiple control program tasks can access a file simultaneously in read-only mode.

Writing to a Disk

The instruction:

```
WRITE (dlun) $in.string
```

writes the string stored in $in.string to the disk file open on dlun. The instruction:

```
error = IOSTAT(dlun)
```

returns any errors generated during the write operation.

Reading From a Disk

The instruction:

```
READ (dlun) $in.string
```

reads (from the open file on dlun) up to the first CR/LF (or end of file if it is encountered) and store the result in $in.string. When the end of file is reached, eV+ error number -504 Unexpected end of file is generated. The IOSTAT() function must be used to recognize this error and halt reading of the file:

```
DO
    READ (dlun) $in.string
    TYPE $in.string
UNTIL IOSTAT(dlun) == -504
```

The GETC function reads the file byte by byte if you want to examine individual bytes from the file (or if the file is not delimited by CR/LFs).

Detaching

When a disk logical unit is detached, any disk file that was open on that unit is automatically closed. However, error conditions detected by the close operation may not be reported. Therefore, it is good practice to use the FCLOSE instruction to close files and to check the error status afterwards. FCLOSE ensures that all buffered data for the file is written to the disk, and updates the disk directory to reflect any changes made to the file. The DETACH instruction frees up the logical unit. The following instructions close a file and detach a disk LUN:

```
FCLOSE (dlun)
    IF IOSTAT(dlun) THEN
        TYPE $ERROR(IOSTAT(dlun))
    END
DETACH (dlun)
```

When a program completes normally, any open disk files are automatically closed. If a program stops abnormally and execution does not proceed, the KILL monitor command closes any files left open by the program.

CAUTION: While a file is open on a floppy disk, do not replace the floppy disk with another disk: Data may be lost and the new disk may be corrupted.

Disk I/O Example

The following example creates a disk file, writes to the file, closes the file, reopens the file, and reads back its contents.

```
        AUTO dlun, i
        AUTO $file.name
        $file.name = "data.tst"

; Attach to a disk logical unit
        ATTACH (dlun, 4) "DISK"
```

```
        IF IOSTAT(dlun) < 0 GOTO 100

; Open a new file and check status
        FOPENW (dlun) $file.name
        IF IOSTAT(dlun) < 0 GOTO 100

; Write the text
        FOR i = 1 TO 10
            WRITE (dlun) "Line "+$ENCODE(i)
            IF IOSTAT(dlun) < 0 GOTO 100
        END

; Close the file
        FCLOSE (dlun)
        IF IOSTAT(dlun) < 0 GOTO 100

; Reopen the file and read its contents
        FOPENR (dlun) $file.name
        IF IOSTAT(dlun) < 0 GOTO 100
        READ (dlun) $txt
        WHILE IOSTAT(dlun) > 0 DO
            TYPE $txt
            READ (dlun) $txt
        END            ;End of file or error
        IF (IOSTAT(dlun) < 0) AND (IOSTAT(dlun) <> -504) THEN
  100     TYPE $ERROR(IOSTAT(dlun))      ;Report any errors
        END
        FCLOSE (dlun)                         ;Close the file
        IF IOSTAT(dlun) < 0 THEN
            TYPE $ERROR(IOSTAT(dlun))
        END
        DETACH (dlun)
```

[1]A variable-length record is a text string terminated by a CR/LF (ASCII 13/ASCII 10).

[2]When accessing files on a remote system, the unit can be any name string, and the file name and extension can be any arbitrary string of characters.

Advanced Disk Operations

This section introduces additional parameters to the FOPEN and FOPENR program instructions. For details, see the FOPEN and FOPENR documentation in the *eV+ Language Reference Guide* for details.

Variable-Length Records

The default disk file access mode is variable-length record mode. In this mode, records can have any length (up to a maximum of 512 bytes) and can cross the boundaries of 512-byte sectors. The end of a record is indicated by a Line-Feed character (ASCII 10). Also, the end of the file is indicated by the presence of a Ctrl+Z character (26 decimal) in the file. Variable-length records should not contain any internal Line-Feed or Ctrl+Z characters as data. This format is used for loading and storing eV+ programs, and is compatible with the IBM PC standard ASCII file format.

Variable-length record mode is selected by setting the record length parameter in the FOPEN_ instruction to zero, or by omitting the parameter completely. In this mode, WRITE instructions automatically append Return (ASCII 13) and Line-Feed characters to the output data-which makes it a complete record. If the /S format control is specified in an output specification, no Return/Line-Feed is appended. Then any subsequent WRITE will have its data concatenated to the current data as part of the same record. If the /Cn format control is specified, n Return/Line-Feeds are written, creating multiple records with a single WRITE.

When a variable-length record is read using a READ instruction, the Return/Line-Feed sequence at the end is removed before returning the data to the eV+ program. If the GETC function is used to read from a disk file, **all** characters are returned as they appear in the file-including Return, Line-Feed, and Ctrl+Z characters.

Fixed-Length Records

In fixed-length record mode, all records in the disk file have the same specific length. Then there are no special characters embedded in the file to indicate where records begin or end. Records are contiguous and may freely cross the boundaries of 512-byte sectors.

Fixed-length record mode is selected by setting the record length parameter in the FOPEN_ instruction to the size of the record, in bytes. WRITE instructions then pad data records with zero bytes or truncate records as necessary to make the record length the size specified. No other data bytes are appended, and the /S format control has no effect.

In fixed-length mode, READ instructions always return records of the specified length. If the length of the file is such that it cannot be divided into an even number of records, a READ of the last record will be padded with zero bytes to make it the correct length.

Sequential-Access Files

Normally, the records within a disk file are accessed in order from the beginning to the end without skipping any records. Such files are called sequential files. Sequential-access files

may contain either variable-length or fixed-length records.

Random-Access Files

In some applications, disk files need to be read or written in a nonsequential or random order. eV+ supports random access only for files with fixed-length records. Records are numbered starting with 1. The position of the first byte in a random-access record can be computed by:

```
byte_position = 1 + (record_number -1) * record_length
```

Random access is selected by setting the random-access bit in the mode parameter of the FOPEN_ instruction. A nonzero record length must also be specified.

A specific record is accessed by specifying the record number in a READ or WRITE instruction. If the record number is omitted, or is zero, the record following the one last accessed is used. (See the FOPEN documentation.)

Buffering and I/O Overlapping

All physical disk I/O occurs as 512-byte sector reads and writes. Records are unpacked from the sector buffer on input, and additional sectors are read as needed to complete a record. To speed up read operations, eV+ automatically issues a read request for the next sector while it is processing the current sector. This request is called a preread. Preread is selected by default for both sequential-access and random-access modes. It can be disabled by setting a bit in the mode parameter of the FOPEN_ instruction. If prereads are enabled, opening a file for read access immediately issues a read for the first sector in the file.

Preread operations may actually degrade system performance if records are accessed in truly random order, since sectors would be read that would never be used. In this case, prereads should be disabled and the FSEEK instruction should be used to initiate a preread of the next record to be used.

The function IOSTAT(lun, 1) returns the completion status for a pending preread or FSEEK operation.

On output, records are packed into sector buffers and written after the buffers are filled. If no-wait mode is selected for a write operation by using the /N format control, the WRITE instruction does not wait for a sector to be written before allowing program execution to continue.

In random-access mode, a sector buffer is not normally written to disk until a record not contained in that buffer is accessed. The FEMPTY instruction empties the current sector buffer by immediately writing it to the disk.

A file may be opened in nonbuffered mode, which is *much slower* than normal buffered mode, but it guarantees that information that is written will not be lost due to a system crash or power failure. This mode was intended primarily for use with log files that are left

opened over an extended period of time and intermittently updated. For these types of files, the additional (significant) overhead of this mode is not as important as the benefit.

When a file is being created, information about the file size is not stored in the disk directory until the file is closed. Closing a file also forces any partial sector buffers to be written to the disk. Note that aborting a program does not force files associated with it to be closed. The files are not closed (and the directory is not updated) until a KILL command is executed or until the aborted program is executed again.

Disk Commands

There are several disk-oriented monitor commands that do not have a corresponding program instruction. The FCMND instruction must be used to perform the following actions from within a program:

- Rename a file

- Format a disk

- Create a subdirectory

- Delete a subdirectory

The MCS instruction can be used to issue an FCOPY command from within a program.

FCMND is similar to other disk I/O instructions in that a logical unit must be attached and the success or failure of the command is returned via the IOSTAT real-valued function. For details, see the documentation for the FCMND program instruction.

The FCMND instruction is described in detail in the *eV+ Language Reference Guide*. See the *MV Controller User's Guide*

Accessing the Disk Directories

The eV+ directory structure is identical to that used by the IBM PC DOS operating system (version 2.0 and later). For each file, the directory structure contains the file name, attributes, creation time and date, and file size. Directory entries may be read after successfully executing an FOPEND instruction.

Each directory record returned by a READ instruction contains an ASCII string with the information shown in the following table.

Disk Directory Format

Byte	Size	Description
1-7	7	Attribute codes, padded with blanks on right

Byte	Size	Description
9	1	ASCII tab character (9 decimal)
10-19	10	ASCII file size, in sectors, right justified
20	1	ASCII tab character (9 decimal)
20-28	9	File revision date in the format dd-mm-yy
29	1	ASCII tab character (9 decimal)
30-38	8	File revision time in the format hh:mm:ss
39	1	ASCII tab character (9 decimal)
40-	8	ASCII file name and extension (size depends on file name size)

The following characters are possible in the file attribute code field of directory entries:

File Attribute Codes

Character	Meaning
D	Entry is a subdirectory
L	Entry is the volume label (not supported by eV+)
P	File is protected and cannot be read or modified
R	File is read-only and cannot be modified
S	File is a system file

The attribute field is blank if no special attributes are indicated.

The file revision date and time fields are blank if the system date and time had not been set when the file was created or last modified. (The system date and time are set with the TIME monitor command or program instruction.)

Serial Line I/O

The eV+ controller has several serial lines that are available for general use. This section describes how these lines are used for simple serial communications.

I/O Configuration

In addition to selecting the protocol to be used, the controller configuration program allows the baud rate and byte format for each serial line to be defined. Once the serial line configuration is defined on the eV+ system boot disk, the serial lines are set up automatically when the eV+ system is loaded and initialized. The following byte formats are available:

- Byte data length of 7 or 8 bits, not including parity

- One or two stop bits

- Parity disabled or enabled

- Odd or even parity (adds 1 bit to byte length)

The following baud rates are available:

110, 300, 600, 1200, 2400, 4800, 7200, 9600, 19200, 38400

In addition, eV+ provides automatic buffering with optional flow control for each serial line. The I/O configuration program can be used to enable output flow control with which eV+ recognizes Ctrl+S (19 decimal) and Ctrl+Q (17 decimal) and uses them to suspend and resume, respectively, serial line output. The configuration program can also enable input flow control, with which eV+ generates Ctrl+S and Ctrl+Q to suspend and resume, respectively, input from an external source. With Ctrl+S and Ctrl+Q flow control disabled, all input and output is totally transparent, and all 8-bit data bytes can be sent and received.

Serial lines may also be configured to use hardware modem control lines for flow control. (The RTS/CTS lines must be installed in the modem cable-standard modem cables often leave these lines out.) For pin assignments, see the documentation on serial I/O connectors in the SmartController EX User's Guide.

Attaching/Detaching Serial I/O Lines

Serial lines must be attached before any I/O operations can take place. Note that only one control program task can be attached to a single serial line at any one time. All other attachment requests will queue or fail, depending on the setting of the mode parameter in the ATTACH program instructions.

Attaching or detaching a serial line automatically stops any output in progress and clears all input buffers. Serial lines are not automatically detached from a program unless it completes with success, so it is possible to single-step through a program or proceed from a PAUSE instruction without loss of data.

Input Processing

Input data is received by eV+ according to the byte format specified by the I/O configuration program. The size of the buffer can be set with the ACE software Controller Configuration Tools. Data errors such as parity or framing errors are also buffered and are returned in the proper order.

The possible data errors from the serial input lines are:

-522   *Data error on device*

A data byte was received with incorrect parity, or the byte generated a framing error.

-524   *Communications overrun*

Data bytes were received after the input buffer was full, or faster than eV+ could process them.

-526   *No data received*

If data is expected, continue polling the serial line.

-504   *Unexpected end of file*

A BREAK was received from the remote device.

Serial line input data is normally read using the GETC function, since it allows the most flexible response to communications errors. The READ instruction also can be used provided that input data is terminated by a Line-Feed character (10 decimal).

eV+ does not support input echoing or input line editing for the serial lines.

Output Processing

All serial line output is performed using the WRITE instruction. All binary data (including NULL characters) is output without conversion. If the serial line is configured to support parity, a parity bit is automatically appended to each data byte.

By default, the WRITE instruction appends a Return character (13 decimal) and a Line-Feed character (10 decimal) to each data record unless the /S format control is specified in the instruction parameter list.

If output flow control is enabled and output has been suspended by a Ctrl+S character from the remote device, a WRITE request may wait indefinitely before completing.

Serial I/O Examples

The first example attaches to a serial line and performs simple WRITEs and READs on the line:

```
.PROGRAM serial.io()
; ABSTRACT: Example program to write and read lines of
; text to and from serial port 1 on the SIO module.

        AUTO slun ;Logical unit to communicate to serial port
        AUTO $text

; Attach to a logical unit(open communications path
; to serial port)

        ATTACH (slun, 4) "SERIAL:1"
        IF IOSTAT(slun) < 0 GOTO 100

; Write text out to the serial port

        WRITE (slun) "Hello there! "
        IF IOSTAT(slun) < 0 GOTO 100

; Read a line of text from the serial port. The incoming
; line of text must be terminated by a carriage return and
; line feed. The READ instruction will wait until a line of
; text is received.

        READ (slun) $text
        IF IOSTAT(slun) < 0 GOTO 100

; Display any errors

100  IF (IOSTAT(slun) < 0) THEN
            TYPE IOSTAT(slun), " ", $ERROR(IOSTAT(slun))
        END

        DETACH (slun)    ;Detach from logical unit
.END
```

The next example reads data from a serial line using the GETC function with no-wait mode.
Records that are received are displayed on the terminal. In this program, data records on the
serial line are assumed to be terminated by an ETX character, which is not displayed. An
empty record terminates the program.

```
.PROGRAM display()
; ABSTRACT: Monitor a serial line and read data when
; available
        AUTO $buffer, c, done, etx, ienod, line
        etx = 3                   ;ASCII code for ETX character
        ienod = -526              ;Error code for no data
        ATTACH (line, 4) "SERIAL:1"
        IF IOSTAT(line) < 0 GOTO 90 ;Check for errors
        $buffer = ""              ;Initialize buffer to empty
        done = FALSE              ;Assert not done
        DO
            CLEAR.EVENT
            c = GETC(line,1)      ;Read byte from the ser. line
```

```
          WHILE c == ienod DO     ;While there is no data...
              WAIT.EVENT 1 ;Wait for an event
              CLEAR.EVENT
              c = GETC(line,1)     ;Read byte from the ser. line
          END
          IF c < 0 GOTO 90        ;Check for errors
          IF c == etx THEN        ;If ETX seen...
              TYPE $buffer, /N     ;Type buffer
              done = (LEN($buffer) == 0)      ;Done if buffer length is 0
              $buffer = ""         ;Set buffer to empty
          ELSE
              $buffer = $buffer+$CHR(c)  ;Append next byte
                ;to buffer

END
       UNTIL done                        ;Loop until empty buffer seen

       GOTO 100                          ;Exit
    90 TYPE "SERIAL LINE I/O ERROR: ", $ERROR(IOSTAT(line))
       PAUSE
   100 DETACH (line)
       RETURN
.END
```

DeviceNet

Omron Adept supports DeviceNet and DeviceNet protocols on the SmartController. For more information on the DeviceNet environment, hardware and software configuration and eV+ programming for DeviceNet components, select a topic from the table below.

To...	Refer to...
Learn about the DeviceNet Environment	*Adept SmartController EX User's Guide*
Configure DeviceNet hardware	*Adept SmartController EX User's Guide*
Configure DeviceNet software	Configuring the Controller as a DeviceNet Slave Change DeviceNet Configuration
Managing DeviceNet components from the eV+ operating system and program environment	DEVICENET — Used for reading DeviceNet status. DN.THROTTLE — On SmartController systems, allows you to specify the number of nodes to be polled by the DeviceNet drivers to increase CPU availability. ATTACH — Makes a device available for use by an application program. FCMND — Generates a device-specific command to the input/output device specified by the logical unit. The FCMND documentation provides the DeviceNet command codes and the format of DeviceNet status information that is available to programs.

Summary of I/O Operations

The following table summarizes the eV+ I/O instructions:

System Input/Output Operations

Keyword	Type	Function
ATTACH	PI	Make a device available for use by the application program.
BITS	PI	Set or clear a group of digital signals based on a value.
BITS	RF	Read multiple digital signals and return the value corresponding to the binary bit pattern present on the signals.
$DEFAULT	SF	Return a string containing the current system default device, unit, and directory path for disk file access.
DEF.DIO	PI	Assign third-party digital I/O boards to standard eV+ signal numbers, for use by standard eV+ instructions, functions, and monitor commands.This instruction requires the Third-Party Board Support license.
DETACH	PI	Release a specified device from the control of the application program.
DEVICE	PI	Send a command or data to an external device and, optionally, return data back to the program. (The actual operation performed depends on the device referenced.)
DEVICE	RF	Return a real value from a specified device. The value may be data or status information, depending upon the device and the parameters.
DEVICES	PI	Send commands or data to an external device and optionally return data. The actual operation performed depends on the device referenced.
FCLOSE	PI	Close the disk file, graphics window, or graphics icon currently open on the specified logical unit.
FCMND	PI	Generate a device-specific command to the input/output device specified by the logical unit.

Keyword	Type	Function
FEMPTY	PI	Empty any internal buffers in use for a disk file or a graphics window by writing the buffers to the file or window if necessary.
FOPENR	PI	Open a disk file for read-only.
FOPENW	PI	Open a disk file for read-write.
FOPENA	PI	Open a disk file for read-write-append.
FOPEND	PI	Open a disk directory for read.
FSEEK	PI	Position a file open for random access and initiate a read operation on the specified record.
GETC	RF	Return the next character (byte) from a device or input record on the specified logical unit.
IOSTAT	RF	Return status information for the last input/output operation for a device associated with a logical unit.
PROMPT	PI	Display a string on the system terminal and wait for operator input.
READ	PI	Read a record from an open file or from an attached device that is not file oriented.
RESET	PI	Turn off all the external output signals.
SETDEVICE	PI	Initialize a device or set device parameters. (The actual operation performed depends on the device referenced.)
SIG	RF	Return the logical AND of the states of the indicated digital signals.
SIG.INS	RF	Return an indication of whether or not a digital I/O signal is configured for use by the system, or whether or not a software signal is available in the system.
SIGNAL	PI	Turn on or off external digital output signals or internal software signals.

Keyword	Type	Function
TYPE	PI	Display the information described by the output specifications on the system terminal. A blank line is output if no argument is provided.
WRITE	PI	Write a record to an open file or to an attached device that is not file oriented.
PI: Program Instruction, RF: Real-Valued Function, P: Parameter, SF: String Function		

Graphics Programming

NOTE: This feature is not supported in eV+ version v2.x.

The ACE software provides a graphical interface for programming your motion (and vision) system. Further, the User Interface Designer, which is included with the ACE software, provides a complete tool set for building custom interfaces for your applications. Therefore, Omron Adept strongly recommends that you use the ACE software for this functionality. For more details, see the chapter User Interface Designer in the *ACE User's Guide*.

Creating Windows

eV+ communicates to windows through logical units, with logical unit numbers (LUNs) 20 to 23 reserved for window use. (Each task has access to its own set of four LUNs.) The basic strategy for using a window (or any of the graphics instructions) is:

1. ATTACH to a logical unit

2. FOPEN a window on the logical unit

3. Perform the window's tasks (or graphics operations)

4. FCLOSE the window

5. FDELETE the window

6. DETACH from the logical unit

ATTACH Instruction

The ATTACH instruction sets up a communications path so a window can be written to and read from. The syntax for the ATTACH instruction is:

```
ATTACH (glun, 4) "GRAPHICS"
```

glun   variable that receives the number of the attached graphics logical unit. (All menus and graphics commands that take place within a window will also use glun.)

FOPEN Instruction

FOPEN creates a new window or reselects an existing window for input and output. When a window is created, its name is placed in the list of available windows displayed when the **adept** logo is clicked on. The simplified syntax for FOPEN is:

```
FOPEN (glun) "window_name /MAXSIZE width height"
```

glun	The logical unit already ATTACHed to.
window_ name	The title that appears at the top of the window. Also used to close and select the window.
width/height	Specify the largest size the window can be opened to.

This instruction will give you a window with all the default attributes. See the description of FOPEN and FSET in the eV+ Language Reference Guide for details on how to control the attributes of a window e6 for example, background color, size, and scrolling.

FCLOSE Instruction

FCLOSE closes a window to input and output (but does not erase it or remove it from memory). The syntax for FCLOSE is:

```
FCLOSE (glun)
```

glun   The logical unit number specified in the FOPEN instruction that opened the window.

FDELETE Instruction

FDELETE removes a closed, attached window from the screen and from graphics memory. The syntax for FDELETE is

```
FDELETE (glun) "window_name"
```

glun   The same values as specified in the FOPEN instruction that created the window.

DETACH Instruction

DETACH frees up a LUN for use by a subsequent ATTACH instruction. The syntax for DETACH is:

```
DETACH (glun)
```

glun   The LUN specified in a previous ATTACH instruction.

Custom Window Example

This section of code will create and delete a window:

```
        AUTO glun                          ; Graphics window LUN

        ATTACH (glun, 4) "GRAPHICS"        ; Attach to a window LUN

; Open the window "Test" with a maximum size of
; 400 x 300 pixels

        FOPEN(glun) "Test","/MAXSIZE 400 300"

; Your code for processing within the window
; goes here; e.g:

        GTYPE (glun) 10, 10, "Hello!"

; When the window is no longer needed, close and delete the
; window and detach from the logical unit

        FCLOSE (glun)
```

```
FDELETE (glun) "Test"
DETACH (glun)
```

Monitoring Events

The key to pointing-device-driven programming is an event loop. In an event loop, you wait for an event (from the keyboard or pointer device) and when the correct event occurs in the proper place, your program initiates some appropriate action. eV+ can monitor many different events including button up, button down, double click, open window, and menu select. The example code in the following sections will use event 2, button up, and event 14, menu select. For details on the different events that can be monitored, see the documentation for the GETEVENT program instruction in the *eV+ Language Reference Guide*.

The basic strategy for an event loop is:

1. Wait for an event to occur.

2. When an event is detected:

    a. If it is the desired event, go to step 3

    b. Otherwise, return to step 1.

3. Check the data from the event array (not necessary for event 14, menu select):

    a. If it is appropriate, go to step 4.

    b. Otherwise, return to step 1.

4. Initiate appropriate action.

5. Return to step 1.

GETEVENT Instruction

The instruction that initiates monitoring of pointer device and keyboard events is GETEVENT. Its simplified syntax is:

```
GETEVENT (lun) event[]
```

lun	Logical unit number of the window to be monitored.
event[]	Array into which the results of the detected event are stored. The value stored in event[0] indicates which event was detected.
If event[0]	is 2, a button-up event was detected, in which case:
event[1]	indicates the number of the button pressed. (For two-button devices, 2 = left button, 4 = right button. For three-button devices, 1 = left button, 2 = middle button, 4 = right button.)
event[2]	is the X value of the pointer location of the click.
event[3]	is the Y value of the pointer location of the click.

If event[0]   is 14, a click on a menu bar selection was detected, in which case:

If event[1]   is 0, a click has been made to the top-level menu bar. In this case, an FSET instruction must be executed to display the pull-down options under the menu bar selection and event[2] is the number (from left to right) of the menu bar option selected.

If event[1]   is 1, then a selection from a pull-down menu has been made and event[2] is the number of the pull-down option selected.

You cannot use the GETEVENT instruction to specify which events to monitor. It monitors all the events that are enabled for the window. For details on using the /EVENT argument for enabling and disabling the monitoring of various events, see the documentation for the FOPEN and FSET program instructions in the *eV+ Language Reference Guide*.

FSET Instruction

FSET is used to alter the characteristics of a window opened with an FOPEN instruction, and to display pull-down menus. We are going to describe only the use of FSET to create the top-level menu bar, create the pull-down menu selections below the top-level menu, and initiate monitoring of events. The instruction for displaying a top-level menu is:

```
FSET (glun) " /MENU 'item1' 'item2' ... 'item10' "
```

glun   is the logical unit of the window the menu is displayed in.

item1-item10   are the menu titles for a top-level bar menu. The items appear
                     from left to right.

The instruction to display a pull-down menu (called when event[0] = 14 and event[1] = 0) is:

```
FSET (glun) "/PULLDOWN", top_level#," 'item1' ... ' itemn '"
```

top_level#   is the number of the top-level selection the pull-down menu is to appear
             under.

item1-   are the menu items in the pull-down menu. The items appear from top
itemn   to bottom.

The relationship between these two uses of FSET will become clear when we actually build a menu structure.

The basic FSET instruction for monitoring menu and mouse events is:

```
        FSET (glun) "/EVENT BUTTON MENU"
```

Building a Menu Structure

The strategy for implementing a menu is:

1. Declare the top-level bar menu.

2. Start a loop monitoring event 14 (menu selection).

3. When event 14 is detected, check to see if the mouse event was on the top-level bar menu or on a pull-down option.

4. If the event was a top-level menu selection, then display the proper pull-down options.

5. If the event was a pull-down selection, use nested CASE structures to take appropriate action based on the selections made to the top-level menu and its corresponding pull-down menu.

Menu Example

This code segment will implement a menu structure for a window open on glun:

```
; Set the top-level menu bar and enable monitoring of events
        FSET (glun) "/menu 'Menu 1' 'Menu 2' 'Menu 3'"
        FSET (glun) "/event button menu"

; Define the strings for the pull-down menus
        $menu[1] = "'Item 1-1' 'Item 1-2'"
        $menu[2] = "'Item 2-1' 'Item 2-2' 'Item 2-3'"
        $menu[3] = "'Quit'"

; Set variable for event to be monitored
        wn.e.menu = 14
; Start the processing loop
        quit = FALSE
        DO
            GETEVENT (glun) event[]
            IF event[0] == wn.e.menu THEN
;The menu event (14) has two components; a button-down component
;    corresponding to a click on a menu bar selection, and a
;    button-up component corresponding to the pull-down selection
;    made when the button is released.
;    After the first component (pointer down on the menu bar),
;    event[1] will be 0 and event[2] will have the number of the
;    menu bar selection.

; Check to see if event[1] is 0, indicating a top-level menu select
                IF event[1] == 0 THEN
; Use the value in event[2] to select a pull-down menu
                    FSET (glun) "/pulldown", event[2], $menu[event[2]]
; Else, execute the appropriate code for each menu selection
                ELSE

; If event[1] is not 0, then the button has been released on a
```

```
;   pull-down selection and:
;       event[1] will have the value of the top-level selection (menu)
;       event[2] will have the value of the pull-down selection (item)
                  menu = event[1]
                  item = event[2]


; The outer CASE structure checks the top-level menu selection
; The inner CASE structure checks the item selected from the pull-down
                  CASE menu OF
                    VALUE 1: ;Menu 1
                      CASE item OF
                        VALUE 1:
        ;code for Item 1-1
                        VALUE 2:
        ;code for Item 1-2
                      END
                    VALUE 2:                ;Menu 2
                      CASE item OF
                        VALUE 1:
        ;code for Item 2-1
                        VALUE 2:
        ;code for Item 2-2
                        VALUE 3:
        ;code for Item 2-3
                      END
                    VALUE 3:                ;Menu 3
                      CASE item OF
                        VALUE 1:
                          quit = TRUE     ;time to quit
                      END
                  END                       ; case menu of
              END                           ; if event[1]
          END                               ; if event[0]
      UNTIL quit

.END
```

Implementing the above code and then clicking on Menu 2 would result in the window shown in the following figure.
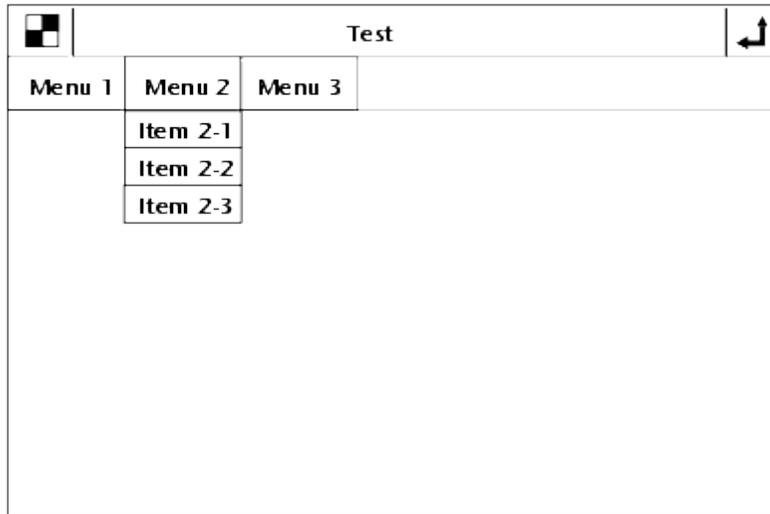
Sample Menu

Defining Keyboard Shortcuts

If you are using AdeptWindows, you can create keyboard shortcuts on menu and pull-down items by placing an ampersand (&) before the desired letter. For example:

```
FSET(lun) "/menu '&File' '&Edit'"
```

In this example, the letters F and E are used as shortcuts when pressed with the ALT key. Thus, pressing ALT+F displays the File menu and ALT+E displays the Edit menu. The letters F and E are underlined on the menu or pull-down item to indicate the keyboard shortcut.

Creating Buttons

Creating a button in a window is a simple matter of placing a graphic representing your button on the screen, and then looking to see if a mouse event occurred within the confines of that graphic.

GPANEL Instruction

The GPANEL instruction is useful for creating standard button graphics. The syntax for GPANEL is:

```
GPANEL (glun, mode) x, y, dx, dy
```

glun   The logical unit of the window the button is in.

mode   is replaced with:

    0   indicating a raised, ungrooved panel

    2   indicating a sunken, ungrooved panel

    4   indicating a raised, grooved panel

    6   indicating a sunken, grooved panel

    (Adding 1 to any of the mode values fills the panel with foreground color.)

x y   Coordinates of the upper left corner of the button.

dx dy   Width and height of the button.

Button Example

This code segment places a button on the screen and then monitor a button-up event at that button (the logical unit the button is accessing must be ATTACHed and FOPENed):

```
; Initialize monitoring of button events for a button
      FSET (glun) "/event button"
; Draw a 45x45 pixel panel at window coordinates 100,100
      GPANEL (glun, 0) 100, 100, 45, 45
; Put a label in the button
      GTYPE (glun) 102, 122, "Label"
; Declare a variable for pointer event 2 (button up)
      btn.up = 2
; Set a variable that will stop the monitoring of button
; events
      hit = FALSE
; Start a loop waiting for a button-up event
      DO
            GETEVENT (glun) event[]
```

```
; The status of a button event will be stored in event[0].; Look to see if
that event was a button-up event.

IF event[0] == btn.up THEN
 ; Check if the button-up event was within the button area
 ; The x location is in event[1], the y location in event[2]
              hit = (event[2] > 99) AND (event[2] < 146)
              hit = hit AND (event[3] > 99) AND (event[3] < 146)
           END
        UNTIL hit
 ; The code for reacting to a button press is placed here.
```

This code will work for a single button but will become very unwieldy if several buttons are used. In the case of several buttons, you should place the button locations in arrays (or a two-dimensional array) and then pass these locations to a subroutine that checks whether the mouse event was within the array parameters passed to it.

Creating a Slide Bar

eV+ allows you to create a eV+feature similar to the window scroll bars called slide bars. The syntax for a slide bar is:

```
GSLIDE (glun, mode) slide_id = x, y, len, max_pos, arrow.inc, handle
```

glun	The logical unit of the window the slide bar is created in.
mode	is replaced with:

    0 indicating a horizontal slide bar is to be created or updated.
    1 indicating a slide bar is to be deleted.
    2 indicating a vertical slide bar is to be created or updated.

slide_id	A number that identifies the slide bar. This number is returned to the event queue so you can distinguish which slide was moved.
x y	The coordinates of the top left corner of the slide bar.
len	The width or height of the bar.
max_pos	Specifies the maximum value the slide bar returns.
arrow_inc	Specifies the increment the slide bar registers when the arrows are clicked. (The slide bar is created with a scroll handle and scroll arrows.)
handle	Specifies position the scroll handle is in when the slide bar is created.

GSLIDE Example

We will be interested in two events when monitoring a slide bar, event 8 (slide bar pointer move) and event 9 (slide bar button up). Additional event monitoring must be enabled with the FSET instruction. Object must be specified to monitor slide bars and move_b2 must be specified to monitor the dragging of the middle button.

The values returned in the GETEVENT array will be:

- event[0] the pointer device event code
- event[1] the ID of the slide bar (as specified by slide_id)
- event[2] the slide bar value
- event[3] the maximum slide bar value

The following code will display and monitor a slide bar:

```
; The slide bar will be in the window open on glun

; The slide bar will use events 8 and 9. A double-click event ; will halt
; monitoring of the slide bar
```

```
        btn.smov = 8
        btn.sup = 9
        btn.dclk = 3

; Slide bar position and start-up values

        x = 20
        y = 60
        length = 200
        max.pos = 100
        arrow_inc = 10
        handle_pos = 50

; Enable monitoring of slide bars and pointer drags

        FSET (glun) "/event object move_b2"

; Display the slide bar

        GSLIDE (glun, 0) 1 = x, y, length, max_pos, arrow_inc, handle_pos
; Begin monitoring events and take action when the slide bar ; is moved.
Monitor
; events until a double click is detected, then delete the
; slide bar

        DO
            GETEVENT (glun) event[]
            IF (event[0] == btn.smov) OR (event[0] == btn.sup THEN
            ; Your code to monitor the slide bar value (event[2]) goes
            ; here

        END
        UNTIL event[0] == btn.dclk

 ; Delete the slide bar

        GSLIDE (glun, 1) 1
```

Graphics Programming Considerations

Buttons and menus can be monitored in the same window. However, the code will get complicated, and you might consider using different windows when the button and menu structure becomes complex.

Only one pull-down menu can be active at any time.

Design your windows with the following mechanical and aesthetic considerations:

- Keep your windows as simple and uncluttered as possible. Use color carefully and purposefully.

- If you are using multiple windows, use similar graphic elements so the screen elements become familiar and intuitive.

- Let the operator know what is going on. Never leave the operator in the dark as to the status of a button push or menu selection.

- Whenever possible, have your windows mimic the real work world of the operator.

In the interest of clarity, the examples in this chapter have not been generalized. When you actually program an application, use generalized subroutine calls for commonly used code, or your code will quickly become unmanageable.

Using IOSTAT( )

The example code in this chapter leaves out critical error detection and recovery procedures. Effective application code requires these procedures. The IOSTAT function should be used to build error-handling routines for use with every ATTACH, FOPEN, FCLOSE, and FSET instruction. The syntax for using IOSTAT to check the status of I/O requests is:

```
IOSTAT(lun)
```

lun   The LUN specified in the previous I/O request.

The IOSTAT function returns the following values:

    1   if the last operation was successful

    0   if the last operation is not yet complete

  < 0   if the last operation failed, a negative number corresponding to a standard Omron Adept error code will be returned.

The following code checks for I/O errors:

```
; Issue I/O instruction (ATTACH, FOPEN, etc.)
    IF IOSTAT(lun) < 0 THEN
      ;your code to handle the error
    END
; The ERROR function can be used to return the text
```

```
; of an error number. The code line is:
     TYPE $ERROR(IOSTAT(lun))
```

Managing Windows

Windows can be:

- Hidden (but not deleted)

  A hidden window is removed from the screen but not from graphics memory, and it can be retrieved at any time:

  ```
  FSET(glun) "/NODISPLAY"          ;Hide a window
  FSET(glun) "/DISPLAY"            ;Redisplay a window
  ```

- Sent behind the parent's window stack:

  ```
  FSET(glun) "/STACK -1"
  ```

- Brought to the front of the window stack:

  ```
  FSET(glun) "STACK 1"
  ```

If you will not be reading events from a window, open it in write-only mode to save memory and processing time.

Only the task that opened a window in read/write mode can read from it (monitor events).

Multiple tasks can write to an open window. A second task can write to an already open window by executing its own ATTACHand OPEN for the window. The logical units' numbers need not match, but the window name must be the same. If a task has the window Test open, other tasks can write to the window by:

```
ATTACH(lun_1, 4) "GRAPHICS"
FOPEN(lun_1) "Test /MAXSIZE 200 200 /WRITEONLY"
```

Communicating With the System Windows

The system has three operating system level windows: the main window, the monitor window, and the vision window (on systems with the AdeptVision option).

The Main Window

You can place menu options on the top-level menu bar by opening the window \Screen_1. For example:

```
ATTACH (glun, 2) "GRAPHICS"
FOPEN(glun)"\Screen_1"
FSET (glun)"/event menu"
FSET (glun)"/menu 'item1' 'item2' 'item3'"
```

opens the main window and place three items on the top-level menu bar. Pull-downs and event monitoring can proceed as described earlier. The instruction:

```
FSET (glun) "/menu "
```

deletes the menu items.

The Monitor Window

The monitor window can be opened in write-only mode to change the characteristics of the monitor window. For example, the following instruction opens the monitor window, disables scrolling, and disallows moving of the window:

```
FOPEN (glun) "Monitor /WRITEONLY /SPECIAL NOPOSITION NOSIZE"
```

To prevent a user from accessing the monitor window, use the instruction:

```
FOPEN (glun) "Monitor /WRITEONLY /NOSELECTABLE"
```

To allow access:

```
FSET (glun) "/SELECTABLE"
```

The Vision Window

For systems equipped with the Adept Vision option, text or graphics can be output to the vision window, and events can be monitored in the vision window. To communicate with the vision window, you open it just as you would any other window. For the window name you must use Vision. For example:

```
FOPEN (glun) "Vision"
```

Remember, graphics output to the vision window is displayed only when a graphics display mode or overlay is selected. When you are finished communicating with the vision window, close and detach from it just as you would any other window. This will free up the logical unit,

but will not delete the vision window. You can close and detach from the vision window, but you cannot delete it.

To preserve the vision system pull-down menus, open the window in write-only mode:

```
FOPEN (glun) "Vision /WRITEONLY"
```

The following example opens the vision window, writes to the vision window, and detaches the vision window:

```
.PROGRAM label.blob()

; ABSTRACT: This program demonstrates how to attach to the
; vision window and how to use the millimeter scaling mode of
; the     GTRANS instruction to label a "blob" in the vision
; window.
;
        AUTO vlun
        cam = 1
        ; Attach the vision window and get a logical unit number
        ATTACH (vlun, 4) "GRAPHICS"
        IF IOSTAT(vlun) < 0 GOTO 100
        FOPEN (vlun) "Vision"           ;Open the vision window
        IF IOSTAT(vlun) < 0 GOTO 100
        ; Select display mode and graphics mode
        VDISPLAY (cam) 1, 1          ;Display grayscale frame and graphics
        ; Take a picture and locate an object
        VPICTURE (cam)               ;Take a processed picture
        VLOCATE (cam, 2) "?"         ;Attempt to locate an object
        IF VFEATURE(1) THEN          ;If an object was found...

            GCOLOR (vlun) 1          ;Select the color black
            GTRANS (vlun, 2)         ;Select millimeter scaling
            GTYPE (vlun) DX(vis.loc), DY(vis.loc), "Blob", 3
        ELSE                         ;Else if object was NOT found...
            GCOLOR (vlun) 3          ;Select the color red
            GTRANS (vlun, 0)         ;Select pixel scaling
            GTYPE (vlun) 100, 100, "No object found!", 3
        END
        ; Detach (frees up the communications path)
        DETACH (vlun)
  100   IF (IOSTAT(vlun) < 0) THEN   ; Check for errors
            TYPE $ERROR(IOSTAT(vlun))
        END

.END
```

Additional Graphics Instructions

The following table lists the graphics instructions available in the eV+ programming language. For complete details on any instruction, click on the command name in the table, or refer to the keyword documentation available in the *eV+ Language Reference Guide.*

List of Graphics Instructions

Command	Action
GARC	Draw an arc or circle in a graphics window.
GCHAIN	Draw a chain of points.
GCLEAR	Clear an entire window to the background color.
GCLIP	Constrain the area of a window within which graphics are displayed.
GCOLOR	Set the foreground and background colors for subsequent graphics instructions.
GCOPY	Copy one area of a graphics window to another area in the window.
GFLOOD	Flood an area with foreground color.
GICON	Allows you to display icons on the screen. You can access the predefined Omron Adept icons or use your own icons.
GLINE	Draw a line.
GLINES	Draw multiple lines.
GLOGICAL	Set the drawing mode for the next graphics instruction. (Useful for erasing existing graphics and simulating the dragging of a graphic across the screen.)
GPOINT	Draw a single point.
GRECTANGLE	Draw a rectangle.
GSCAN	Draw a series of horizontal lines.
GSLIDE	Create a slide bar.
GTEXTURE	Develop a texture for subsequent graphics. Set subsequent graphics

Command	Action
	to transparent or opaque.
GTRANS	Define a transformation to apply to all subsequent G instructions.
GTYPE	Display a text string.

Programming the Omron Adept T20 Pendant

This version of eV+ uses the Omron Adept T20 pendant. For more information, see Programming the T20 Pendant and the *Adept T20 Pendant User's Guide*.

The following topics are described in this chapter:

Introduction

This section provides an overview of how to program the Omron Adept T20 pendant. You can refer to the *Adept T20 Pendant User's Guide* for information on installing and operating the pendant.

ATTACHing and DETACHing the Pendant

Before an application program can communicate with the pendant, the pendant must first be ATTACHed using the ATTACH instruction. The logical unit number (lun) for the pendant is 1. The following code readies the pendant for communication:

```
t20_lun = 1
ATTACH (t20_lun)
```

As with all other devices that are ATTACHed by a program, the pendant should be DETACHed when the program is finished with the pendant. The following instruction frees up the pendant:

```
DETACH (t20_lun)
```

When the pendant has been ATTACHed by an application program, the user can interact with the pendant without selecting manual mode.

As with all I/O devices, the IOSTAT function should be used to check for errors after each I/O operation.

Writing to the Pendant Display

Pendant Display

The pendant allows users to display a title and a message body, and to modify the labels for function keys F1 through F4. Any field may be an empty string (""). The message body can process HTML-tagged code.

Using PDNT.WRITE with the Pendant

The following instructions:

```
$p.title = "Operator Control"
$p.msg[0] = "Select Options from buttons below"
$p.f[1] = "Apps"
$p.f[2] = "Status"
$p.f[3] = ""
$p.f[4] = ""
PDNT.WRITE $p.title, $p.msg[], $p.f[1], $p.f[2], $p.f[3], $p.f[4]
```

Create the following user dialog :

Using PDNT.NOTIFY with the Pendant

The following instruction:

```
PDNT.NOTIFY "Waiting on Parts", "Please add parts to Feeder"
```

Creates the following user dialog :

Using PDNT.CLEAR

The PDNT.CLEAR instructions clears the pendant display, and returns to the Home 1 screen.

Detecting User Input

Input from the pendant can be received from a single key press from any of the keys that can be detected. Single-key presses can be monitored in three different modes:

- The keys can be monitored like keys on a normal keyboard.

- The keys can be monitored in toggle mode (on or off). The state of the key is changed each time the key is pressed.

- The keys can be monitored in level mode. The state of the key is considered ON only when the key is held down.

The PENDANT( ) function is used to detect key presses. The KEYMODE instruction is used to set the key behavior.

Detecting Pendant Key Presses

Individual pendant key presses are detected with the PENDANT( ) real-valued function. (The following figure provides a reference for the numbers of the keys on the T20 pendant.) This function returns the number of the first acceptable key press. The interpretation of a key press is determined by previous execution of the KEYMODE instruction. See the *eV+ Language Reference Guide* for complete details. The basic use of these two operations is described below.

T20 Pendant Key Map

Keyboard Mode

The default mode is Keyboard. If a PENDANT( ) function requests any keyboard input, the key number of the first Keyboard mode key pressed is returned. The following code detects the first function key pressed:

```
; Set the function keys to keyboard mode
        KEYMODE 1,4 = 0
; Wait for a key press from keys 1 - 4
        DO
                key = PENDANT(0)
        UNTIL key < 5
```

The arguments to the KEYMODE instruction indicate that pendant keys 1 through 4 are to be configured in Keyboard mode. The 0 argument to the PENDANT( ) function indicates that the key number of the first key pressed is to be returned.

Toggle Mode

To detect the state of a key in Toggle mode, the PENDANT( ) function must specify the key to be monitored.

When a key is configured as a toggle key, its state is maintained as ON (-1) or OFF (0). The state is toggled each time the key is pressed. The following code sets the F1 key to Toggle mode and waits until F1 is pressed:

```
; Set the F1 key to toggle
        KEYMODE 1 = 1
; Wait until the F1 key is pressed
        WHILE NOT PENDANT(1) DO
                RELEASE -1
        END
```

Level Mode

To detect the state of a key in Level mode, the PENDANT( ) function must specify the key to be monitored.

When a key has been configured as a level key, the state of the key is ON as long as the key is pressed. When the key is not pressed, its state is OFF. The following code will poll the OK button's state until it has been held down for an amount of time. If it is released prematurely, the counter is reset.

```
ATTACH(1) "PENDANT"
counter = 0
KEYMODE 18, 18 = 2

WHILE counter < 20 DO
        IF PENDANT(18) THEN
                counter = counter+1
        ELSE
                counter = 0
        END
```

```
        WAIT.EVENT , 0.1
    END
    DETACH (1)
```

Notes on Key Behaviors

On the T20 pendant, all Green keys and Select Robot will still have their normal functionality when in a Custom Window (through PDNT.WRITE). All other keys will have their primary functionality disabled. These keys are intended to be read by the eV+ program through the PENDANT instruction.

Monitoring the Pendant Speed Signal

The speed that is sent from the pendant has a value from 1 to 100 depending on the currently displayed speed on the Pendant. An argument of -2 to the PENDANT( ) function returns this value to eV+. The following example shows how to print the currently displayed Pendant speed to the monitor.

```
; Set the OK key to toggle
    KEYMODE 18 = 1
; Display speed value until the OK key is pressed
    DO
        TYPE PENDANT(-2)
        WAIT
    UNTIL PENDANT(18)
```

Reading the State of the Pendant

It is good programming practice to check the state of the pendant before ATTACHing to it. The instruction:

```
    cur.state = PENDANT(-3)
```

returns a value to be interpreted as follows:

1. Indicates that the pendant is in the Background state (not ATTACHed to an application program).

2. Indicates that an error is being displayed.

3. Indicates that the pendant is in the USER state (ATTACHed to an application program).

See the section Programming Example: Pendant Menu for a program example that checks the pendant state.

Programming Example: Pendant Menu

The following code implements a menu structure on the Omron Adept T20 pendant. The resulting screens are shown after the code sample.

```
.PROGRAM operatorcontrol()
; ABSTRACT:  This program creates and monitors a menu structure on the T20.
;
; INPUT PARAMS: None
;
; OUTPUT PARAMS: None
;
; GLOBAL VARS: None

        AUTO $p.title, $p.msg[15], $p.f[15], key, app.sel
        AUTO pendant.lun

        pendant.lun = 1

; Attach to the Pendant

        ATTACH (pendant.lun)

; Verify ATTACH was successful

        IF IOSTAT(pendant.lun) <> 1 THEN
            TYPE "Pendant is either busy or not connected"
            GOTO 100    ;ERROR
        END

; Menu elements

        $prog.list[0] = "Cookie1"
        $prog.list[1] = "Cookie2"
        $prog.list[2] = "Cookie3"
        $prog.list[3] = "Cookie4"
        $prog.list[4] = "Cookie5"
        prog.len = 5

; Initialize variables

        key = -1
        app.sel = 0
        $p.title = ""
        FOR i = 0 TO 10
            $p.msg[i] = ""
        END
        FOR i = 0 TO 4
            $p.f[i] = ""
        END

; Screen 1 - Operator Control

        $p.title = "Operator Control"
        $p.msg[0] = "Select Option from buttons below"
        $p.f[1] = "Apps"
        $p.f[2] = "Status"
        $p.f[3] = ""
        $p.f[4] = ""
```

```
; Display Custom Message

        PDNT.WRITE $p.title, $p.msg[], $p.f[1], $p.f[2], $p.f[3], $p.f[4]

        WHILE TRUE DO
            KEYMODE 1, 4 = 0     ;Set Keymode of F keys detect next keypress
            key = PENDANT(0)     ;Obtain Pressed key
            CASE key OF
              VALUE 1:              ;Apps pressed, continue to Screen 2
                GOTO 10
              VALUE 2:              ;Display Filler Notification
                ; This will be cleared on Pendant by pressing OK or Cancel
                PDNT.NOTIFY "Status", "No Program Running" ;Obtain info of
task 1 here
            END
        END

; Setup Screen 2

    10  $p.title = "List of Applications"
        $p.f[1] = "Run"
        $p.f[2] = "Up"
        $p.f[3] = "Down"
        $p.f[4] = "Done"
        FOR i = 0 TO 10
            $p.msg[i] = ""
        END
        FOR i = 0 TO prog.len-1
            IF i == app.sel THEN
                ; Underline the currently selected Application
                $p.msg[i] = $p.msg[i]+"<center><u>"+$prog.list[i]+"
</u></center>"
            ELSE
                $p.msg[i] = $p.msg[i]+"<center>"+$prog.list[i]+"</center>"
            END
        END

; Display Screen 2 - List of Applications

        PDNT.WRITE (prog.len) $p.title, $p.msg[], $p.f[1], $p.f[2], $p.f[3],
$p.f[4]
        KEYMODE 1, 28 = 0
        WHILE TRUE DO
            key = PENDANT(0)
            CASE key OF
              VALUE 1, 18:      ;F1 or OK
                CALLS $prog.list[app.sel]()
                GOTO 10 ;
              VALUE 2, 17:      ;F2 or Up arrow
                app.sel = (app.sel-1) MOD prog.len
                GOTO 10
              VALUE 3, 21:      ;F3 or Down arrow
                app.sel = (app.sel+1) MOD prog.len
```

```
            GOTO 10
          VALUE 4:              ;F4
            GOTO 90
        END
      END

; Finished normally, clear screen and detach

  90  PDNT.CLEAR
      DETACH (1)

  100  RETURN

.END
```


Screen 1

Screen 2

Conveyor Tracking

This chapter describes the Conveyor Tracking (moving-line) feature.

The ACE software provides a graphical interface for programming your Omron Adept motion (and vision) system. Further, the ACE Process Manager, which is included with the ACE software, allows you to build conveyor-tracking applications through a point-and-click interface. Therefore, Omron Adept strongly recommends that you use the ACE software for this functionality. For more details, see the chapter Process Control in the *ACE User's Guide*. Optionally, you can use eV+ to manually program a conveyor-tracking application, as described in this chapter.

The following sections contain installation and application instructions for using the conveyor-tracking feature in eV+. Before using this chapter, you should be familiar with eV+ and the basic operation of the robot.

Introduction to Conveyor Tracking

NOTE: The ACE software provides the following functionality through its graphical user interface. Therefore, Omron Adept strongly recommends that you use the ACE software.

This chapter describes the Conveyor Tracking (moving-line) feature. The moving-line feature allows the programs to specify locations that are automatically modified to compensate for the instantaneous position of a conveyor belt. Motion locations that are defined relative to a belt can be taught and played back while the belt is stationary or moving at arbitrarily varying speeds. Conveyor tracking is available only for systems that have the optional eV+ Extensions software.

For eV+ to determine the instantaneous position and speed of a belt, the belt must be equipped with a device to measure its position and speed. As part of the moving-line hardware option, Omron Adept provides an interface for coordinating two separate conveyor belts. Robot motions and locations can be specified relative to either belt.

There are no restrictions concerning the placement or orientation of a conveyor belt relative to the robot. In fact, belts that move uphill or downhill (or at an angle to the reference frame of the robot) can be treated as easily as those that move parallel to an axis of the robot reference frame. The only restriction regarding a belt is that its motion must follow a straight-line path in the region where the robot is to work.

The following sections contain installation and application instructions for using the moving-line feature. Before using this chapter, you should be familiar with eV+ and the basic operation of the robot.

Installation

To set up a conveyor belt for use with a robot controlled by the eV+ system:

1. Install all the hardware components and securely fasten them in place. The conveyor frame and robot base must be mounted rigidly so that no motion can occur between them.

2. Install the encoder on the conveyor.

3. Since any jitter of the encoder will be reflected as jitter in motions of the robot while tracking the belt, make sure the mechanical connection between the belt and the encoder operates smoothly. In particular, eliminate any backlash in gear-driven systems.

4. Wire the encoder to the robot controller. (See the Adept MV Controller User's Guide for location of the encoder ports.)

5. Start up the robot system controller in the normal manner.

6. Use the Belt Calibration group in the ACE Process Manager to calibrate the location of the conveyor belt relative to the robot. For details, see the section Belt Calibrations in the *ACE User's Guide*.

When these steps have been completed, the system is ready for use. The next section describes loading belt calibration.

Calibration

The position and orientation of the conveyor belt must be precisely known in order for the robot to track motion of the belt. Use the Belt Calibration group in the ACE Process Manager to calibrate the location of the conveyor belt relative to the robot. For details, see the section Belt Calibrations in the *ACE User's Guide*.

The DEFBELT and WINDOW program instructions must be executed before the associated belt is referenced in a eV+ program. For details, see Belt Variable Definitions on page 199. We suggest you include these instructions in an initialization section of your application program. Although these instructions need be executed only once, no harm is done if they are executed subsequently.

While the robot is moving relative to a belt (including motions to and from the belt), all motions must be of the straight-line type. Thus APPROS, DEPARTS, MOVES, and MOVEST can be used, but APPRO, DEPART, DRIVE, MOVE, and MOVET cannot. Motion relative to a belt is terminated when the robot moves to a location that is not defined relative to the belt variable or when a belt-window violation occurs.

Basic Programming Concepts

This section describes the basic concepts of the conveyor-tracking feature. First, the data used to describe the relationship of the conveyor belt to the robot is presented. Then a description is given of how belt-relative motion instructions are specified. Finally, a description is presented of how belt-relative locations are taught.

The eV+ operations associated with belt tracking are disabled when the BELT system switch is disabled. Thus, application programs that use those operations must be sure the BELT switch is enabled.

Belt Variables

The primary mechanism for specifying motions relative to a belt is a eV+ data type called a belt variable. By defining a belt variable, the program specifies the relationship between a specific belt encoder and the location and speed of a reference frame that maintains a fixed position and orientation relative to the belt. Alternatively, a belt variable can be thought of as a transformation (with a time-varying component) that defines the location of a reference frame fixed to a moving conveyor. As a convenience, more than one belt variable can be associated with the same physical belt and belt encoder. In this way, several work stations can be easily referenced on the same belt.

Like other variable names in eV+, the names of belt variables are assigned by the programmer. Each name must start with a letter and can contain only letters, numbers, periods, and underline characters. (Letters used in variable names can be entered in either lowercase or uppercase. eV+ always displays variable names in lowercase.)

To differentiate belt variables from other data types, the name of a belt variable must be preceded by a percent sign (%). As with all other eV+ data types, arrays of belt variables are permitted. Hence the following are all valid belt-variable names:

```
%pallet.on.belt      %base.plate      %belt[1]
```

The DEFBELT instruction must be used to define belt variables (see Conveyor-Tracking Programming on page 199). Thus, the following are **not** valid operations:

```
SET %new_belt = %old_belt or HERE %belt[1]
```

Compared to other eV+ data types, the belt variable is rather complex in that it contains several different types of information. Briefly, a belt variable contains the following information:

1. The nominal transformation for the belt. This defines the position and direction of travel of the belt and its approximate center.

2. The number of the encoder used for reading the instantaneous location of the belt (from 1 to 6).

3. The belt encoder scaling factor, which is used for converting encoder counts to

millimeters of belt travel.

4. An encoder offset, which is used to adjust the origin of the belt frame of reference.

5. Window parameters, which define the working range of the robot along the belt.

These components of belt variables are described in detail in the following sections.

Unlike other eV+ data types, belt variables cannot be stored in a disk file for later loading. However, the location and real-valued data used to define a belt variable can be stored and loaded in the normal ways. After the data is loaded from disk, DEFBELT and WINDOW instructions must be executed to define the belt variable. For details, see Belt Variable Definitions on page 199. (The file LOADBELT.V2 on the Utility Disk contains a subroutine that will read belt data from a disk file and execute the appropriate DEFBELT and WINDOW instructions.)

Nominal Belt Transformation

The position, orientation, and direction of motion of a belt are defined by a transformation called the nominal belt transformation. This transformation defines a reference frame aligned with the belt as follows: its X-Y plane coincides with the plane of the belt, its X axis is parallel to the direction of belt motion, and its origin is located at a point (fixed in space) chosen by the user.

Since the direction of the X axis of the nominal belt transformation is taken to be the direction along which the belt moves, this component of the transformation must be determined with great care. Furthermore, while the point defined by this transformation (the origin of the frame) can be selected arbitrarily, it normally should be approximately at the middle of the robot's working range on the belt. This transformation is usually defined using the FRAME location-valued function with recorded robot locations on the belt. (The easiest way to define the nominal belt transformation is with the conveyor belt calibration program provided by Omron Adept.)

The instantaneous location described by the belt variable will almost always be different from that specified by the nominal transformation. However, since the belt is constrained to move in a straight line in the working area, the instantaneous orientation of a belt variable is constant and equal to that defined by the nominal belt transformation.

To determine the instantaneous location defined by a belt variable, the eV+ system performs a computation that is equivalent to multiplying a unit vector in the X direction of the nominal transformation by a distance (which is a function of the belt encoder reading) and adding the result to the position vector of the nominal belt transformation. Symbolically, this can be represented as

```
instantaneous_XYZ =
nominal_XYZ + (belt_distance *  X_direction_of_nominal_transform)
```

where

```
belt_distance =
(encoder_count - encoder_offset) * encoder_scaling_factor
```

The encoder variables contained in this final equation are described in later sections.

The Belt Encoder

Six belt encoders are supported by the conveyor tracking feature.

Each belt encoder generates pulses that indicate both the distance that the belt has moved and the direction of travel. The pulses are counted by the belt interface, and the count is stored as a signed 24-bit number. Therefore, the value of an encoder counter can range from 2^{23} -1 (8,388,607) to -2 ^{23} (-8,388,608). For example, if a single count of the encoder corresponds to 0.02 millimeters (0.00008 inch) of belt motion, then the full range of the counter would represent motion of the belt from approximately -167 meters (-550 feet) to +167 meters (+550 feet).

After a counter reaches its maximum positive or negative value, its value will roll over to the maximum negative or positive value, respectively. This means that if the encoder value is increasing and a rollover occurs, the sequence of encoder counter values will be ... ; 8,388,606; 8,388,607; -8,388,608; -8,388,607; ... As long as the distance between the workspace of the robot and the nominal transformation of the belt is within the distance that can be represented by the maximum encoder value, eV+ application programs normally do not have to take into account the fact that the counter will periodically roll over. The belt_ distance equation described above is based upon a relative encoder value:

```
encoder_count - encoder_offset
```

and eV+ automatically adjusts this calculation for any belt rollover that may occur.

Care must be exercised, however, if an application processes encoder values in any way. For example, a program may save encoder values associated with individual parts on the conveyor, and then later use the values to determine which parts should be processed by the robot. In such situations the application program may need to consider the possibility of rollover of the encoder value.

The Encoder Scaling Factor

For any given conveyor/encoder installation, the encoder scaling factor is a constant number that represents the amount the encoder counter changes during a change in belt position. The units of the scaling factor are millimeters/count.

This factor can be determined either directly from the details of the mechanical coupling of the encoder to the belt or experimentally by reading the encoder as the belt is moved. The belt calibration program supports either method of determining the encoder scaling factor.

If the encoder counter decreases as the belt moves in its normal direction of travel, the scaling factor will have a negative value.

The Encoder Offset

The last encoder value needed for proper operation of the conveyor-tracking system is the belt encoder offset. The belt encoder offset is used by eV+ to establish the instantaneous location of the belt reference frame relative to its nominal location.

In particular, if the belt offset is set equal to the current belt encoder reading, the instantaneous belt transformation will be equal to the nominal transformation. The belt encoder offset can be used, in effect, to zero the encoder reading, or to set it to a particular value whenever necessary. Unlike the encoder scaling factor, which is constant for any given conveyor/encoder setup, the value of the belt encoder offset is variable and will usually be changed often.

Normally, the instantaneous location of the reference frame will be established using external input from a sensory device such as a photocell. The DEVICE real-valued function also returns latched or unlatched encoder values for use with SETBELT.

The encoder offset is set with the SETBELT program instruction, described in Belt Variable Definitions on page 199.

The Belt Window

The belt window controls the region of the belt in which the robot is to work. The figure Conveyor Terms illustrates the terms used here. A window is a segment of the belt bounded by two planes that are perpendicular to the direction of travel of the belt. When defining the window, ensure that the robot can reach all conveyor locations within the belt window. This is especially important for revolute (i.e., non-Cartesian) robots.

> **NOTE:** The window has limits only in the direction along the belt.

Within eV+, a belt window is defined by two transformations with a WINDOW program instruction. The window boundaries are computed by eV+ as planes that are perpendicular to the direction of travel of the belt and that pass through the positions defined by the transformations.

If the robot attempts to move to a belt-relative location that has not yet come within the window (is upstream of the window), the robot can be instructed either to pause until it can accomplish the motion or immediately generate a program error. If a destination moves out of the window (is downstream of the window), it is flagged as an error condition and the application program can specify what action is to be taken. (See the description of the BELT.MODE system parameter in eV+ Language Reference Guide.)

If the normal error testing options are selected, whenever the eV+ system is planning a robot motion to a belt-relative location and the destination is outside the belt window but upstream, the system automatically delays motion planning until the destination is within the window. However, if an application program attempts to perform a motion to a belt-relative destination that is out of the window at planning time (or is predicted to be out by the time the destination would be reached) and this destination is downstream, a window-

violation condition exists. Also, if during the execution of a belt-relative motion or while the robot is tracking the belt, the destination moves outside the belt window for any reason, a window violation occurs. Depending upon the details of the application program, the program either prints an error message and halts execution or branches to a specified subroutine when a window violation occurs.

In order to provide flexibility with regard to the operation of the window-testing mechanism, several modifications to the normal algorithms can be selected by modifying the value of the BELT.MODE system parameter.

To assist in teaching the belt window, the conveyor belt calibration program contains routines that lead the operator through definition of the required bounding transformations.
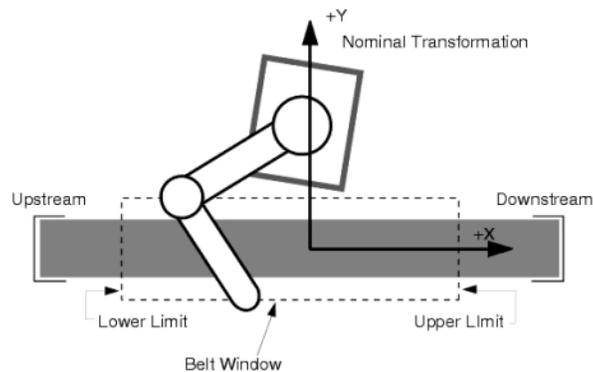
Conveyor Terms

Belt-Relative Motion Instructions

To define a robot motion relative to a conveyor belt, or to define a relative transformation with respect to the instantaneous location of a moving frame of reference, a belt variable can be used in place of a regular transformation in a compound transformation. For example, the instruction

```
MOVES %belt:loc_1
```

directs the robot to perform a straight-line motion to location loc_1, which is specified relative to the location defined by the belt variable %belt. If a belt variable is specified, it must be the first (that is, leftmost) element in a compound transformation. Only one belt variable can appear in any compound transformation.

Motions relative to a belt can be only of the straight-line type. Attempting a joint-interpolated motion relative to a belt causes an error and halts execution of the application program. Except for these restrictions, motion statements that are defined relative to a belt are treated just like any other motion statement. In particular, continuous-path motions relative to belts are permitted.

Once the robot has been moved to a destination that is defined relative to a belt, the robot tool will continue to track the belt until it is directed to a location that is not relative to the belt. For example, the following series of instructions would move the tool to a location relative to a belt, open the hand, track the belt for two seconds, close the hand, and finally move off the belt to a fixed location.

```
MOVES %belt[1]:location3
OPENI
DELAY 2.00
CLOSEI
MOVES fixed.location
```

If this example did not have the second MOVES statement, the robot would continue to track the belt until a belt window violation occurred.

As with motions defined relative to a belt, motions that move the tool off a belt (that is, to a fixed location) must be of the straight-line type.

Motion Termination

When moving the robot relative to a belt, special attention must be paid to the conditions used to determine when a motion is completed. At the conclusion of a continuous-path motion eV+ normally waits until all the joints of the manipulator have achieved their final destinations to within a tight error tolerance before proceeding to the next instruction. In the case of motions relative to a belt, the destination is constantly changing and, depending upon the magnitude and variability of the belt speed, the robot may not always be able to achieve final positions with the default error tolerance.

Therefore, if a motion does not successfully complete (that is, it is aborted due to a Time-out nulling error), or if it takes an excessive amount of time to complete, the error tolerance for the motion should be increased by preceding the motion instruction with a COARSE instruction. In extreme situations it may even be necessary to entirely disable checking of the final error tolerance. This can be done by specifying NONULL before the start of the motion.

Defining Belt-Relative Locations

In order to define locations relative to a belt, belt-relative compound transformations can be used as parameters to all the standard eV+ teaching aids. For example, all the following commands define a location loc_1 relative to the current belt location:[1]

```
HERE %belt:loc_1
```

In each of these cases, the instantaneous location corresponding to %belt would be determined (based upon the reading of the belt encoder associated with %belt); loc_1 would be set equal to the difference between the current tool location and the instantaneous location defined by %belt.

While a belt variable can be used as the first (leftmost) element of a compound transformation to define a transformation value, a belt variable cannot appear by itself. For example, LISTL will not display a belt variable directly. To view the value of a belt variable, enter the command:

```
LISTL %belt_variable:NULL
```

---

[1]Before defining a location relative to a belt, you must make sure the belt encoder offset is set properly. That usually involves issuing a monitor command in the form:

```
DO SETBELT %belt = BELT(%belt)
```

Conveyor-Tracking Programming

This section describes how to access the conveyor-tracking capabilities within eV+. A functional overview is presented that summarizes the extensions to eV+ for Conveyor Tracking. All the eV+ conveyor-tracking keywords are described in detail in the *eV+ Language Reference Guide*.

The conveyor-tracking extensions to eV+ include:

- Instructions and functions (there are no monitor commands)
- System switch
- System parameters

Instructions and Functions

This section summarizes the eV+ instructions and functions dedicated to conveyor-tracking processing. The belt-related functions return real values.

Belt Variable Definitions

The following keywords are used to define the parameters of belt variables. Some parameters are typically set once, based upon information derived from the belt calibration procedure. Other parameters are changed dynamically as the application program is executing.

DEFBELT   Program instruction that creates a belt variable and defines its static characteristics: nominal transformation, encoder number, and encoder scaling factor.

SETBELT   Program instruction to set the encoder offset of a belt variable. This defines the instantaneous belt location relative to that of the nominal belt transformation.

WINDOW   Program instruction for establishing the belt window boundaries and specifying a window-violation error subroutine.

Encoder Position and Velocity Information

The following function is used to read information concerning the encoder associated with a belt variable.

BELT   Real-valued function that returns the instantaneous encoder counter value or the rate of change of the encoder counter value.

Window Testing

The following function allows an application program to incorporate its own specialized working-region strategy, independent of the strategy provided as an integral part of the eV+ conveyor tracking system.

WINDOW   Real-valued function that indicates where a belt-relative location is (or will be at some future time) relative to a belt window.

Status Information

The following function indicates the current operating status of the conveyor-tracking software.

BSTATUS   Real-valued function that returns bit flags indicating the status of the conveyor-tracking software.

System Switch

The switch BELT enables/disables the operation of the conveyor-tracking software. (See the description of ENABLE, DISABLE, and SWITCH for details on setting and displaying the value of BELT.)

BELT   This switch must be enabled before any conveyor tracking processing begins.

System Parameters

The following parameter selects alternative modes of operation of the belt window testing routines. See the description of PARAMETER for details on setting and displaying the parameter values.

BELT.MODE   Bit flags for selecting special belt window testing modes of operation.

Sample Programs

The following program is an example of a robot task working from a moving conveyor belt. The task consists of the following steps:

1. Wait for a signal that a part is present.

2. Pick up the part.

3. Place the part at a new location on the belt.

4. Return to a rest location to wait for the next part.

CAUTION: These programs are meant only to illustrate programming techniques useful in typical applications. Moving-line programs are hardware dependent because of the belt parameters, so care must be exercised if you attempt to use these programs.

```
; *** PROGRAM TO RELOCATE PART ON CONVEYOR ***
; Set up belt parameters

        ENABLE BELT
        PARAMETER BELT.MODE = 0
        belt.scale = 0.03067        ;Encoder scale factor

; Define belt twice, for two stations

        DEFBELT %b1 = belt, 1, 32, belt.scale
        WINDOW %b1 = window.1, window.2, window.error
        DEFBELT %b2 = belt, 2, 32, belt.scale
        WINDOW %b2 = window.1, window.2, window.error

        WHILE TRUE DO               ;Loop indefinitely
            WAIT part.ready         ;Wait for signal that part present
            bx = BELT(%b1)          ;Read present belt position
            SETBELT %b1 = bx        ;Set encoder offset for pick-up...
            SETBELT %b2 = bx        ;... and drop-off stations
            APPROS %b1:p1, 50       ;Move to the part and pick it up
            MOVES %b1:p1
            CLOSEI
            DEPARTS 50
            APPROS %b2:p2, 50       ;Carry part to drop-off location
            MOVES %b2:p2
            OPENI
            DEPARTS 50
            MOVES wait.location     ;Return to rest location
        END                         ;Wait for the next part
; *** End of program ***
```

The WINDOW instruction in the above program indicates that whenever a window violation occurs, a subroutine named window.error is to be executed. The following is an example of what such a routine might contain.

```
; *** WINDOW VIOLATION ROUTINE ***
        TYPE /B, /C1, "** WINDOW ERROR OCCURRED **", /C1

; Find out which end of window was violated

        IF DISTANCE(HERE,window.1) < DISTANCE(HERE,window.2) THEN

; Error occurred at window.2

        TYPE "Part moved downstream out of reach"

 ;...(Respond to downstream window error) .

        ELSE                        ; Error occurred at window.1
            TYPE "Part moved upstream out of reach"
  ;...(Respond to upstream window error) .
        END

        MOVES wait.location       ;Move robot to rest location

; Use digital output signals to sound alarm and stop belt

        SIGNAL alarm, stop.belt
        HALT                      ;Halt program execution
```

Multiprocessor Systems

NOTE: This feature is no longer supported.

The ACE software allows you to control multiple instances of eV+ (each instance runs on a separate controller) and communicate information between each instance. For more details, see the *ACE User's Guide*.

Example eV+ Programs

The following topics are described in this chapter:

Introduction

This chapter contains a sampling of eV+ programs. This chapter contains a sample eV+ program. The first program is presented twice: once in its entirety exactly as it is displayed by eV+ and a second time with a line-by-line explanation.

The program keywords are detailed in the *eV+ Language Reference Guide*.

> **NOTE:** The programs in this manual are not necessarily complete. In most cases further refinements could be added to improve the programs. For example, the programs could be made more tolerant of unusual events such as error conditions.

Pick and Place

This program demonstrates a simple pick-and-place application. The robot picks up parts at one location and places them at another.

Features Introduced

- Program initialization
- Variable assignment
- System parameter modification
- FOR loop
- Motion instructions
- Hand control
- Terminal output

Program Listing

```
.PROGRAM move.parts()

; ABSTRACT: Pick up parts at location pick and put them down at place

        parts = 100                  ;Number of parts to be processed

        height1 = 25.4               ;Approach/depart height at "pick"

        height2 = 50.8               ;Approach/depart height at "place"

        PARAMETER HAND.TIME = 0.16   ;Set up for slow hand

        OPEN                         ;Make sure the hand is open
        RIGHTY                       ;Make sure configuration is correct
        MOVE start                   ;Move to safe starting location

        FOR i = 1 TO parts           ;Process the parts

           APPRO pick, height1       ;Go toward the pick-up
           MOVES pick                ;Move to the part
           CLOSEI                    ;Close the hand
           DEPARTS height1           ;Back away

           APPRO place, height2      ;Go toward the put-down
           MOVES place               ;Move to the destination
           OPENI                     ;Release the part
           DEPARTS height2           ;Back away

        END                          ;Loop for next part

        TYPE "All done. ", /I0, parts, " parts processed"
```

```
        RETURN
        ;End of the program
 .END
```

Detailed Description

This program has five sections: formal introduction, initialization of variables, initialization of the robot location, performance of the desired motion sequence, and notice to the operator of completion of the task. Each of these sections is described in detail below.

The first line of every program must have the form of the line below. It is a good practice to follow that line with a brief description of the purpose of the program. If there are any special requirements for use of the program, they should be included as well.

```
    .PROGRAM move.parts()
```

This line identifies the program to the eV+ system. In this case we see that the name of the program is move.parts.

```
    ; ABSTRACT: Pick up parts at location "pick" and put them down at
    "place"
```

This is a very brief description of the operation performed by the program. (Most programs requires a more extensive summary.)

Use variables to represent constants for two reasons: Using a variable name throughout a program makes the program easier to understand, and only one program line must be modified if the value of the constant is changed.

```
    parts = 100
```

Tell the program how many parts to process during a production run. In this case, 100 parts are processed.

```
    height1 = 25.4
```

Height1 controls the height of the robot path when approaching and departing from the location where the parts are to be picked up. Here it is set to 25.4 millimeters (that is, 1 inch).

```
    height2 = 50.8
```

Similar to height1, height2 sets the height of the robot path when approaching and departing from the put-down location. It is set to 50.8 millimeters (2 inches).

```
    PARAMETER HAND.TIME 0.16
```

Set the system parameter HAND.TIME so that sufficient time is allowed to actuate the robot hand.

This setting causes OPENI and CLOSEI instructions to delay program execution for 160 milliseconds while the hand is actuated.

Initially, you should also make sure that the robot has the desired hand opening, is at a safe starting location, and that SCARA robots have the desired configuration.

```
RIGHTY
```

Make sure the robot has a right-handed configuration (with the elbow of the robot to the right side of the workspace). This is important if there are obstructions in the workspace that must be avoided.

This instruction causes the robot to assume the requested configuration during its next motion.

```
OPEN
```

Make sure the hand is initially open. This instruction is executed during the next robot motion, rather than immediately as is done by the OPENI instruction.

```
MOVE start
```

Move to a safe starting location. Because of the preceding two instructions, the robot assumes a right-handed configuration with the hand open.

The location **start** must be defined before the program is executed. That can be done, for example, with the HERE command. The location must be chosen such that the robot can move from it to the pick-up location for the parts without hitting anything.

After initialization, the following program section performs the application tasks.

```
FOR i = 1 TO parts
```

Start a program loop. The following instructions (down to the END) will be executed **parts** times. After the last time the loop is executed, program execution continues with the TYPE instruction following the END below.

```
APPRO pick, height1
```

Move the robot to a location that is **height1** millimeters above the location pick.

The APPROS instruction is not used here because its straight-line motion would be slower than the motion commanded by APPRO.

```
MOVES pick
```

Move the robot to the pick-up location **pick**, which must have been defined previously.

The straight-line motion commanded by MOVES assures that the hand does not hit the part during the motion. A MOVE instruction could be used here if there is sufficient clearance between the hand and the part to allow for a nonstraight-line path.

```
CLOSEI
```

Close the hand. To assure that the part is grasped before the robot moves away, the I form of the CLOSE instruction is used-program execution will be suspended while the hand is closing.

```
        DEPARTS height1
```

Now that the robot is grasping the part, we can back away from the part holder. This instruction moves the hand back **height1** millimeters, following a straight-line path to make sure the part does not hit its holder.

```
        APPRO place, height2
        MOVES place
        OPENI
        DEPARTS height2
```

Similar to the above motion sequence, these instructions cause the part to be moved to the put-down location and released.

```
        END
```

This marks the end of the FOR loop. When this instruction is executed, control is transferred back to the FOR instruction for the next cycle through the loop (unless the loop count specified by parts is exceeded).

The final section of the program simply displays a message on the system terminal and terminates execution.

```
        TYPE "All done. ", /I0, parts, " pieces processed."
```

The above instruction outputs the message:

```
        All done. 100 pieces processed.
```

(The /I0 format specification in the instruction causes the value of parts to be output as an integer value without a decimal point.)

```
        RETURN
```

Although not absolutely necessary for proper execution of the program, it is good programming practice to include a RETURN (or STOP) instruction at the end of every program.

```
        .END
```

This line is automatically included by the eV+ editor to mark the program's end.

Menu Program

This program displays a menu of operations from which an operator can choose.

Features Introduced

- Subroutines
- Local variables
- Terminal interaction with operator
- String variables
- WHILE and CASE structures

Program Listing

```
.PROGRAM sub.menu()

 ; ABSTRACT: This program provides the operator with a menu of
 ;       operation selections on the system terminal. After accepting
 ;       input from the keyboard, the program executes the desired
 ;       operation. In this case, the menu items include execution of
 ;       the pick and place program, teaching locations for the pick
 ;       and place program, and returning to a main menu.
 ;
 ; SIDE EFFECTS: The pick and place program may be executed, and
 ;       locations may be defined.

        AUTO choice, quit, $answer

        quit = FALSE

        DO

            TYPE /C2, "PICK AND PLACE OPERATIONAL MENU"
            TYPE /C1, " 1 => Initiate pick and place"
            TYPE /C1, " 2 => Teach locations"
            TYPE /C1, " 3 => Return to previous menu", /C1

            PROMPT "Enter selection and press RETURN: ", $answer

            choice = VAL($answer)        ;Convert string to number

            CASE choice OF               ;Process menu request...
              VALUE 1:                   ;...selection 1
                TYPE /C2, "Initiating Operation..."
                CALL move.parts()
              VALUE 2:                   ;...selection 2
                CALL teach()
              VALUE 3:                   ;...selection 3
                quit = TRUE
              ANY                        ;...any other selection
```

```
            TYPE /B, /C1, "** Invalid input **"
      END                            ;End of CASE structure
      UNTIL quit                     ;End of DO structure
.END
```

External Encoder Device

The following topics are described in this chapter:

Introduction

The external-encoder inputs on the system controller are normally used for conveyor belt tracking with a robot. However, these inputs can also be used for other sensing applications. In such applications, the DEVICE real-valued function and SETDEVICE program instruction allow the external encoders to be accessed in a more flexible manner than the belt-oriented instructions and functions.

This appendix describes the use of the DEVICE real-valued function and the SETDEVICEprogram instruction to access the external encoder device.

In general, SETDEVICE allows a scale factor, offset, and limits to be specified for a specified external encoder unit. The DEVICE real-valued function returns error status, position, or velocity information for the specified encoder.

Accessing the external encoders via DEVICE and SETDEVICE is independent of any belt-tracking commands or instructions. Setting belt parameters with SETBELT and setting encoder parameters with SETDEVICE have no effect on each other. The only exceptions are the SETDEVICE initialize command and reset command, which reset all errors for the specified external encoder, including any belt-related errors.

> **NOTE:** See the *eV+ Language Reference Guide*. for complete information on the DEVICE real-valued function and the SETDEVICE program instruction.

Parameters

The external encoder **device type** is 0. This means that the type parameter in all DEVICE or SETDEVICE instructions that reference the external encoders must have a value of zero.

The standard controller allows two external encoder units. These **units** are numbered 0 and 1. All DEVICE functions and SETDEVICE instructions that reference the external encoders must specify one of these unit numbers for the unit parameter.

Device Setup

The SETDEVICE program instruction allows the external encoders to be initialized and various parameters to be set up. The action taken by the SETDEVICE instruction depends upon the value of the command parameter.

The syntax of the SETDEVICE instruction is

```
SETDEVICE (0, unit, error, command) p1, p2
```

The following table describes the valid commands.

Command Parameter Values

Command	Description
0	Initialize Device This command sets all scale factors, offsets, and limits to their default values, as follows: offset = 0; scale factor = 1; no limit checking. This command also resets any errors for the specified device.This command should be issued before any other commands for a particular unit and before using the DEVICE real-valued function for the unit.
1	Reset Device This command clears any errors associated with this encoder unit. It does not affect the scale factor, offset, or limits.
8	Set Scale Factor This command sets the position and velocity scale factor for this encoder unit to the value of parameter p1. The units are millimeters per encoder count. The scale factor must be set before setting the offset or limits. If the scale factor is changed, the offset and limit values will need to be updated.
9	Set Position Offset This command sets the position offset for this encoder unit to the value of parameter p1. The units are millimeters. The scale factor must be set before setting the offset.
10	Set Position Limits This command sets the position limits for the encoder unit to the values of optional parameters p1 and p2, which are the lower and upper limits, respectively. If a parameter is omitted, no checking is performed for that limit. The units are millimeters. The scale factor must be set **before** setting the limits.

Reading Device Data

The DEVICE real-valued function returns information about the encoder error status, position, and velocity. The scale factor, offset, and limits defined by the SETDEVICE instruction affect the velocity and position values returned.

The syntax for this function is

```
DEVICE(0, unit, error, select)
```

The value returned depends upon the value of the select parameter, as described in the following table.

Select Parameter Values

select	Description
0	Read Hardware Status The error status of the encoder unit is returned as a 24-bit value. The valid error bits for this device are listed below. The corresponding error listed is the one eV+ would report if the error occurred while tracking a belt encoder. **Bit # Bit Mask Corresponding Error Message and Code** 19 ^H040000 *Lost encoder sync* (-1012) 20 ^H080000 *Encoder quadrature error* (-1013) 21 ^H100000 *No zero index* (-1011) Only bit #20, for encoder quadrature error, is detected by the error parameter of the DEVICE function to generate an error.
1	Read Position The current position of the encoder (in millimeters) is returned, subject to the scale factor, offset, and limits defined by the SETDEVICE instruction. The value returned is computed by: position = scale*(encoder-offset) position = MAX(position, lower_limit) position = MIN(position, upper_limit)
2	Read Velocity The current value of the encoder velocity (in millimeters per second) is returned, subject to the scale factor defined by the SETDEVICE instruction. The value returned is computed by: velocity = scale*encoder_velocity

select	Description
3	Read Predicted Position The predicted position of the encoder (in millimeters) is returned. The position is predicted 32 milliseconds in the future, based upon the current position and velocity. The value is scaled the same as the current position described above.
4	Read Latched Position The position or the encoder (in millimeters) when the last external trigger occurred is returned. The LATCHED real-valued function may be used to determined when an external trigger has occurred and a valid position has been recorded.

Character Sets

The tables ASCII Control Values and Omron Adept Character Set list the standard Omron Adept character set. Values 0 to 127 (decimal) are the standard ASCII character set. Characters 1 to 31 are the common set of special and line-drawing characters. Characters 0 and 127 to 141 are Omron Adept additions to the standard sets. Characters 32 to 255 (excluding 127 through 141) are the ISO standard 8859-1 character set. Characters 145 to 159 are overstrike characters (see the OVERSTRIKE attribute to the /TERMINAL argument for the FSET instruction in the *eV+ Language Reference Guide*). Values 1 to 31 are also given special meaning in the extended Omron Adept character set when they are output to a graphics window with the GTYPE instruction.

> **NOTE:** The full character set is defined for font #1 only. Fonts #2 (medium font), #3 (large font), and #4 (small font) have defined characters for ASCII values 0 and 32 - 127. Fonts #5 and #6 have standard English characters for ASCII values 0 and 32 - 135 while ASCII 136 - 235 are Katakana and Hiragana characters. Font #5 is standard size and font #6 contains large characters. The last column in Omron Adept Character Set shows the Katakana and Hiragana characters. The Katakana characters are at ASCII 161 - 223. The Hiragana characters are at ASCII 136 - 159 and 224 - 255.
>
> The character sets listed in ASCII Control Values and Omron Adept Character The sets are for use with graphics-based systems only.

Characters with values 0 to 31 and 127 (decimal) have the control meanings listed in the following table when output to a serial line, an ASCII terminal, or the monitor window (with TYPE, PROMPT, or WRITE instructions). In files exported to other text editors or transmitted across serial lines, characters 0 to 31 are generally interpreted as having the specified control meaning. The symbols shown for characters 0 to 31 and 127 in the table Omron Adept Character Set can be displayed only with the GTYPE instruction.

Characters in the extended Omron Adept character set can be output using the $CHR function. For example:

```
TYPE $CHR(229)
```

outputs the character å to the monitor window. The instruction:

```
GTYPE (glun) 50, 50, $CHR(229)
```

outputs the same character to the window open on logical unit glun.

ASCII Control Values

Charac-ter	Deci-mal Value	He-x. Val-ue	Meaning of Control Character
NUL	000	00	Null
SOH	001	01	Start of heading
STX	002	02	Start of text
ETX	003	03	End of text
EOT	004	04	End of transmission
ENQ	005	05	Enquiry
ACK	006	06	Acknowledg-ment
BEL	007	07	Bell
BS	008	08	Backspace
HT	009	09	Horizontal tab
LF	010	0A	Line feed
VT	011	0B	Vertical tab
FF	012	0C	Form feed
CR	013	0D	Carriage return
SO	014	0E	Shift out
SI	015	0F	Shift in

Charac-ter	Deci-mal Value	He-x. Val-ue	Meaning of Control Character
DLE	016	10	Data link escape
DC1	017	11	Direct control 1
DC2	018	12	Direct control 2
DC3	019	13	Direct control 3
DC4	020	14	Direct control 4
NAK	021	15	Negative acknowledge
SYN	022	16	Synchronous idle
ETB	023	17	End of transmission block
CAN	024	18	Cancel
EM	025	19	End of medium
SUB	026	1A	Substitute
ESC	027	1B	Escape
FS	028	1C	File separator
GS	029	1D	Group separator

Charac-ter	Deci-mal Value	He-x. Val-ue	Meaning of Control Character
RS	030	1E	Record separator
US	031	1F	Unit separator
DEL	127	7F	Delete

Omron Adept Character Set

De-c. Val-ue	He-x. Val-ue	Descripti-on	Fo-nt 1	Fonts 2, 3, 4, 5, & 6
000	00	cell outline	☐	☐
001	01	diamond	u	not defined
002	02	checkerbo-ard	▪	not defined
003	03	HT (Horizont-al Tab)	$^{H}_{\ T}$	not defined
004	04	FF (Form Feed)	$^{F}_{\ F}$	not defined
005	05	CR (Carriage Return)	$^{C}_{\ R}$	not defined
006	06	LF (Line Feed)	$^{L}_{\ F}$	not defined

Dec. Value	Hex. Value	Description	Font 1	Fonts 2, 3, 4, 5, & 6
007	07	degree symbol	o	not defined
008	08	plus/minus	±	not defined
009	09	NL (New line)	^N_L	not defined
010	0A	VT (Vertical Tab)	^V_T	not defined
011	0B	lower right corner	⌋	not defined
012	0C	upper right corner	⌐	not defined
013	0D	upper left corner	⌈	not defined
014	0E	lower left corner	⌊	not defined
015	0F	intersection	+	not defined
016	10	scan line 3	-	not defined
017	11	scan line 6	-	not defined
018	12	scan line 9	-	not defined

De-c. Val-ue	He-x. Val-ue	Descripti-on	Fo-nt 1	Fonts 2, 3, 4, 5, & 6
019	13	scan line 12	-	not defined
020	14	scan line 15	-	not defined
021	15	left T-bar	├	not defined
022	16	right T-bar	┤	not defined
023	17	bottom T-bar	⊥	not defined
024	18	top T-bar	┬	not defined
025	19	vertical bar	|	not defined
026	1A	less than or equal to	≤	not defined
027	1B	greater than or equal to	≥	not defined
028	1C	pi (lowercas-e)	π	not defined
029	1D	not equal to	≠	not defined
030	1E	sterling	£	not defined

De-c. Val-ue	He-x. Val-ue	Descripti-on	Fo-nt 1	Fonts 2, 3, 4, 5, & 6
031	1F	centered dot	·	not defined
032	20	space		not defined
033	21	exclamati-on	!	!
034	22	double quote	"	"
035	23	pound	#	#
036	24	dollar sign	$	$
037	25	percent	%	%
038	26	ampersan-d	&	&
039	27	single quote	'	'
040	28	open paren	(	(
041	29	close paren	)	)
042	2A	asterisk	*	*
043	2B	plus	+	+
044	2C	comma	,	,
045	2D	hyphen	-	-
046	2E	period	.	.

De-c. Val-ue	He-x. Val-ue	Descripti-on	Fo-nt 1	Fonts 2, 3, 4, 5, & 6
047	2F	slash	/	/
048	30	zero	0	0
049	31	one	1	1
050	32	two	2	2
051	33	three	3	3
052	34	four	4	4
053	35	five	5	5
054	36	six	6	6
055	37	seven	7	7
056	38	eight	8	8
057	39	nine	9	9
058	3A	colon	:	:
059	3B	semicolon	;	;
060	3C	less than	<	<
061	3D	equal to	=	=
062	3E	greater than	>	>
063	3F	question	?	?
064	40	at	@	@
065	41	A	A	A

De-c. Val-ue	He-x. Val-ue	Descripti-on	Fo-nt 1	Fonts 2, 3, 4, 5, & 6
066	42	B	B	B
067	43	C	C	C
068	44	D	D	D
069	45	E	E	E
070	46	F	F	F
071	47	G	G	G
072	48	H	H	H
073	49	I	I	I
074	4A	J	J	J
075	4B	K	K	K
076	4C	L	L	L
077	4D	M	M	M
078	4E	N	N	N
079	4F	O	O	O
080	50	P	P	P
081	51	Q	Q	Q
082	52	R	R	R
083	53	S	S	S
084	54	T	T	T

Dec. Value	Hex. Value	Description	Font 1	Fonts 2, 3, 4, 5, & 6
085	55	U	U	U
086	56	V	V	V
087	57	W	W	W
088	58	X	X	X
089	59	Y	Y	Y
090	5A	Z	Z	Z
091	5B	left bracket	[	[
092	5C	back slash	\	\
093	5D	right bracket	]	]
094	5E	circumflex (caret)	^	^
095	5F	underscore	_	_
096	60	grave accent		
097	61	a	a	a
098	62	b	b	b
099	63	c	c	c
100	64	d	d	d
101	65	e	e	e

De-c. Val-ue	He-x. Val-ue	Descripti-on	Fo-nt 1	Fonts 2, 3, 4, 5, & 6
102	66	f	f	f
103	67	g	g	g
104	68	h	h	h
105	69	i	i	i
106	6A	j	j	j
107	6B	k	k	k
108	6C	l	l	l
109	6D	m	m	m
110	6E	n	n	n
111	6F	o	o	o
112	70	p	p	p
113	71	q	q	q
114	72	r	r	r
115	73	s	s	s
116	74	t	t	t
117	75	u	v	u
118	76	v	v	v
119	77	w	w	w
120	78	x	x	x

De-c. Val-ue	He-x. Val-ue	Descripti-on	Fo-nt 1	Fonts 2, 3, 4, 5, & 6
121	79	y	y	y
122	7A	z	z	z
123	7B	right brace	}	{
124	7C	bar	|	|
125	7D	left brace	}	}
126	7E	tilde	~	~
127	7F	solid	▮	▮
128	80	copyright	©	©
129	81	registered trademar-k	®	®
130	82	trademar-k	TM	TM
131	83	bullet		·
132	84	superscrip-t +		+
133	85	double quote (modified)		"
134	86	checkmar-k	✓	
135	87	right-pointing	▶	

De-c. Val-ue	He-x. Val-ue	Descripti-on	Fo-nt 1	Fonts 2, 3, 4, 5, & 6
		triangle		
136	88	approxim-ately equal symbol	≈	≈
137	89	OE ligature		a
138	8A	oe ligature		i
139	8B	beta	ß	u
140	8C	Sigma	Σ	e
141	8D	Omega	Ω	o
142	8E	blank		ya
143	8F	blank		yu
144	90	dotless i	ı	yo
145	91	grave accent		Dbl next conson-ant
146	92	acute accent		-
147	93	circumflex		A
148	94	tilde		I
149	95	macron	¯	U

De-c. Val-ue	He-x. Val-ue	Descripti-on	Fo-nt 1	Fonts 2, 3, 4, 5, & 6
150	96	breve	˘	E
151	97	dot accent	˙	O
152	98	dieresis	¨	KA
153	99	blank		KI
154	9A	ring	°	KU
155	9B	cedilla	˛	KE
156	9C	blank		KO
157	9D	hungaru-mlaut	˝	SA
158	9E	ogonek	˛	SHI
159	9F	caron	ˇ	SU
160	A0	blank		Yen symbol
161	A1	inverted exclamati-on point	i	Closed circle
162	A2	cent	¢	Start quote
163	A3	sterling	£	End quote
164	A4	currency	¤	Comma
165	A5	yen	¥	End sentenc-

De-c. Val-ue	He-x. Val-ue	Descripti-on	Fo-nt 1	Fonts 2, 3, 4, 5, & 6
				e
166	A6	broken bar	¦	o
167	A7	section	§	a
168	A8	dieresis	¨	i
169	A9	copyright	©	u
170	AA	feminine ordinal	ª	e
171	AB	left guillemot	«	o
172	AC	logical not	¬	¬ya
173	AD	en dash	-	yu
174	AE	registered	®	yo
175	AF	macron	—	Dbl next conson-ant
176	B0	degree	°	-
177	B1	plus/minu-s	±	A
178	B2	superscrip-t 2	²	I
179	B3	superscrip-t 3	³	U

Dec. Value	Hex. Value	Description	Font 1	Fonts 2, 3, 4, 5, & 6
180	B4	acute accent	´	E
181	B5	mu	µ	O
182	B6	paragraph	¶	KA
183	B7	centered dot	·	KI
184	B8	cedilla	¸	KU
185	B9	¹	¹	KE
186	BA	masculine ordinal	º	KO
187	BB	right guillemot	»	SA
188	BC	1/4	¼	SHI
189	BD	1/2	½	SU
190	BE	3/4	¾	SE
191	BF	inverted question mark	¿	SO
192	C0	A grave	À	TA
193	C1	A acute	Á	CHI
194	C2	A circumflex	Â	TSU
195	C3	A tilde	Ã	TE

Dec. Value	Hex. Value	Description	Font 1	Fonts 2, 3, 4, 5, & 6
196	C4	A dieresis	Ä	TO
197	C5	A ring	Å	NA
198	C6	AE ligature	Æ	NI
199	C7	C cedilla	Ç	NU
200	C8	E grave	È	NE
201	C9	E acute	É	NO
202	CA	E circumflex	Ê	HA
203	CB	E dieresis	Ë	HI
204	CC	I grave	Ì	FU
205	CD	I acute	Í	HE
206	CE	I circumflex	Î	HO
207	CF	I dieresis	Ï	MA
208	D0	Eth	Đ	MI
209	D1	N tilde	Ñ	MU
210	D2	O grave	Ò	ME
211	D3	O acute	Ó	MO
212	D4	O circumflex	Ô	YA

De-c. Val-ue	He-x. Val-ue	Descripti-on	Fo-nt 1	Fonts 2, 3, 4, 5, & 6
213	D5	O tilde	Õ	YU
214	D6	O dieresis	Ö	YO
215	D7	multiply	×	RA
216	D8	O slash	Ø	RI
217	D9	U grave	Ù	RU
218	DA	U acute	Ú	RE
219	DB	U circumflex	Û	RO
220	DC	U dieresis	Ü	WA
221	DD	Y acute	Ý	N
222	DE	Thorn	Þ	Voiced conson-ant
223	DF	German double s	ß	Voiced conson-ant-P
224	E0	a grave	à	SE
225	E1	a acute	á	SO
226	E2	a circumflex	â	TA
227	E3	a tilde	ã	CHI
228	E4	a dieresis	ä	TSU

De-c. Val-ue	He-x. Val-ue	Description	Fo-nt 1	Fonts 2, 3, 4, 5, & 6
229	E5	a ring	å	TE
230	E6	ae ligature	æ	TO
231	E7	c cedilla	ç	NA
232	E8	e grave	è	NI
233	E9	e acute	é	NU
234	EA	e circumflex	ê	NE
235	EB	e dieresis	ë	NO
236	EC	i grave	ì	HA
237	ED	i acute	í	HI
238	EE	i circumflex	î	FU
239	EF	i dieresis	ï	HE
240	F0	eth	ð	HO
241	F1	n tilde	ñ	MA
242	F2	o grave	ò	MI
243	F3	o acute	ó	MU
244	F4	o circumflex	ô	ME
245	F5	o tilde	õ	MO

Dec. Value	Hex. Value	Description	Font 1	Fonts 2, 3, 4, 5, & 6
246	F6	o dieresis	ö	YA
247	F7	divide	÷	YU
248	F8	o slash	ø	YO
249	F9	u grave	ù	RA
250	FA	u acute	ú	RI
251	FB	u circumflex	û	RU
252	FC	u dieresis	ü	RE
253	FD	y acute	ý	RO
254	FE	thorn	þ	WA
255	FF	y dieresis	ÿ	N

OMRON Corporation   **Industrial Automation Company**

Kyoto, JAPAN

Contact: www.ia.omron.com

Regional Headquarters

OMRON EUROPE B.V.
Wegalaan 67-69, 2132 JD Hoofddorp
The Netherlands
Tel: (31)2356-81-300/Fax: (31)2356-81-388

OMRON ASIA PACIFIC PTE. LTD.
No. 438A Alexandra Road # 05-05/08 (Lobby 2),
Alexandra Technopark,
Singapore 119967
Tel: (65) 6835-3011/Fax: (65) 6835-2711

OMRON ELECTRONICS LLC
2895 Greenspoint Parkway, Suite 200 Hoffman Estates,
IL 60169 U.S.A.
Tel: (1) 847-843-7900/Fax: (1) 847-843-7787

OMRON ADEPT TECHNOLOGIES, INC.
4550 Norris Canyon Road, Suite 150, San Ramon, CA 94583 U.S.A.
Tel: (1) 925-245-3400/Fax: (1) 925-960-0590

OMRON (CHINA) CO., LTD.
Room 2211, Bank of China Tower, 200 Yin Cheng Zhong Road,
PuDong New Area, Shanghai, 200120, China
Tel: (86) 21-5037-2222/Fax: (86) 21-5037-2200

Authorized Distributor:

Printed in USA

Cat. No. I604-E-01   0316