

CHAPTER ONE

Programming Fundamentals

IN THIS CHAPTER

This chapter is a guide to general 6K programming tasks. It is divided into these main topics:

- | | | | |
|-----------------------------------------------------------|----|-------------------------------------------|----|
| • Motion Planner programming environment | 2 | • Restricted commands during motion | 17 |
| • Command syntax | 3 | • Using Variables | 18 |
| • Creating programs | 9 | • Program flow control | 23 |
| • Storing programs | 10 | • Program interrupts | 29 |
| • Executing programs | 13 | • Error handling | 30 |
| • Creating and executing a set-up program | 13 | • Non-volatile memory | 33 |
| • Program Security | 14 | • System performance considerations | 34 |
| • Controlling execution – programs & command buffer | 15 | | |

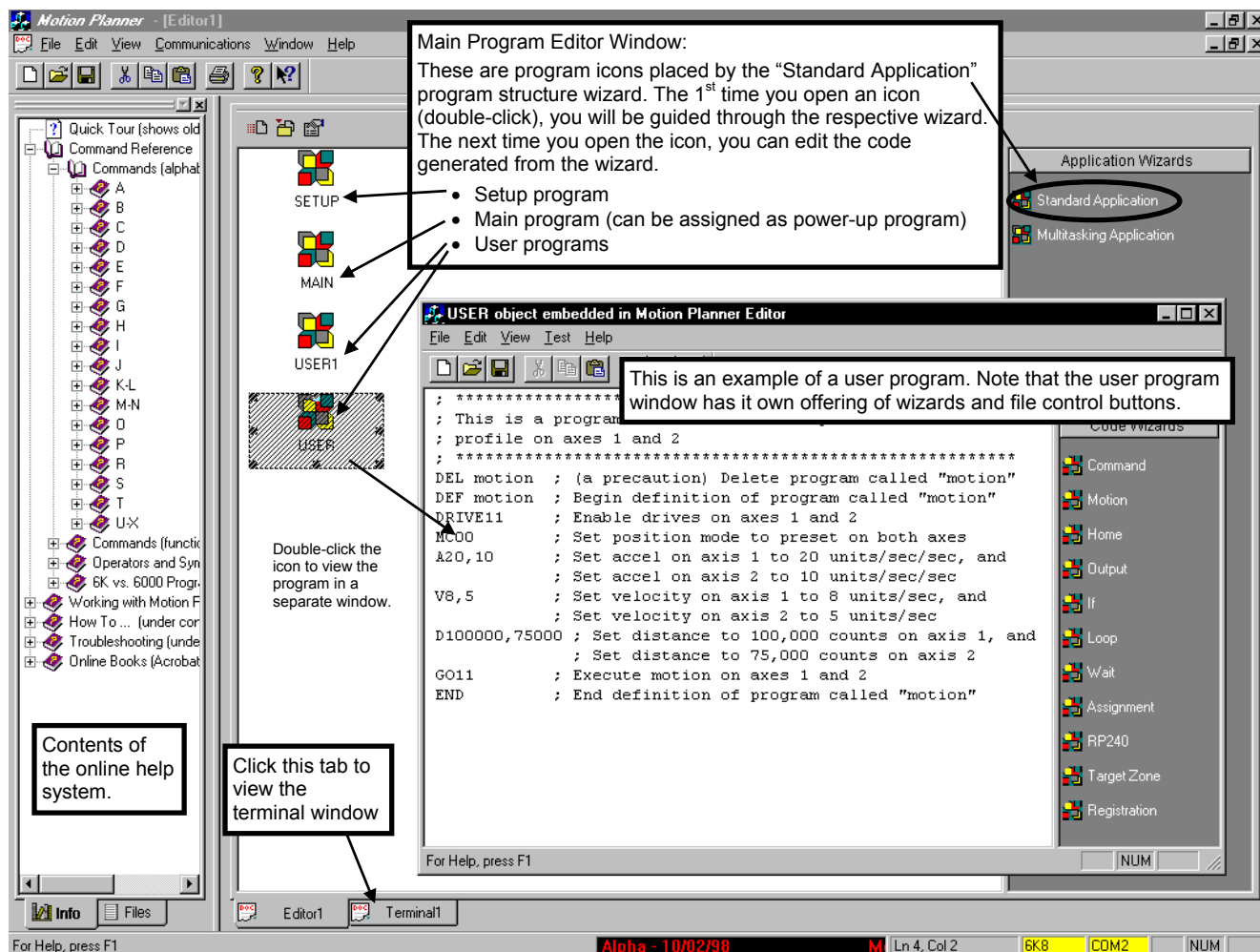
Motion Planner Programming Environment

Every 6K Series controller is shipped with Motion Planner, a Windows-based programming tool designed to simplify your programming efforts. The Motion Planner interface allows you to:

- Create, edit, download, and upload programs (or code modules).
- Tune your system to optimize performance.
- Test & debug programs and controller operation with customizable displays.
- Organize all of your programs and resource files for your programming project.

PROGRAMMING SUPPORT. To help you program with speed and efficiency, Motion Planner provides these “performance support” features:

- **Smart Editor:** The smart editor is the focal point for your programming tasks: The smart editor watches over your shoulder and provides syntax checking on the fly (as you type). To get detailed information on the command you’re using, just press the F1 key. At any point, you can check the entire program file for logic flow and syntax errors.
- **Programming Help with Wizards:** While you are working in the Editor, you can use the wizards to speed up your programming tasks and minimize your need to learn the details of the programming language. Wizards are available for general program structure, general system setup (including servo tuning), error programming, and a host of other programming tasks.



Command Syntax

Introduction

The 6K programming language accommodates a wide range of needs by providing basic motion control building blocks, as well as sophisticated motion and program flow constructs.

The language comprises simple ASCII mnemonic commands, with each command separated by a command delimiter (carriage return, colon, or line feed). The command delimiter signals the 6K product that a command is ready for processing.

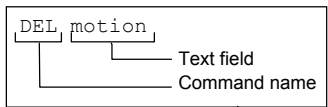
Upon receiving a command followed by a command delimiter, the 6K controller places the command in its internal command buffer, or queue. Here the command is executed in the order in which it is received. To make the command execute immediately, place an exclamation point (!) in front of it (e.g., The `TAS` command will be executed after all commands ahead of it in the command buffer are executed; but `!TAS` will execute before any other commands in the command buffer).

Sample program, as viewed in an editor:

```

; *****
; This is a program that executes a trapezoidal motion
; profile on axes 1 and 2
; *****
DEL motion      ; (a precaution) Delete program called
                "motion"
DEF motion      ; Begin definition of program called "motion"
DRIVE11        ; Enable drives on axes 1 and 2
MC00           ; Set position mode to preset on both axes
A20,10         ; Set accel on axis 1 to 20 units/sec/sec, and
                ; Set accel on axis 2 to 10 units/sec/sec
V8,5           ; Set velocity on axis 1 to 8 units/sec, and
                ; Set velocity on axis 2 to 5 units/sec
D100000,75000  ; Set distance to 100,000 counts on axis 1,
                and
                ; Set distance to 75,000 counts on axis 2
; *****
    
```

These are command line comments, comprising a semi-colon and text. The comments are separated from the command by a tab. A carriage return is placed at the end of each command line.

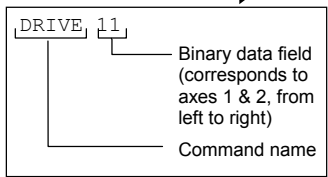


```

; *****
; This is a program that executes a trapezoidal motion
; profile on axes 1 and 2
; *****
    
```

```

DEL motion
DEF motion
DRIVE11
    
```



```

MC00
A20,10

V8,5

D100000,75000

GO11
END
    
```

```

; (a precaution) Delete program called "motion"
; Begin definition of program called "motion"
; Enable drives on axes 1 and 2
; Set position mode to preset on both axes
; Set accel on axis 1 to 20 units/sec/sec, and
; Set accel on axis 2 to 10 units/sec/sec
; Set velocity on axis 1 to 8 units/sec, and
; Set velocity on axis 2 to 5 units/sec
; Set distance to 100,000 counts on axis 1, and
; Set distance to 75,000 counts on axis 2
; Execute motion on axes 1 and 2
; End definition of program called "motion"
    
```

Spaces and tabs within a command are processed as neutral characters. Comments can be specified with the semicolon (;) character — all characters following the semicolon and before the command delimiter are considered program comments.

Some commands contain one or more data fields in which you enter numeric or binary values or text:

- **Numeric data fields.** For example, `A20,10` is an acceleration (A) command that sets the acceleration for axes 1 and 2 to 20 units/sec² and 10 units/sec², respectively.
- **Binary fields.** For example, `DRIVE1011` is a drive enable (DRIVE) command that enables axes 1, 3 and 4 and disables axis 2.
- **Text fields.** For example, `STARTPpowrup` is a startup program assignment (STARTP) command that assigns the program called “powrup” as the startup program to be executed automatically when the 6K product is power up or reset.
- To check what the data field settings are for a particular command, simply type in the command without the data fields. The 6K will display the command settings. For example, after executing the `A20,10` noted above, you could type in the `A` command by itself and the 6K controller would respond with `A20,10`.
- **Shortcuts.** Most 6K language commands supply axis-related data, and have one field per axis, separated by commas. Each command field correlates, left to right, to the physical axis. For example, to specify a velocity of 10 on axes 6 and 8, the command “`V , , , , ,10, ,10`” would be used. As can be seen from the example, the required number of commas can be awkward, and could be a potential source of typographical error. The 6K products allow an axis specifier to be placed in front of a command with axis fields to identify the starting axis number for the fields in that command. For example, the above `V` command could be given as “`6V10, ,10`”. If the velocity were to be given to axis 6 only, the command would simply be “`6V10`”. An axis identifier placed in front of a data command without parameters (e.g. `6V`) will report the value for that axis only.

Description of Syntax Letters and Symbols

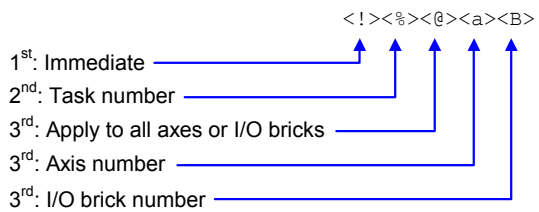
The command descriptions provided within the *6K Series Command Reference* use alphabetic letters and ASCII symbols within the **Syntax** description to represent different parameter requirements (see `INEN` example below).

INEN		Input Enable		Product	Rev
Type	Inputs; Program Debug Tools			6K	5.0
Syntax	<code><!>INEN<d><d><d>...<d></code>				
Units	d = Ø, 1, E, or X				
Range	Ø = off, 1 = on, E = enable, X = don't change				
Default	E				
Response	INEN: *INENEEEE_EEEE_EEEE_EEEE_E				
See Also	[IN], INFNC, INLVL, INPLC, INSTW, TIN, TIO				

Letter/Symbol	Description
a	Represents an axis specifier, numeric value from 1 to 8.
B	Represents the number of the product's I/O brick. External I/O bricks are represented by numbers 1 through n (to connect external I/O bricks, see your product's <i>Installation Guide</i>). On-board I/O are address at brick location zero (Ø). If the brick identifier is omitted from the command, the controller assumes the command is supposed to affect the onboard I/O.
b*	Represents the values 1, 0, X or x; does not require field separator between values.
c	Represents a character (A to Z, or a to z)
d	Represents the values 1, 0, X or x, E or e ; does not require field separator between values. E or e enables a specific command field. X or x leaves the specific command field unchanged or ignored. In the <code>ANIEN</code> command, the "d" symbol can also represent a real numeric value.
i	Represents a numeric value that cannot contain a decimal point (integer values only). The numeric range varies by command. Field separator required.
r	Represents a numeric value that can contain a decimal point, but is not required to have a decimal point. The numeric range varies by command. Field separator required.
t	Represents a string of alpha numeric characters from 1 to 6 characters in length. The string must start with a alpha character.
!	Represents an immediate command. Changes a buffered command to an immediate command. Immediate commands are processed immediately, even before previously entered buffered commands.
%	(Multitasking Only) Represents a task identifier. To address the command to a specific task, prefix the command with "i%", where "i" is the task number. For example, the <code>4%<i>CUT</i></code> command uses task 4 to execute the program called " <i>CUT</i> ".
,	(comma) Represents a field separator. Commands with the symbol <code>r</code> or <code>i</code> in their Syntax description require field separators. Commands with the symbol <code>b</code> or <code>d</code> in their Syntax description <u>do not</u> require field separators (but they can be included). See <i>General Guidelines</i> table below.
@	Represents a global specifier, where only one field need be entered. Applicable to all commands with multiple command fields. (e.g., <code>@V1</code> sets velocity on all axes to 1 rps).
< >	Indicates that the item contained within the < > is optional, not required by that command. NOTE: Do not confuse with <code><cr></code> , <code><sp></code> , and <code><lf></code> , which refer to the ASCII characters corresponding to a carriage return, space, and line feed, respectively.
[]	Indicates that the command between the [] must be used in conjunction with another command, and cannot be used by itself.

* The ASCII character `b` can also be used within a command to precede a binary number. When the `b` is used in this context, it is not to be replaced with a 0, 1, X, or x. Examples are assignments such as `VARB1=b10001`, and comparisons such as `IF(3IN=b1001X1)`.

Order of Precedence for Command Prefix Characters (from left to right):



General Guidelines for Syntax

Guideline Topic	Guideline	Examples
Neutral Characters <ul style="list-style-type: none"> Space (<sp>) Tab (<tab>) 	Using neutral characters anywhere within a command will not affect the command. (In the examples on the right, a space is represented by <sp>, a tab is <tab>), and a carriage return is <cr>	Set velocity on axis 1 to 10 rps and axis 2 to 25 rps: <code>V<sp>10,<sp>25,,<cr></code> Add a comment to the command: <code>V 10, 25,,<tab> ;set accel <cr></code>
Command Delimiters: <ul style="list-style-type: none"> Carriage rtn (<cr>) Line feed (<lf>) Colon (:) 	All commands must be separated by a command delimiter. A carriage return is the most commonly used delimiter. To use a line in a live terminal emulator session, press ctrl/J. The colon (:) delimiter allows you to place multiple commands on one line of code, but only if you add it in the program editor (not during a live terminal emulator session).	Set acceleration on axis 2 to 10 rev/sec/sec: <code>A,10,,<cr></code> <code>A,10,,<lf></code> <code>A,10,, : V,25,, : D,25000,, : @GO<cr></code>
Case Sensitivity	There is no case sensitivity. Use upper or lower case letters within commands.	Initiate motion on axes 1, 3 and 4: <code>GO1011</code> <code>go1011</code>
Comment Delimiter (;)	All text between a comment delimiter and a command delimiter is considered <i>program comments</i> .	Add a comment to the command: <code>V10<tab> ;set velocity</code>
Field Separator (,)	Commands with the symbol <i>r</i> or <i>i</i> in their Syntax description require field separators. Commands with the symbol <i>b</i> or <i>d</i> in their Syntax description do not require field separators (but they can be included). Axes not participating in the command need not be specified; however, field separators that are normally required must be specified (unless the axis prefix is used).	Set velocity on axes 1 - 4 to 10 rps, 25 rps, 5 rps and 10 rps, respectively: <code>V10,25,5,10</code> Initiate motion on axes 1, 3 and 4: <code>GO1011</code> <code>GO1,0,1,1</code> Set velocity on axes 4 and 6 to 5 rps: <code>V,,,5,,5</code> Alternative is to use the axis prefix: <code>4V5,,5</code>
Global Command Identifier (@)	When you wish to set the command value equal on all axes, add the @ symbol at the beginning of the command (enter only the value for one command field). The @ symbol is also useful for checking the status of all axes, or all inputs or outputs on all I/O bricks.	Set velocity on all axes to 10 rps: <code>@V10</code> Check the status of all digital outputs (onboard, and on external I/O bricks): <code>@OUT</code>
Bit Select Operator (.)	The bit select operator allows you to affect one binary bit without having to enter all the preceding bits in the command. Syntax for setup commands: <code>[command name].[bit #]-[binary value]</code> Syntax for conditional expressions: <code>[command name].[bit #]=[binary value]</code>	Enable error-checking bit 9: <code>ERROR.9-1</code> IF statement based on value of axis status bit 12 for axis 1: <code>IF(1AS.12=b1)</code>
Left-to-right Math	All mathematical operations assume left-to-right precedence.	<code>VAR1=5+3*2</code> Result: Variable 1 is assigned the value of 16 (8*2), not 11 (5+6).
Binary and hexadecimal values	When making assignments with or comparisons against binary or hexadecimal values, you must precede the binary value with the letter "b" or "B", and the hex value with "h" or "H". In the binary syntax, an "x" simply means the status of that bit is ignored.	Binary: <code>IF(IN=b1x01)</code> ↑ Hexadecimal: <code>IF(IN=h7F)</code> ↑
Multi-tasking Task Identifier (%)	Use the % command prefix to identify the command with a specific task.	Launch the "move1" program in Task 1: <code>1%move1</code> Check the error status for Task 3: <code>3%TER</code> Check the system status for Task 3: <code>3%TSS</code>

NOTE: The command line is limited to 100 characters (excluding spaces).

Command Value Substitutions

Many commands can substitute one or more of its command field values with one of these substitution items (demonstrated in the programming example below):

VAR.....Places current value of the numeric variable in the corresponding command field.
VARBUses the value of the binary variable to establish all the command fields.
VARIPlaces current value of the integer variable in the corresponding command field.
READInformation is requested at the time the command is executed.
DREAD.....Reads the RP240's numeric keypad into the corresponding command field.
DREADF...Reads the RP240's function keypad into the corresponding command field.
TW.....Places the current value set on the thumbwheels in the corresponding command field.
DAT.....Places the current value of the data program (DATP) in the corresponding command field.

Programming Example: (NOTE: The substitution item must be enclosed in parentheses.)

```
VAR1=15           ; Set variable 1 to 15
A5,(VAR1),4,4     ; Set acceleration to 5,15,4,4 for axes 1-4, respectively
VARB1=b1101XX1   ; Set binary variable 1 to 1101XX1 (bits 5 & 6 not affected)
GO(VARB1)         ; Initiate motion on axes 1, 2 & 4 (value of binary
                  ; variable 1 makes it equivalent to the GO1101 command)
OUT(VARB1)        ; Turn on outputs 1, 2, 4, and 7
VARS1="Enter Velocity" ; Set string variable 1 to the message "Enter Velocity"
V2,(READ1)        ; Set the velocity to 2 on axis 1. Read in the velocity for
                  ; axis 2, output variable string 1 as the prompting message
                  ; 1. Operator sees "ENTER VELOCITY" displayed on the
                  ; screen.
                  ; 2. Operator enters velocity prefixed by '!' (e.g., '!20').
HOMV2,1,(TW1)     ; Set homing velocity to 2 and 1 on axes 1 and 2,
                  ; respectively.
                  ; Read in the home velocity for axis 3 from thumbwheel set 1
HOMV2,1,(DAT1)    ; Set homing velocity to 2 and 1 on axes 1 and 2,
                  ; respectively.
                  ; Read home velocity for axis 3 from data program 1.
VARI1=2*3         ; Set integer variable 1 to 6 (2 multiplied by 3)
D(VARI2),,(VARI3) ; Set the distance of axis 1 equal to the value of
                  ; integer variable 2, and the distance of axis 3 equal to
                  ; the value of integer variable 3.
```

RULE OF THUMB

Not all of the commands allow command field substitutions. In general, commands with a binary command field (in the command syntax) will accept the VARB substitution. Commands with a real or integer command field (<r> or <i> in the command syntax) will accept VAR, VARI, READ, DREAD, DREADF, TW or DAT.

Assignment and Comparison Operators

Comparison and assignment operators are used in command arguments for various functions such as variable assignments, conditional branches, wait statements, conditional GOs, etc. Some examples are listed below:

- Assign to numeric variable 6 the value of the encoder position on axis 3 (uses the PE operator): VAR6=3PE
- Wait until onboard inputs 3 & 6 become active (uses the IN operator): WAIT(IN=bxxlxx1)
- Continue until the value of numeric variable 2 is less than 36: UNTIL(VAR2<36)
- IF condition based on if a target zone timeout occurs on axis 2 (uses the AS axis status operator, where status bit 25 is set if a target zone timeout occurs): IF(2AS.25=b1)

The available comparison and assignment operators are listed below. For full descriptions, see the *6K Series Command Reference* (be sure to refer only to the commands in brackets—e.g., A is the acceleration setup command, but [A] is the acceleration assignment/comparison operator).

* denotes operators that have a correlated status display command.
(e.g., To see a full-text description of each axis status bit accessed with the AS operator, send the TASF command to the 6K controller.)
See page 226.

A Acceleration
AD Deceleration
ANI Voltage at the analog inputs on an expansion I/O brick (see page 76 for bit patterns) *
ANO Voltage at the analog outputs on an expansion I/O brick (see page 76 for bit patterns) *
AS Axis status *
ASX Extended axis status (additional axis status items) *
D Distance
DAC Digital-to-analog converter (output voltage) value *
DAT Data program number
DKEY Value of RP240 Key
DPTR Data pointer location *
DREAD Data from the numeric keypad on the RP240
DREADF Data from the function keypad on the RP240
ER Error status *
FB Position of current selected feedback sources *
FS Following status *
IN Input status (input bit patterns, see page 76) *
INO “Other” input status (ENABLE input reported with bit 6) *
LIM Limit status (end-of-travel limits and home limits) *
MOV Axis moving status
NMCY Current master cycle number *
OUT Output status (output bit patterns, see page 76) *
PANI Position of analog input, at 205 counts/volts unless otherwise scaled (servo axes) *
PC Commanded position *
PCC Captured commanded position *
PCE Captured encoder position *
PCME Captured master encoder position *
PCMS Captured master cycle position *
PER Position error (servo axes only) *
PME Current master encoder position *
PMAS Current master cycle position *
PE Position of master encoder *
PSHF Net position shift since constant Following ratio *
PSLV Current commanded position of the slave axis *
READ Read a numeric value to a numeric variable (VAR)
SC Controller status *
SCAN Runtime of the last scanned PLC program *
SEG Number of segments available in Compiled Profile memory *
SS System status *
SWAP Current active status of tasks *
TASK Number of the controlling task *
TIM Timer value *
TRIG Trigger interrupt status *
TW Thumbwheel data read
US User status *
V Velocity (programmed)
VAR Numeric variable substitution
VARI Integer variable substitution
VARB Binary variable substitution
VEL Velocity (commanded by the controller) *
VELA Velocity (actual, as measured by a position feedback device) *
VMAS Current velocity of the master axis *

Bit Select Operator

The bit select operator (.) makes it easier to base a command argument on the condition of one specific status bit. For example, if you wish to base an IF statement on the condition that a user fault input is activated (error status bit 7 is a binary status bit that is “1” if a user fault occurred and “Ø” if it has not occurred), you could use this command: IF (ER=bxxxxxx1) . Using a bit select operator, you could instead use this command: IF (ER. 7=b1) .

NOTE: You can use a bit select operator to set a particular status bit (e.g., to turn on onboard programmable output 5, you would type the OUT . 5-1 command; to enable error-checking bit 4 to check for drive faults, you would type the ERROR . 4-1 command). You can also check specific status bits (e.g., to check axis 2’s axis status bit 25 to see if a target zone timeout

occurred, type the `2TAS.25` command and observe the response).

Binary and Hex Values

When making assignments with or comparisons against binary or hexadecimal values, you must precede the binary value with the letter “b” or “B”, and the hex value with “h” or “H”.

Examples: `IF (IN=b1x01)` and `IF (IN=h7F)`. In the binary syntax, an “x” simply means the status of that bit is ignored. Refer also to *Using Binary Variables* (page 22).

Related Operator Symbols

Command arguments include special operator symbols (e.g., +, /, &, ', >=, etc.) to perform bitwise, mathematical, relational, logical, and other special functions. These operators are described in detail, along with programming examples, at the beginning of the *Command Descriptions* section of the *6K Series Command Reference*.

Programmable Inputs and Outputs Bit Patterns

I/O pin outs, specifications, and circuit drawings are provided in each 6K Series Hardware Installation Guide.

The 6K product has programmable inputs and outputs. The total number of onboard inputs and outputs (trigger inputs, limit inputs, digital outputs) depends on the product. The total number of expansion inputs and outputs (analog inputs, digital inputs and digital outputs) depends on your configuration of expansion I/O bricks connected to the “EXPANSION I/O” connector.

These programmable I/O are represented by binary bit patterns, and it is the bit pattern that you reference when programming and checking the status of specific inputs and outputs. The bit pattern is referenced in commands like `WAIT (IN.4=b1)`, which means wait until onboard programmable input 4 (TRG-2B) becomes active. To ascertain your product’s I/O offering and bit patterns, see Chapter 3 (page 76).

Creating Programs

Debugging Programs: Refer to page 225 for methods to isolate and resolve programming problems.

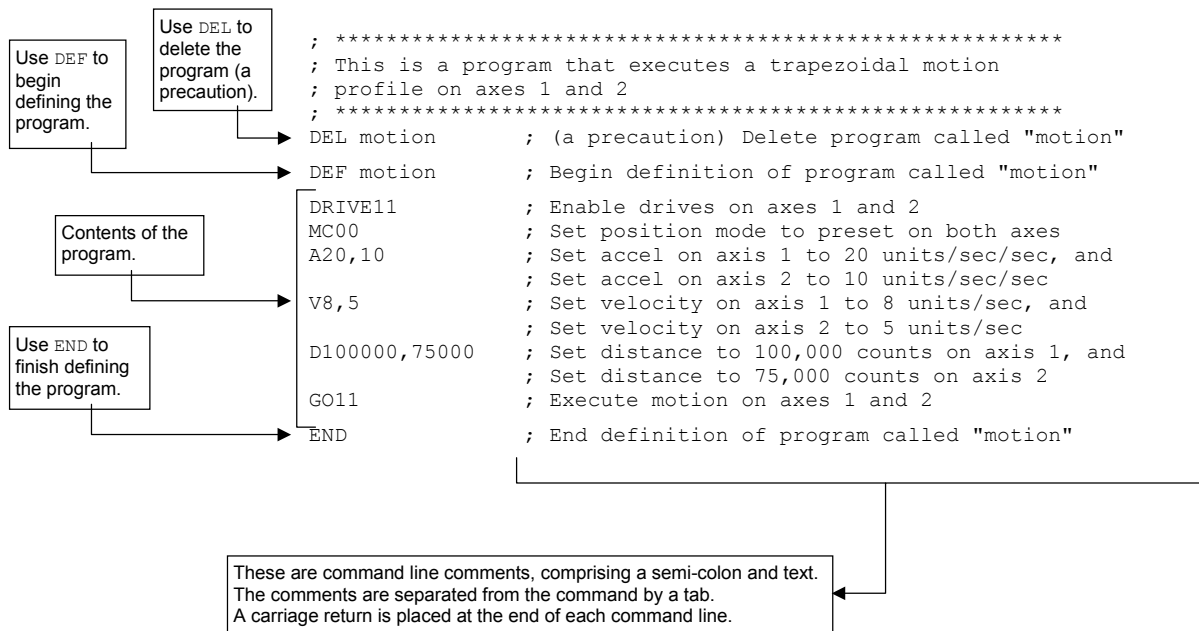
A *program* is a series of commands. These commands are executed in the order in which they are programmed. Immediate commands (commands that begin with an exclamation point [!]) cannot be stored in a program. Only buffered commands can be used in a program. Refer to the program example below.

A *subroutine* is defined the same as a program, but it is executed with an unconditional branch command, such as `GOSUB`, `GOTO`, or `JUMP`, from another program (see page 23 for details about unconditional branching). Subroutines can be nested up to 16 levels deep. **NOTE:** The 6K family does not support recursive calling of subroutines.

Compiled profiles & PLC programs are defined like programs, using the `DEF` and `END` commands, but are compiled with the `PCOMP` command and executed with the `PRUN` command (PLC programs are usually launched in PLC Scan Mode with the `SCANP` command). Compiled profiles and PLC programs also affect a different part of the product’s memory, called *compiled memory*. A compiled *profile* could be a multi-axis contour (a series of arcs and lines), an individual axis profile (a series of `GOBUF` commands), or a compound profile (combination of multi-axis contours and individual axis profiles). A compiled *PLC program* is a pre-compiled program that mimics PLC functionality by scanning through the I/O faster than in normal program execution. For information on contouring, see page 124; for information on compiled individual axis profiles, see page 136; and for information on PLC programs, see page 104.

Program Example

The illustration below identifies the elements that comprise the general structure of a program.



Use the Wizards in Motion Planner

Motion Planner provides wizards that make it easy to create your program. Below is a partial list of the wizards available.

- Application Wizards (for program structure guidance)
 - Standard Application
 - Multitasking Application
- Program Wizards
 - Setup Program
 - Main Program
 - User Program
 - Error Program
- Setup Wizards
 - Drive
 - Feedback
 - Scaling
 - Limit
 - Servo Tuner
 - On-board I/O
 - Expansion I/O
 - Jogging
 - Joystick
 - Variable
- Action Wizards
 - Motion
 - Home
 - Output
 - If
 - Loop
 - Wait
 - Assignment
 - Target Zone
 - Registration

Storing Programs

After a program or compiled program/profile is defined (DEF) or downloaded to the 6K controller, it is automatically stored in non-volatile memory (battery-backed RAM). Information on controlling memory allocation is provided below (Memory Allocation, see page 11).

Memory Allocation

Your controller's memory has two partitions: one for storing *programs* and one for storing *compiled profiles & PLC programs*. The allocation of memory to these two areas is controlled with the MEMORY command.

“Programs” vs. “Compiled Profiles & PLC Programs”

Programs are defined with the DEF and END commands, as demonstrated in the *Program Example* on page 10.

Compiled Profiles & PLC Programs are defined like programs, using the DEF and END commands, but are compiled with the PCOMP command and executed with the PRUN command (PLC programs are usually executed in PLC Scan Mode with the SCANP). A compiled profile could be a multi-axis *contour* (a series of arcs and lines), an *individual axis profile* (a series of GOBUF commands), or a *compound profile* (combination of multi-axis contours and individual axis profiles). A PLC program is a pre-compiled program that mimics PLC functionality by scanning through the I/O faster than in normal program execution.

Programs intended to be compiled are stored in program memory. After they are compiled with the PCOMP command, they remain in program memory and the *segments* (see diagram below) from the compiled program are stored in compiled memory. The TDIR report indicates which programs are compiled as compiled profiles (“COMPILED AS A PATH”) and which programs are compiled as PLC programs (“COMPILED AS A PLC PROGRAM”).

For information on contouring, see page 124; for information on compiled individual axis profiles, see page 136; and for information on PLC programs, see page 104.

MEMORY
command
syntax
(example)

MEMORY150000, 150000

Memory allocation for Programs (bytes). Storage requirements depend on the number of ASCII characters in the program.

Memory allocation for Compiled Profiles & Programs (bytes). Storage requirements depend on the number of *segments* (1 segment consumes 76 bytes). A segment could be one of these commands:

Contouring:	Compiled Motion:	PLC (PLCP) Program:
PARCM	GOBUF *	IF **
PARCOM	PLOOP	ELSE
PARCOP	GOWHEN	NIF
PARCP	TRGFN	L
PLIN	POUTA	LN
	POUTB	OUT
	POUTC	ANO
	POUTD	EXE
	POUTE	PEXE
	POUTF	VARI **
	POUTG	VARB **
	POUTH	

* GOBUF commands may require up to 4 segments.

** IF statement require at least 2 segments, each AND or OR compound requires an additional segment. VARI and VARB each require 2 segments.

The table below identifies memory allocation defaults and limits for all 6K Series products. When specifying the memory allocation, use only even numbers. The minimum storage capacity for one partition area (program or compiled) is 1,000 bytes.

Feature	All Other Products
Total memory (bytes)	300,000
Default allocation (program,compiled)	150000,150000
Maximum allocation for programs	299000,1000
Maximum allocation for compiled profiles & PLC programs	1000,299000
Maximum No. of programs	400
Maximum No. of labels	600
Maximum No. of compiled profiles & PLC programs	300
Maximum No. of compiled profile segments	2069
Maximum No. of numeric variables (VAR)	225
Maximum No. of integer variables (VARI)	225
Maximum No. of binary variables (VARB)	125
Maximum No. of string variables (VARS)	25

When teaching variable data to a data program (DATP), be aware that the memory required for each data statement of four data points (43 bytes) is taken from the memory allocation for program storage (see *Variable Arrays* in Chapter 3, page 94, for details).

CAUTION

Using a memory allocation command (e.g., MEMORY200000,100000) will erase all existing programs and compiled profile segments & PLC programs. However, issuing the MEMORY command without parameters (i.e., type MEMORY <cr> to request the status of how the memory is allocated) will not affect existing programs or compiled segments/programs.

Checking Memory Status

To find out what programs reside in your controller's memory, and how much of the available memory is allocated for programs and compiled profile segments, issue the TDIR command (see example response below). Entering the TMEM command or the MEMORY command (without parameters) will also report the available memory for programs and compiled profile segments.

Sample response to
TDIR command

```
*1 - SETUP USES 345 BYTES
*2 - PIKPRT USES 333 BYTES
*149322 OF 150000 BYTES (98%) PROGRAM MEMORY REMAINING
*1973 OF 1973 SEGMENTS (100%) COMPILED MEMORY REMAINING
```

Two system status bits (reported with the TSS and SS commands) are available to check when compiled profile segment storage is 75% full or 100% full. System status bit 29 is set when segment storage reaches 75% of capacity; bit 30 indicates when segment storage is 100% full.

Executing Programs (options)

Following is a list of the primary options for executing programs stored in your controller:

Method	Description	See Also
Execute from a terminal emulator	Type in the name of the program and press enter; or write a program to prompt the operator to select a program from the terminal.	-----
Execute as a subroutine from a "main" program	Use a branch (<code>GOTO</code> , <code>GOSUB</code> , or <code>JUMP</code>) from the main program to execute another stored program.	Page 23
Execute automatically when the controller is powered up	Assign a specific program as a startup program with the <code>STARTP</code> command. When you <code>RESET</code> or cycle power to the controller, the startup program is automatically executed.	Page 13
Execute from a PLC program	Write a PLC program that executes a program (using <code>EXE</code> or <code>PEXE</code>) based on a specific condition (e.g., input state). Use the <code>SCANP</code> command to launch the PLC program in the PLC Scan Mode.	Page 104
Execute a specific program with BCD weighted inputs	Define programmable inputs to function as BCD select inputs, each with a BCD weight. A specific program (identified by its number) is executed based on the combination of active BCD inputs. Related commands: <code>INSELP</code> and <code>INFNCi-B</code> or <code>LIMFNCi-B</code> .	Page 82
Execute a specific program with a dedicated input	Define a programmable input to execute a specific program (by number). Related commands: <code>INSELP</code> and <code>INFNCi-IP</code> or <code>LIMFNCi-P</code> .	Page 88
"Call" from a high-level program	Using a programming language such as BASIC or C, write a program that enables the computer to monitor processes and orchestrate motion and I/O by executing stored programs (or individual commands) in the controller.	Page 118
Execute from an RP240 (remote operator interface)	Execute a stored program from the <code>RUN</code> menu in the RP240's standard menu system.	Page 111
Execute from your own custom Windows program	Use a programming language (e.g., Visual Basic, Visual C++, etc.) and the 6K Communications Server (provided on the Motion Planner CD) to create your own windows application to control the 6K product.	-----

Creating and Executing a Setup Program

The intent of the Setup program is to place the 6K controller in a ready state for subsequent motion control. The setup program must be called from the "main" program for your application; or you can designate (with `STARTP`) the setup program as the program to be automatically executed when the 6K product is powered up or when the `RESET` command is executed. The setup program typically contains elements such as feedback device configuration, tuning gain selections, programmable I/O definitions, scaling, homing configuration, variable initialization, etc. (more detail on these "basic" features is provided in Chapter 3, *Basic Operation Setup*).

The basic process of creating a setup program is:

1. Create a program to be used as the setup program.
2. Save the program and download it to the 6K product.
3. Execute the `STARTP` command to assign your new program as the "start-up" program (e.g., `STARTP setup` assigns the program called "setup" as the start-up program). The next time the controller is powered up or reset, the assigned `STARTP` program will be executed.

Or call the setup program from the main program for your application.

Use **Motion Planner's Setup wizard** to help you create the basic configuration program. By simply responding to a series of dialog boxes, a program is created with a specific name (as if you created it in the usual process with the `DEF` and `END` commands). You can further edit this program in Motion Planner's Editor if you wish. Use the following procedure:

1. From the main Editor window, click the "Standard Application" wizard button (located on the right-hand side of the screen under Application Wizards) and select "Setup" and "Main" from the dialog. When you click "Finish", Motion Planner places a Setup program icon and a Main program icon in the Editor window.
2. Double-click the Setup program icon to launch the wizard. Complete the wizard dialogs and click "Finish" to complete the wizard. (The next time you open the icon, you will see a program editor with the code resulting from the setup wizard.) Setup elements include:
 - Product selection
 - Drives
 - Feedback (encoder, analog input)
 - Scaling
 - Hardware end-of-travel limits
 - Servo tuning
3. Double-click the Main program icon to launch the wizard. Select this program as the program to launch when the 6K controller is reset or powered up (this is equivalent to the `STARTP` command function). In the dialog for selecting the Setup Program, select the program developed in step 2 above.
4. Save the Editor files.
5. Download the files to the 6K controller.

Program Security

Issuing the `INFNCi-Q` or `LIMFNCi-Q` command enables the *Program Security* feature and assigns the *Program Access* function to the specified programmable input. The "i" represents the number of the programmable input to which you wish to assign the function (see page 76 programmable input bit patterns for your product).

The program security feature denies you access to the `DEF`, `DEL`, `ERASE`, `MEMORY`, `INFNC`, and `LIMFNC` commands until you activate the program access input. Being denied access to these commands effectively restricts altering the user memory allocation. If you try to use these commands when program security is active (program access input is not activated), you will receive the error message `*ACCESS DENIED`.

For example, once you issue the `INFNC5-Q` command, onboard input 5 is assigned the program access function and access to the `DEF`, `DEL`, `ERASE`, `MEMORY`, `INFNC`, and `LIMFNC` commands will be denied until you activate onboard input 5.

NOTE: To regain access to these commands without the use of the program access input, you must issue the `INEN` command to disable the program security input, make the required user memory changes, and then issue the `INEN` command to re-enable the input. For example, if input 3 on I/O brick 2 is assigned as the Program Security input, use `2INEN.3=1` to disable the input and leave it activated, make the necessary user memory changes, and then use `2INEN.3=E` to re-enable the input.

Controlling Execution of Programs and the Command Buffer

The 6K controller command buffer is capable of storing 2000 characters waiting to be processed. (This is separate from the memory allocated for program storage – see Memory Allocation, page 11.) COMEXC affects command execution. Three additional commands, COMEXL, COMEXR and COMEXS, affect the execution of programs and the command buffer.

COMEXC (Continuous Command Execution)

The COMEXC1 command enables the Continuous Command Execution Mode (default is COMEXC0). This mode allows the program to continue to the next command before motion is complete. This is useful for:

- Monitoring other processes while motion is occurring
- Performing calculations in advance of motion completion
- Pre-emptive GOs — executing a new profile with new attributes (distance, accel/decel, velocity, positioning mode, and Following ratio) before motion is complete: The motion profile underway is pre-empted with a new profile when a new GO is issued. The new GO both constructs and launches the pre-empting profile. Pre-emptive GOs are appropriate when the desired motion parameters are not known until motion is already underway. For a detailed description, see On-The-Fly Motion on page 151.
- Pre-process the next move while the current move is in progress (see CAUTION). This reduces the processing time for the subsequent move to only a few microseconds.

CAUTION: Avoid executing moves prematurely

With continuous command execution enabled (COMEXC1), if you wish motion to stop before executing the subsequent move, place a WAIT (AS.1=b0) statement before the subsequent GO command. If you wish to ensure the load settles adequately before the next move, use the WAIT (AS.24=b1) command instead (this requires you to define end-of-move settling criteria — see Target Zone Mode on page 74 for details).

In the programming example below, by enabling the continuous command execution mode (COMEXC1), the controller is able to turn on output 3 after the encoder moves 4000 units of its 125000-unit move. Normally, with COMEXC disabled (COMEXC0), command processing would be temporarily stopped at the GO1 command until motion is complete.

Programming Example (portion of program only)

```
COMEXC1           ;Enable continuous command execution mode
D125000           ;Set distance
V2                ;Set velocity
A10              ;Set acceleration
GO1              ;Initiate motion on axis 1
WAIT(1PE>4000)   ;Wait for the encoder position to exceed 4000
OUTXX1           ;Turn on onboard programmable output 3
WAIT(AS.1=b0)    ;Wait for motion to complete on axis 1 (AS bit 1 = zero)
OUTXX0           ;Turn off onboard programmable output 3
```

COMEXL (Save Command Buffer on Limit)

The COMEXL command enables saving the command buffer and maintaining program execution when a hardware or software end-of-travel limit is encountered. COMEXL is axis specific (e.g., COMEXL1xx1xxx1 enables saving the buffer for axes 1, 4, and 8).

For more information
on end-of-travel limits,
see page 57.

- COMEXL0: (This is the default setting.) When a limit is hit, every command in the command buffer will be discarded and program execution will be terminated.
- COMEXL1: When a limit is hit, all remaining commands in the command buffer will remain in the command buffer (excluding the command being executed at the time the limit is hit).

COMEXR (Effect of Pause/Continue Input)

The COMEXR command affects whether a “Pause” input (i.e., an input configured as a pause/continue input with the INFNCi-E command or the LIMFNCi-E command) will pause only program execution or both program execution and motion.

COMEXR0: (This is the default setting.) Upon receiving a pause input, only program execution will be paused; any motion in progress will continue to its predetermined destination. Releasing the pause input or issuing a !C command will resume program execution.

COMEXR1: Upon receiving a pause input, both motion and program execution will be paused; the motion stop function is used to halt motion. *After motion has come to a stop (not during deceleration)*, you can release the pause input or issue a !C command to resume motion and program execution.

Other Ways to Pause

- Issue the PS command before entering a series of buffered commands (to cause motion, activate outputs, etc.), then issue the !C command to execute the commands.
- While program execution is in progress, issuing the !PS command stops program execution, but any move currently in progress will be completed. Resume program execution with the !C command.

COMEXS (Save Command Buffer on Stop)

The COMEXS command determines the impact on motion, program execution, and the command buffer when the 6K receives a Stop command (S, !S, S1, or !S1) or an external Stop input (an input assigned a stop function with INFNCi-D or LIMFNCi-D).

COMEXS0: Under factory default conditions (COMEXS0), when the 6K receives a stop command (S, !S, S1, or !S1) or a stop input (INFNCi-D or LIMFNCi-D), the following will happen:

- Motion decelerates to a stop, using the present AD and ADA deceleration values. The motion profile cannot be resumed.
- If S, !S or Stop input:
 - All commands in the 6K’s command buffer are discarded.
 - Program execution is terminated and cannot be resumed.
- If S1, or !S1 (an axis number is included in the command):
 - All commands in the 6K’s command buffer are retained.
 - Program execution continues.

COMEXS1: Using the COMEXS1 mode, the 6K allows more flexibility in responding to stop conditions, depending on the stop method (see table below).

Stop Method	What Stops?		Resume Motion Profile. (Allow resume with a !C command or a resume input *)	Resume Program. (Allow resume with a !C command or a resume input *)	Save Command Buffer. (Save the commands that were in the command buffer when the stop was commanded)
	Motion	Program			
!S or S	Yes	Yes	Yes	Yes	Yes
!S1 or S1	Yes	No	No	No	Yes
Stop input	Yes	Yes	No	Yes	Yes
Pause input * (if COMEXR1)	Yes	Yes	Yes	Yes	Yes
Pause input * (if COMEXR0)	No	Yes	No	Yes	Yes

* A Pause input is an input configured with the INFNCi-E command or the LIMFNCi-E command. This is also the Resume input that can be used to resume motion and program execution after motion is stopped.

COMEXS2: Using the COMEXS2 mode, the 6K responds as it does in the COMEXS0 mode, with the exception that you can still use the program-select inputs to select programs (INSELP value is retained). The program-select input functions are: BCD select (INFNCi-B or LIMFNCi-B), and one-to-one select (INFNCi-P or LIMFNCi-P).

Restricted Commands During Motion

When motion is in progress on a given axis (or task), some commands cannot have their parameters changed until motion is complete (see table below).

For the commands identified in the table, if the continuous command execution mode is enabled (COMEXC1) and you try to enter new command parameters, you will receive the error response MOTION IN PROGRESS. If the continuous command execution mode is disabled (COMEXC0), which is the default setting, you will receive the response MOTION IN PROGRESS only if you precede the command with the immediate (!) modifier (e.g., !V20); if you enter a command without the immediate modifier (e.g., V20), you will not receive an error response and the new parameter will be ignored and the old parameter will remain in effect.

Multi-Tasking

If you are using multi-tasking, the restriction on commands is applicable only for the task to which the command is direct. For example, suppose axes 1 and 2 are associated with Task 1 (TSKAX1, 2) and axes 3 and 4 are associated with Task 2 (TSKAX3, 4). If motion is in progress on axes 1 and 2, Task 1 is considered "in motion" and Task 1 cannot execute a command from the list below. However, while motion is in progress in Task 1 and not Task 2, Task 2 can execute these commands without encountering an error.

All of the commands in the table below, except for SCALE, are axis-dependent. That is, if one axis is moving you can change the parameters on the other axes, provided they are not in motion.

Command	Description
CMDDIR.....	Commanded Direction Polarity
DRES	Drive Resolution
DRIVE.....	Drive Shutdown
ENCPOL.....	Encoder Polarity
ERES	Encoder Resolution
FOLEN.....	Following Mode Enable
GOL	Initiate Linear Interpolated Motion
HOM	Go Home
HOMA	Home Acceleration
HOMAA.....	Average Home Acceleration
HOMAD.....	Home Deceleration
HOMADA.....	Average Home Deceleration
HOMV	Home Velocity
HOMVF.....	Home Final Velocity
JOG	Jog Mode Enable
JOGA	Jog Acceleration
JOGAA.....	Average Jog Acceleration
JOGAD.....	Jog Deceleration
JOGADA.....	Average Jog Deceleration
JOGVH.....	Jog Velocity High
JOGVL.....	Jog Velocity Low

Command	Description
JOY.....	Joystick Mode Enable
JOYA.....	Joystick Acceleration
JOYAA	Average Joystick Acceleration
JOYAD	Joystick Deceleration
JOYADA	Average Joystick Deceleration
JOYVH	Joystick Velocity High
JOYVL	Joystick Velocity Low
LHAD.....	Hard Limit Deceleration
LHADA	Average Hard Limit Deceleration
LSAD.....	Soft Limit Deceleration
LSADA	Average Soft Limit Deceleration
PSET.....	Establish Absolute Position
SCALE	Enable/Disable Scale Factors *
SCLA.....	Acceleration Scale Factor
SCLD.....	Distance Scale Factor
SCLV.....	Velocity Scale Factor

* If any axis is in motion, you will cause an error if you attempt to change this command's parameters.

Variables

6K Series controllers have four types of variables, each designated with a different command. All four types are automatically stored in non-volatile memory.

Type	Command	Quantity	Function
Numeric (real)	VAR	225	Store real numeric data (range is $\pm 999,999,999.99999999$). Can be used to perform mathematical (=, +, -, *, /, SQRT), trigonometric (ATAN, COS, PI, SIN, TAN), and Boolean (&, , ^, ~) operations. Can also be used to store ("teach") variable data in variable arrays (called <i>data programs</i>) and later use the stored data as a source for motion program parameters (see <i>Variable Arrays</i> on page 94 for details).
Integer	VARI	225	Store integer numeric data (range is $\pm 2,147,483,647$). Can be used to perform mathematical (=, +, -, *, /) and Boolean (&, , ^, ~) operations.
Binary	VARB	125	Store 32-bit binary or hexadecimal values. Can also store the binary status bits from status registers. Frequently used registers are: inputs (IN), outputs (OUT), limits (LIM), system (SS), Following (FS), axis (AS & ASX), and error (ER). For example, the VARB2=IN.3 command assigns the binary state of input 12 to binary variable 2. Also use to perform bitwise operations (&, , ^, ~, >>, <<).
String	VARS	25	Store message strings of 50 characters or less. These message strings can be predefined error messages, user messages, etc. The programming example in the <i>Command Value Substitutions</i> (page 7) demonstrates the use of a string variable. Enhancements as of OS revision 5.1.0: <ul style="list-style-type: none"> Copy one VARS variable to another VARS variable. VARS_n=VARS_m can be used, as well as variable substitutions for "n" and "m". VARS message string was increased from 20 to 50 characters.

NOTE: Variables do not share the same memory (e.g., VAR1, VARI1, VARB1, and VARS1 can all exist at the same time and operate separately).

Converting Between Binary and Numeric Variables

Using the Variable Type Conversion (VCVT) operator, you can convert numeric (VAR or VARI) values to binary (VARB) values, and vice versa. The operation is a signed operation as the binary value is interpreted as a two's complement number. Any *don't cares* (x) in a binary value is interpreted as a zero (∅).

If the mathematical statement's result is a numeric value, then VCVT converts binary values to numeric values. If the statement's result is a binary value, then VCVT converts numeric values to binary values.

Numeric to Binary

Example	Description/Response
VAR1=-5	Set numeric variable value = -5
VARB1=VCVT (VAR1)	Convert the numeric value to a binary value
VARB1	*VARB1=1101_1111_1111_1111_1111_1111_1111_1111

Binary to Numeric

Example	Description/Response
VARB1=b0010_0110_0000_0000_0000_0000_0000_0000	Set binary variable = +100.0
VAR1=VCVT (VARB1)	Convert binary value to numeric
VAR1	*VAR1=+100.0

Using Numeric (VAR and VARI) Variables

NOTES

- The examples below show the use of real numeric variables (VAR). Integer variables can be used in the same operations with these exceptions:
 - Values are truncated to nearest integer value
 - Operations using square root (SQRT) and trigonometric (ATAN, COS, PI, SIN, TAN) operators are not allowed
- Some numeric variable operations reduce precision. The following operations reduce the precision of the return value: Division and Trigonometric functions yield 5 decimal places; Square Root yields 3 decimal places; and Inverse Trigonometric functions yield 2 decimal places.

Mathematical Operations

The following examples demonstrate how to perform math operations with numeric variables. Operator precedence occurs from left to right (e.g., VAR1=1+1+1*3 sets VAR1 to 9, not 5).

Addition (+)

Example	Response
VAR1=5+5+5+5+5+5	
VAR1	*VAR1=35.0
VAR23=1000.565	
VAR11=VAR1+VAR23	
VAR11	*VAR11=+1035.565
VAR1=VAR1+5	
VAR1	*VAR1=+40.0

Subtraction (-)

Example	Response
VAR3=20-10	
VAR20=15.5	
VAR3=VAR3-VAR20	
VAR3	*VAR3=-5.5

Multiplication (*)

Example	Response
VAR3=10	
VAR3=VAR3*20	
VAR3	*VAR3=+200.0

Division (/)

Example	Response
VAR3=10	
VAR20=15.5	
VAR20	*+15.5
VAR3=VAR3/VAR20	
VAR3	*+0.64516
VAR30=75	
VAR30	*+75.0
VAR19=VAR30/VAR3	
VAR19	*+116.25023

Square Root (SQRT)

Example	Response
VAR3=75	
VAR20=25	
VAR3=SQRT (VAR3)	
VAR3	*+8.660
VAR20=SQRT (VAR20)+SQRT (9)	
VAR20	*+8.0

Trigonometric Operations

The examples below demonstrate how to perform trigonometric operations with numeric variables.

Sine

Example

```
RADIAN0
VAR1=SIN(0)
VAR1
VAR1=SIN(30)
VAR1
VAR1=SIN(45)
VAR1
VAR1=SIN(60)
VAR1
VAR1=SIN(90)
VAR1
RADIAN1
VAR1=SIN(0)
VAR1
VAR1=SIN(PI/6)
VAR1
VAR1=SIN(PI/4)
VAR1
VAR1=SIN(PI/3)
VAR1
VAR1=SIN(PI/2)
VAR1
```

Response

```
*VAR1=+0.0
*VAR1=+0.5
*VAR1=+0.70711
*VAR1=+0.86603
*VAR1=+1.0
*VAR1=+0.0
*VAR1=+0.5
*VAR1=+0.70711
*VAR1=+0.86603
*VAR1=+1.0
```

Cosine

Example

```
RADIAN0
VAR1=COS(0)
VAR1
VAR1=COS(30)
VAR1
VAR1=COS(45)
VAR1
VAR1=COS(60)
VAR1
VAR1=COS(90)
VAR1
RADIAN1
VAR1=COS(0)
VAR1
VAR1=COS(PI/6)
VAR1
VAR1=COS(PI/4)
VAR1
VAR1=COS(PI/3)
VAR1
VAR1=COS(PI/2)
VAR1
```

Response

```
*VAR1=+1.0
*VAR1=+0.86603
*VAR1=+0.70711
*VAR1=+0.5
*VAR1=+0.0
*VAR1=+1.0
*VAR1=+0.86603
*VAR1=+0.70711
*VAR1=+0.5
*VAR1=+0.0
```


Using Binary Variables

The following examples illustrate the 6K Series product's ability to perform bitwise functions with binary variables.

Storing binary values. The 6K Series Language allows you to store binary numbers in the binary variables (VARB) command. The binary variables start at the left with the least significant bit, and increase to the right. For example, to set bit 1, 5, and 7 you would issue the command VARB1=b1xxx1x1. Notice that the letter b is required. When assigning a binary variable, any bit set to “x” remains “x” until set to “1” or “0”. Any bit that is unspecified is set to “x”. To change, or check, one bit without affecting the others, use the bit-select operator (e.g., use VARB1.3-1 to set only bit 3 of VARB1).

Example VARB1=b1101XX1	Response *VARB1=1101_XX1X_XXXX_XXXX_XXXX_XXXX_XXXX
----------------------------------	--------------------------------------------------------------

Storing hexadecimal values. Hexadecimal values can also be stored in binary variables (VARB). The hexadecimal value must be specified the same as the binary value—left is least significant byte, right is most significant. For example, to set bit 1, 5, and 7 you would issue the command VARB1=h7FAD. Notice that the letter h is required. **NOTE:** When assigning a hexadecimal value to a binary variable, all unspecified bits are set to zero.

Example VARB1=h7FAD VARB1	Response *VARB1=1110_1111_0101_1011_0000_0000_0000_0000
----------------------------------------	-------------------------------------------------------------------

Bitwise And (&)	Example VARB1=b1101 VARB1 VARB1=VARB1 & bXXX1 1101 VARB1 VARB1=h0032 FDA1 & h1234 VARB1	Response *VARB1=1101_XXXX_XXXX_XXXX_XXXX_XXXX_XXXX_XXXX *VARB1=XX01_XX0X_XXXX_XXXX_XXXX_XXXX_XXXX_XXXX 43E9 *VARB1=0000_0000_1100_0000_0010_1000_0101_1000
-----------------	------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Bitwise Or ()	Example VARB1=h32FD VARB1 VARB1=VARB1 bXXX1 1101 VARB1 VARB1=h0032 FDA1 h1234 VARB1	Response *VARB1=1100_0100_1111_1011_0000_0000_0000_0000 *VARB1=11X1_1101_1111_1X11_XXXX_XXXX_XXXX_XXXX 43E9 *VARB1=1000_0100_1100_0110_1111_1111_0111_1001
----------------	------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Bitwise Exclusive Or (^)	Example VARB1=h32FD ^ bXXX1 1101 VARB1 VARB1=h0032 FDA1 ^ h1234 VARB1	Response *VARB1=XXX1_1001_XXXX_XXXX_XXXX_XXXX_XXXX_XXXX 43E9 *VARB1=1000_0100_0000_0110_1101_0111_0010_0001
--------------------------	------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------

Bitwise Not (~)	Example VARB1=~(h32FD) VARB1 VARB1=~(b1010 XX11 0101) VARB1	Response *VARB1=0011_1011_0000_0100_1111_1111_1111_1111 *VARB1=0101_XX00_1010_XXXX_XXXX_XXXX_XXXX_XXXX
-----------------	--------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------

Shift Left to Right (>>)	Example VARB1=h32FD >> h4 VARB1 VARB1=b1010 XX11 0101 >> b11 VARB1	Response *VARB1=0000_1100_0100_1111_1011_0000_0000_0000 b11 *VARB1=0001_010X_X110_101X_XXXX_XXXX_XXXX_XXXX
--------------------------	---------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------

Shift Right to Left (<<)	Example VARB1=h32FD << h4 VARB1 VARB1=b1010 XX11 0101 << b11 VARB1	Response *VARB1=0100_1111_1011_0000_0000_0000_0000_0000 b11 *VARB1=0XX1_1010_1XXX_XXXX_XXXX_XXXX_XXXX_X000
--------------------------	---------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------

Program Flow Control

Program flow refers to the order in which commands will be executed, and when or whether they will be executed at all. In general, commands are executed in the order in which they are received. However, certain commands can redirect the order in which commands will be processed.

You can affect program flow with:

- Unconditional Loops and Branches
- Conditional Loops and Branches

Unconditional Looping and Branching

Unconditional Looping

The Loop (L) command is an unconditional looping command. You can use this command to repeat the execution of a group of commands for a predetermined number of iterations. You can nest Loop commands up to 16 levels deep. The code sample (portion of a program) below demonstrates a loop of 5 iterations.

```
MA0      ; Sets unit to Incremental mode
A50      ; Sets acceleration to 50
V5       ; Sets velocity to 5
L5       ; Loops 5 times
D2000    ; Sets distance to 2,000
GO1      ; Executes the move (Go)
T2       ; Delays 2 seconds after the move
LN       ; Ends loop
```

Unconditional Branching

When an unconditional branch is processed, the flow of program execution (“control”) passes to the program or label specified in the branch command. Depending on the branch command used, processing may or may not return to the original program (the “calling” program). There are three ways to branch unconditionally:

GOSUB: The GOSUB command branches to the program name or label stated in the GOSUB command. After the called program or label is executed, processing returns to the calling program at the next command line after the GOSUB branch command.

GOTO: The GOTO command branches to the program name or label stated in the GOTO command. After the called program or label is executed, processing **does not** return to the calling program—instead, the program will end. This holds true unless the subroutine in which the GOTO resides was called with a GOSUB by another program; in this case, the END in the GOTO program will initiate a return to the calling program. For example, if processing flows from a GOSUB in program A to program B, and then a GOTO from program B to program C, when the END command is processed in program C, processing returns to program A at the command line after the GOSUB.

JUMP: The JUMP command branches to the program name or label stated in the JUMP command. All nested IFs, WHILEs, and REPEATs, loops (L), and subroutines are cleared; thus, the program or label that the JUMP initiates **will not** return control to the calling program; instead, the called program will end.

If an invalid program or label name is entered, the branch command will be ignored and processing will continue with the next line in the program.

6K Series products do not support recursive calling of subroutines.

Using labels: Labels, defined with the \$ command, provide a method of branching to specific locations within the same program. Labels can only be defined within a program and executed with a GOTO, GOSUB, or JUMP command from within the same program (see *Example B below*).

NOTE

Be careful about performing a GOTO within a loop or branch statement area (i.e., between L & LN, between IF & NIF, between REPEAT & UNTIL, or between WHILE & NWHILE). Branching to a different location within the same program will cause the next L, IF, REPEAT, or WHILE statement encountered to be nested within the previous L, IF, REPEAT, or WHILE statement area, unless an LN, NIF, UNTIL, or NWHILE command has already been encountered.

** To avoid this nesting situation, use the JUMP command instead of the GOTO command.

Example A

DESCRIPTION: The program `cut1` is executed until it gets to the command `GOSUB prompt`. From there it branches unconditionally to the subroutine (actually a program) called `prompt`. The subroutine `prompt` queries the operator for the number of parts to process. After the part number is entered (e.g., operator enters the `! ' 12` command to process 12 parts), the rest of the `prompt` subroutine is executed and control goes back to the `cut1` program and resumes program execution with the next command after the `GOSUB`, which is `MA00`.

```
DEL cut1                ; Delete a program before defining it
DEF cut1                ; Begin definition of program cut1
HOM11                  ; Send axes 1 and 2 to the home position
WAIT(1AS=b0XXX1 AND 2AS=b0XXX1) ; Wait for axes 1 and 2 to come
                        ; to a halt at home
GOSUB prompt           ; Go to subroutine program called prompt
MA00                   ; Place axes 1 and 2 in the incremental mode
A10,30                 ; Set acceleration: axis 1 = 10, axis 2 = 30
AD5,12                 ; Set deceleration: axis 1 = 5, axis 2 = 12
V5,8                   ; Set velocity: axis 1 = 5, axis 2 = 8
D16000,100000         ; Set distance: axis 1 = 16,000; axis 2 = 100,000
OUT.3-1                ; Turn on onboard output number 3
T5                     ; Wait for 5 seconds
L(VAR2)                ; Begin loop (number of loops = value of VAR2)
  GO11                  ; Initiate moves on axes 1 and 2
  T3                    ; Wait for 3 seconds
LN                      ; End loop
OUT.3-0                ; Turn off onboard output number 3
END                     ; End definition of program cut1

DEF prompt              ; Begin definition of program prompt
VAR1="Enter part count >" ; Place message in string variable 1
VAR2=READ1              ; Prompt operator with string variable 1,
                        ; and read data into numeric variable 2
                        ; NOTE: Type !' before the part count number.
END                     ; End definition of program prompt
```

Example B

DESCRIPTION: This example demonstrates the use of labels (\$).

```
DEL pick                ; Delete a program before defining it
DEF pick                ; Begin definition of program pick
GO1100                  ; Initiate motion and axes 1 and 2
IF(VAR1=5)              ; If variable 1 = 5, then execute commands
                        ; between IF and ELSE. Otherwise, execute
                        ; commands between ELSE and NIF
  GOTO pick1            ; Goto label pick1
  ELSE                  ; Else part of IF statement
    GOTO pick2          ; Goto label pick2
NIF                      ; End of IF statement
$ pick1                 ; Define label for pick1
GO0011                  ; Initiate motion on axes 3 and 4
BREAK                   ; Break out of current subroutine or program
$ pick2                 ; Define label for pick2
GO1001                  ; Initiate motion on axes 1 and 4
END                     ; End definition of program pick
```

Conditional Looping and Branching

Conditional looping (REPEAT/UNTIL and WHILE/NWHILE) entails repeating a set of commands until or while a certain condition exists. In *conditional branching* (IF/ELSE/NIF), a specific set of commands is executed based on a certain condition. Both rely on the fulfillment of a conditional *expression*, a condition specified in the UNTIL, WHILE, or IF commands.

A wait command pauses command execution until a specific condition exists.

Flow Control Expression Examples

This section provides examples of expressions that can be used in conditional branching and looping commands (UNTIL, WHILE, and IF) and the wait command. These expressions can be constructed, in conjunction with relational and logical operators, with the following operands:

- Numeric variables and binary variables
- Inputs and outputs
- Current motion parameters and status
- Current commanded and actual position
- Error, axis, and system status
- Timer value
- Data read from the serial port
- Data read from the RP240
- Following conditions
- Multi-tasking conditions

Numeric and Binary Variables

A numeric variable (VAR or VARI) can be used within an expression if it is compared against another numeric variable, a value, or one of the comparison commands (see list on page 7). When comparing a variable against another value, variable, or comparison command, the relational operators (=, >, >=, <, <=, <>) and logical operators (AND, OR, NOT) are used.

Expression

(VAR1<VAR2)
 (VAR1>=2500)
 (VAR1=1AD)
 (VAR1<VAR2 AND VAR4>1PE)

Description

True expression if variable 1 is less than variable 2
 True expression if variable 1 is greater than or equal to 2500
 True expression if variable 1 is equal to the decel of axis 1
 True expression if variable 1 is less than variable 2 and variable 4 is greater than the value of encoder 1

A binary variable (VARB) can be used within an expression, if the variable is compared against another binary variable, or a value. When comparing a variable against another value or variable, the relational operators (=, >, >=, <, <=, <>) and logical operators (AND, OR, NOT) are used.

Expression

(VARB1<>VARB2)
 (VARB1=b1101 X111)
 (VARB1<VARB2 AND VARB4>hF)

Description

True expression if binary variable 1 is not equal to binary variable 2
 True expression if binary variable 1 is equal to 1101 X111
 True expression if binary variable 1 is less than binary variable 2 and binary variable 4 is greater than the hexadecimal value of F

Inputs and Outputs

An input or output operand (ANI, ANO, IN, INO, LIM, OUT, TRIG) can be used within an expression, if the operand is compared against a binary variable or a binary or hexadecimal value. When making the comparison, the relational operators (=, >, >=, <, <=, <>) and logical operators (AND, OR, NOT) are used.

Expression Description

(IN.3=b1) True expression if onboard input 3 is equal to 1
 (LIM>h3) True expression if limit status is greater than hexadecimal 3

Current Motion Parameters and Status

Motion parameters consist of A, AD, D, V, VEL, status MOV. The motion parameters can be used within an expression, if the operand is compared against a numeric variable or value. The motion status operand must be compared against a binary variable or a binary or hexadecimal value. When making the comparison, the relational operators (=, >, >=, <, <=, <>) and logical operators (AND, OR, NOT) are used. (Following conditions are addressed below.)

Expression	Description
(VAR1<1VEL)	True expression if the value of variable 1 is less than the commanded velocity of axis 1
(1AD=25000)	True expression if axis 1 deceleration equals 25000
(MOV=b00)	True expression if moving status equals 00 (axes 1 & 2 are not moving)

Current Commanded & Actual Position

The current commanded and actual positions (ANI, DAC, FB, PANI, PC, PCC, PCE, PCME, PCMS, PE, PER, PE, PMAS, PME, PSHF, PSLV) can be used within an expression, if the operand is compared against a numeric variable or value. When making the comparison, the relational operators (=, >, >=, <, <=, <>) and logical operators (AND, OR, NOT) are used.

Expression	Description
(VAR1<1FB)	True expression if the value of variable 1 is less than the actual position (position of the assigned feedback device) of axis 1
(2PC=4000)	True expression if axis 2 commanded position equals 4000
(VAR1<1PME)	True expression if VAR1 is < master encoder position of axis 1
(2PE=25000)	True expression if axis 2 encoder position equals 25000

Error, Axis, and System Status

The error status, axis status, and system status operands (ER, AS, ASX, SS) can be used within an expression, if the operand is compared against a binary variable or a binary or hexadecimal value. When making the comparison, the relational operators (=, >, >=, <, <=, <>) and logical operators (AND, OR, NOT) are used. Refer to page 226 for a list of status bit functions.

Expression	Description
(ER.12=b1)	True expression if error status bit 12 is equal to 1
(AS=h3FFD)	True expression if axis status is equal to hexadecimal 3FFD

Timer Value

The current timer value (TIM) can be used within an expression, if the operand is compared against a numeric variable or value. When making the comparison, the relational operators (=, >, >=, <, <=, <>) and logical operators (AND, OR, NOT) are used.

Expression	Description
(VAR1<TIM)	True expression if the value of variable 1 is less than the timer value

Data Read from the Communications Port

The READ command can be used to input data from a serial port or the Ethernet port into a numeric variable. After the data has been read into a numeric variable, that variable can be used in an expression.

Example	Description
VAR8="ENTER DATA"	Define message (string variable 8)
VAR2=READ8	Send message (string variable 8) and then wait for immediate data to be read (into numeric variable 2)
!'88.3	Immediate data input (must type '!' before the numeric value)
IF (VAR2<=100)	Evaluate expression to see if data read is < or equal to 100
.	
NIF	End of IF

Data Read from the RP240

The DREAD and DREADF commands can be used to input data from the RP240 into a numeric variable. DREAD reads a number from the RP240's numeric keypad. DREADF reads a number representing a RP240 function key. After the data has been read into a numeric variable, that variable can be used in an expression. The DKEY operator allows you to read the current state of the RP240 keypad (each key has a unique numeric value).

DCLEAR0	; Clear RP240 display
DWRITE"HIT F4"	; Send message to RP240 display
VAR3=DREADF	; Read data from a RP240 function key into numeric variable 3
IF (DKEY=24)	; Evaluate expression to see if function key F4 was hit
DCLEAR2	; Clear RP240 display line 2
DWRITE"TRY AGAIN"	; Send message to RP240 display
NIF	; End of IF

The DREADI1 command allows continual numeric or function key data entry from the RP240 (when used in conjunction with the DREAD and/or DREADF commands). In this immediate mode, program execution is not paused (waiting for data entry) when a DREAD or DREADF command is encountered. Refer to the DREAD and DREADF command descriptions for programming examples.

NOTES

- While in the Data Read Immediate Mode, data is read into numeric variables only (VAR).
- This feature is not designed to be used in conjunction with the RP240's standard menus; the RUN, JOG, and DJOG menus will disable the DREADI mode.
- Do not assign the same variable to read numeric data and function key data—pick only one.

Following Conditions

These Following conditions are available for conditional expressions: Axis status bit 26 (AS.26), Error status bit 14 (ER.14), Following status (FS), NMCY, PCME, PCMS, PMAS, PME, PSHF, PSLV, and VMAS.

Expression	Description
(2AS.26=b1)	True if a new motion profile on axis 2 is waiting for the GOWHEN condition to be true or a TRGFNC1xxxxxxx trigger.
(1ER.14=b1)	True if the GOWHEN condition on axis 1 is already true when the subsequent GO, GOL, FSHFC, or FSHFD command is executed.
(3FS.7=b0)	True if the master for follower axis 3 is in motion.
(2NMCY>200)	True if the master for axis 2 has moved through 200 cycles.
(1PMAS>12)	True if the master for axis 1 has traveled more than 12 units.
(1PSHF>1.5)	True if follower axis 1 has shifted more than 1.5 units.
(3PSLV>24)	True if follower axis 3's commanded position is more than 24 units.
(1VMAS<2)	True if the velocity of the master for axis 1 is less than 2 units/sec.

*Multi-Tasking
Conditions*

These Multi-tasking conditions are available for conditional expressions: Status of which tasks are current active (SWAP), identity of the task in which the command is executed (TASK).

Expression	Description
(SWAP.3=b1)	True if Task 3 is active.
(TASK=3)	True if the task executing the conditional expression is Task 3.

Conditional Looping

The 6K controller supports two conditional looping structures—REPEAT/UNTIL and WHILE/NWHILE.

All commands between REPEAT and UNTIL are repeated until the expression contained within the parenthesis of the UNTIL command is true. The example below illustrates how a typical REPEAT/UNTIL conditional loop works. In this example, the REPEAT loop will execute 1 time, at which point the expression stated within the UNTIL command will be evaluated. If the expression is true, command processing will continue with the first command following the UNTIL command. If the expression is false, the REPEAT loop will be repeated.

```

VAR5=0           ; Initializes variable 5 to 0
DEL prog10      ; Delete a program before defining it
DEF prog10      ; Defines program prog10
INFNC1-A        ; Assign onboard input 1 as g.p. input for use with IN
INFNC2-A        ; Assign onboard input 2 as g.p. input for use with IN
INFNC3-A        ; Assign onboard input 3 as g.p. input for use with IN
INFNC4-A        ; Assign onboard input 4 as g.p. input for use with IN
OUTFNC1-A       ; Assign onboard output 1 is a general-purpose output
A50             ; Acceleration is 50
AD50            ; Deceleration is 50
V5             ; Sets velocity to 5
D25000         ; Distance is 25,000
REPEAT         ; Begins the REPEAT loop
  GO1           ; Executes the move (Go)
  VAR5=VAR5+1   ; Variable 5 counts up from 0
UNTIL(IN=b1110 OR VAR5>10) ; When inputs 1-4 are 1110, respectively, or

```

```

                                ; VAR5 is greater than 10, the loop will stop.
OUT1                             ; Turn on output 1 when finished with REPEAT loop
END                               ; End program definition
RUN prog10                        ; Initiate program prog10

```

All commands between WHILE and NWHILE are repeated as long as the WHILE condition is true. The following example illustrates how a typical WHILE/NWHILE conditional loop works. In this example, the WHILE loop will execute if the expression is true. If the expression is false, the WHILE loop will not execute.

```

VAR5=0                            ; Initializes variable 5 to 0
DEL prog10                        ; Delete a program before defining it
DEF prog10                        ; Defines program prog10
INFNC1-A                          ; Assign onboard input 1 as g.p. input for use with IN
INFNC2-A                          ; Assign onboard input 2 as g.p. input for use with IN
INFNC3-A                          ; Assign onboard input 3 as g.p. input for use with IN
INFNC4-A                          ; Assign onboard input 4 as g.p. input for use with IN
OUTFNC1-A                         ; Assign onboard output 1 is a general-purpose output
A50                               ; Acceleration is 50
AD50                              ; Deceleration is 50
V5                                ; Sets velocity to 5
D25000                           ; Distance is 25,000
WHILE(IN=b1110 OR VAR5>10)       ; While the inputs 1-4 are 1110, respectively
                                ; or VAR5 is greater than 10, the loop will continue.
    GO1                           ; Executes the move (Go)
    VAR5=VAR5+1                  ; Variable 5 counts up from 0
NWHILE                            ; End WHILE command
OUT1                             ; Turn on output 1 when finished with WHILE loop
END                               ; End program definition

; *****
; * To run prog10, execute the "RUN prog10" command *
; *****

```

Conditional Branching

You can use the IF command for conditional branching. All commands between IF and ELSE are executed if the expression contained within the parentheses of the IF command is true. If the expression is false, the commands between ELSE and NIF are executed. If the ELSE is not needed, it can be omitted. The commands between IF and NIF are executed if the expression is true. Examples of these commands are as follows.

```

DEL prog10                        ; Delete a program before defining it
DEF prog10                        ; Defines program prog10
INFNC1-A                          ; Assign onboard input 1 as g.p. input for use with IN
INFNC2-A                          ; Assign onboard input 2 as g.p. input for use with IN
INFNC3-A                          ; Assign onboard input 3 as g.p. input for use with IN
INFNC4-A                          ; Assign onboard input 4 as g.p. input for use with IN
A50                               ; Acceleration is 50
AD50                              ; Deceleration is 50
V5                                ; Sets velocity to 5
IF (VAR1>0)                       ; IF variable 1 is greater than zero
    D25000                        ; Distance is 25,000
    ELSE                          ; Else
    D50000                        ; Distance is 50,000
NIF                                ; End if command
IF(IN=b1110)                      ; If onboard inputs 1-4 are 1110, initiate axis 1 move
    GO1                           ; Executes the move (Go)
NIF                                ; End IF command
END                               ; End program definition

; *****
; * To run prog10, execute the "RUN prog10" command *
; *****

```

Program Interrupts (*ON Conditions*)

Multi-Tasking
 Each task has its own ONP program and its own set of ON conditions.

While executing a program, the 6K controller can interrupt the program based on several possible *ON conditions*: programmable input(s) status, user status, or the value of numeric variables 1 or 2. These ON conditions are enabled with the ONCOND command, and are defined with the commands listed below. After the ON conditions are enabled (with the ONCOND command), an ON condition interrupt can occur at any point in program execution. When an ON condition occurs, the controller performs a GOSUB to the program assigned as the ON program and then passes control back to the original program and resumes command execution at the command line from which the interruption occurred.

Within the ON program, the programmer is responsible for checking which ON condition caused the branch (if multiple ON conditions are enabled with the ONCOND command). Once a branch to the ON program occurs, the ON program will not be called again until after it has finished executing. After returning from the ON program, the condition that caused the branch must evaluate false before another branch to the ON program will be allowed.

SETUP FOR PROGRAM INTERRUPT (see programming example below)

1. Define a program to be used as the ON program to which the controller will GOSUB when an ON condition evaluates true.
2. Use the ONP command to assign the program as the ON program.
3. Use the ONCOND command to enable the ON conditions that you are using. The syntax for the ONCOND command is ONCOND, where the first is for the ONIN condition, the second for ONUS, the third for ONVARA, and the fourth for ONVARB.

ON conditions:

- ONIN Specify an input bit pattern that will cause a GOSUB to the program assigned as the ON program (see programming example below).
- ONUS Specify an user status bit pattern that will cause a GOSUB to the ON program. The user status bits are defined with the INDUST command.
- ONVARA Specify the range of numeric variable 1 (VAR1) that will cause a GOSUB to the ON program. For example, ONVARAØ, 2Ø establishes the condition that if the value of VAR1 is ≤0 or ≥20, the ON program will be called.
- ONVARB This is the same function as ONVARA, but for numeric variable 2 (VAR2)

Programming Example: Configures the controller to increment variable 1 when input 1 goes active. If input 1 does go active, control will be passed (GOSUB) to the ON program (`onjump`), the commands within the ON program will be executed, and control will then be passed back to the original program.

```

DEF onjump       ; Begin definition of program onjump

VAR1=VAR1+1     ; Increment variable 1
END             ; End definition of program onjump

VAR1=0          ; Initialize variable 1
ONIN1           ; When input 1 becomes active, branch to the ON program
ONP onjump      ; Assign the onjump program as the ON program
ONCOND1000      ; Enable only the ONIN function. Disable the ONUS,
                 ; ONVARA, and ONVARB functions, respectively
  
```

Situations in which ON conditions will not interrupt immediately

These are situations in which an ON condition does not immediately interrupt the program in progress. However, the fact that the ON condition evaluated true is retained, and when the condition listed below is no longer preventing the interrupt, the interrupt will occur.

- While motion is in progress due to GO, GOL, GOWHEN, HOM, JOY, JOG, or PRUN and the continuous command execution mode is disabled (COMEXCØ).
- While a WAIT statement is in progress
- While a time delay (T) is in progress
- While a program is being defined (DEF)
- While a pause (PS) is in progress
- While a data read (DREAD, DREADI, DREADF, or READ) is in progress

Error Handling

USE MOTION PLANNER

Motion Planner's editor provides an error handling wizard.

DEBUG TOOLS

For information on program debug tools, see page 225.

The 6K Series products have the ability to detect and recover the following error conditions:

- Error bit 1 Stepper axes only: Stall detected on any axis
- Error bit 2 Hardware end-of-travel limit encountered on any axis
- Error bit 3 Software end-of-travel limit encountered on any axis
- Error bit 4 Drive fault input activated any axis
- Error bit 5 Commanded kill or stop
- Error bit 6 Kill input activated
- Error bit 7 User fault input activated
- Error bit 8 Stop input activated
- Error bit 9 ENABLE input not grounded
- Error bit 10 Profile for a pre-emptive GO or a registration move is not possible
- Error bit 11 Servo Axes Only: Target zone settling timeout
- Error bit 12 Servo Axes Only: Allowable position error (SMPER) exceeded
- Error bit 14 GOWHEN condition already true when the subsequent GO, GOL, FSHFC, or FSHFD command was executed
- Error bit 16 Bad command is detected
- Error bit 17 Encoder failure is detected, if EFAIL1 mode is enabled
- Error bit 18 Expansion I/O brick cable is disconnected or powered down
- Error bit 22 Ethernet failed due to hardware disconnect or COM6SRVR server failure.

Enabling Error Checking

To detect and respond to the error conditions noted above, the corresponding error-checking bit(s) must be enabled with the `ERROR` command (see the *ERROR Bit* column in the table below). If an error condition occurs and the associated error-checking bit has been enabled with the `ERROR` command, the 6K controller will branch to the error program.

For example, if you wish the 6K controller to branch to the error program when a hardware end-of-travel limit is encountered (error bit 2) or when a drive fault occurs (error bit 4), you would issue the `ERRORØ1Ø1` command to enable error-checking bits 2 and 4.

MULTI-TASKING

If you are operating multiple tasks, be aware that you must enable error conditions (`ERROR`) and specify an error program (`ERRORP`) for each task (e.g., `2%ERROR.2-1` and `2%ERRORP FIX` for Task 2). Each task has its own error status register (reported with `ER`, `TER`, and `TERF`). Regarding axis-related error conditions (e.g., drive fault, end-of-travel limit, etc.), only errors on the task's associated (`TSKAX`) axes will cause a branch to the task's `ERRORP` program.

⇒ **Hint:** Within the structure of your error program, you can use the `IF` and `ER` commands to check which error caused the call to the `ERRORP` program and respond accordingly.

Defining the Error Program

The purpose of the error program is to provide a programmed response to certain error conditions (see list above) that can occur during the operation of your system. Programmed responses typically include actions such as shutting down the drive(s), activating or deactivating outputs, etc. Refer to the error program set-up example below.

Using the `ERRORP` command, you can assign any previously defined program as the error program. For example, to assign a previously defined program named `CRASH` as the error program, enter the `ERRORP CRASH` command. To un-assign a program from being the error program, issue the `ERRORP CLR` command (e.g., as in this example, it does not delete the `CRASH` program, but merely unlinks it from its assignment as the error program).

Canceling the Branch to the Error Program

NOTE: In addition to canceling the branch to the error program, you must also remedy the cause of the error; otherwise, the error program will be called again when you resume operation. Refer to the *How to Remedy the Error* column in the table below for details.

If an error condition occurs and the associated error-checking bit has been enabled with the ERROR command, the 6K controller will branch to the error program. The error program will be continuously called/repeated until you cancel the branch to the error program. (This is true for all cases except error condition number 9, ENABLE input activated, in which case the error program is called only once.)

There are three options for canceling the branch to the error program:

- Disable the error-checking bit with the ERROR.n-Ø command, where "n" is the number of the error-checking bit you wish to disable. For example, to disable error checking for the kill input activation (bit 6), issue the ERROR.6-Ø command. To re-enable the error-checking bit, issue the ERROR.n-1 command.
- Delete the program assigned as the ERRORP program (DEL <name of program>).
- Satisfy the *How to Remedy the Error* requirement identified in the table below.

ERROR Bit	Cause of the Error	Branch Type to Error Program	How to Remedy the Error
1	Stepper axes only: Stall detected (Stall Detection and Kill On Stall must be enabled first—see ESTALL and ESK, respectively).	Gosub	Issue a GO command.
2	Hard Limit Hit. (hard limits must be enabled first—see LH)	If COMEXLØ, then Goto; If COMEXL1, then Gosub	Change direction & issue GO command on the axis that hit the limit; or issue LHØ.
3	Soft Limit Hit. (soft limits must be enabled first—see LS)	If COMEXLØ, then Goto; If COMEXL1, then Gosub	Change direction & issue GO command on the axis that hit the limit; or issue LSØ.
4	Drive Fault (Detected only if you enable drive, DRIVE1, and drive fault input, DRFEN, and set correct drive fault level, DRFLVL; See page 46.)	Goto	Clear the fault condition at the drive, & issue a DRIVE1 command for the faulted axis.
5	Commanded Stop or Kill (whenever a K, !K, <ctrl>K, K, or !S command is sent). <div style="border: 1px solid black; padding: 2px; width: fit-content; margin: 5px auto;">See note below entitled "Commanded Kill or Stop".</div>	If !K, then Goto; If !S & COMEXSØ, then Goto; If !S & COMEXS1, then Gosub, but need !C	No fault condition is present—there is no error to clear. If you want the program to stop, you must issue the !HALT command.
6	Kill Input Activated (see INFNCi-C or LIMFNCi-C)	Goto	Deactivate the kill input.
7	User Fault Input Activated (see INFNCi-F or LIMFNCi-F).	Goto	Deactivate the user fault input, or disable it by assigning it a different function.
8	Stop input activated (see INFNCi-D or LIMFNCi-C).	Goto	Deactivate the stop input, or disable it by assigning it a different function.
9	ENABLE input not grounded. (see "ESTOP" below)	Goto	Re-ground ENABLE input, and issue @DRIVE1.
10	Profile for pre-emptive GO or registration move not possible at the time of attempted execution.	Gosub	Issue another GO command.
11	Servo axes only: Target Zone Timeout (STRGTT value has been exceeded).	Gosub	Issue these commands in this order: STRGTEØ, DØ, GO, STRGTE1
12	Servo axes only: Exceeded Maximum Allowable Position Error (set with the SMPER command).	Gosub	Issue a DRIVE1 command to the axis that exceeded the allowable position error. Verify that feedback device is working properly.
14	GOWHEN condition was already true when the subsequent GO, GOL, FGADV, FSHFC, or FSHFD command was executed.	Goto	Issue another GOWHEN command; or issue a !K command and check the program logic (use the TRACE and STEP features if necessary).
16	Bad command was detected.	Gosub	Issue the TCMDEP command to I.D. the command.
17	Encoder failure detected (EFAIL1 must be enabled before this error can be detected).	Gosub	Reconnect the encoder while the axis is in the EFAIL1 mode.
18	Cable to an expansion I/O brick is disconnected, or power to the I/O brick is lost.	Goto	Reconnect I/O brick cable. Issue the ERROR.18-0 command and then the ERROR.18-1 command.
22	Ethernet connection failed (hardware disconnect or COM6SRVR communications server failure).	Gosub	Clear the error bit (ERROR.22-0), re-establish the Ethernet connection, and then issue ERROR.22-1.

Reserved Bits: Bits 13, 15, 19-21 and 23-32.

Branching Types: If the error condition calls for a GOSUB, then after the ERRORP program is executed, program control returns to the point at which the error occurred. To prevent a return to the point at which the error occurred, use the HALT command to end program execution or use the GOTO command to go to a different program. If the error condition calls for a GOTO, there is no way to return to the point at which the error occurred.

Commanded Kill or Stop: When error bit 5 is enabled (ERROR.5-1), a Stop (S or !S) or a Kill (K, !K or <ctrl>K) command will cause the controller to branch to the error program. Note, however, that this error condition does not set an error bit (ER), because there is no way to clear the error condition upon leaving the error program. Therefore, you should use the IF (ER=b00000000000000000000000000000000) statement in your error program to determine if the cause of the error was a commanded kill or stop (i.e., if no error bits are set).

If ESTOP (ENABLE input) also cuts power to drives: This note is for systems in which power to the drives (not the 6K) is cut if the ENABLE input is opened. If you enable ERROR bit 9: When the ENABLE input is opened (and power is cut to the drives), the ERRORP program is called. Because the drives have lost power, the 6K will detect drive faults, which in turn kills the ERRORP program. If the ENABLE input is still ungrounded, the ERRORP program will again be called and then killed because of the drive fault condition (causing an endless loop). The resolution is to place the @DRFEN0 command in the ERRORP program so that it can disable checking the drive fault input and thereby circumvent the endless loop of calling the ERRORP program. Be sure to later re-enable drive fault checking with @DRFEN1 after the ENABLE input is re-grounded.

Error Program Set-up Example

The following is an example of how to set up an error program. This particular example is for handling the occurrence of a user fault.

Step 1 Create a program file (in Motion Planner's Editor module) to set up the error program:

```
; *****  
; * Assign the user fault input function to onboard trigger input 1. *  
; * The purpose of the user fault input is to detect the occurrence *  
; * of a fault external to the 6K controller and the motor/drive. *  
; * This input will generate an error condition. *  
; *****  
INFNC1-F ; Define onboard trigger input 1 as a user fault input  
  
; *****  
; * Define a program to respond to the user fault (call the program *  
; * fault), and then assign that program as the error program. The *  
; * purpose of the fault program is to display a message to inform *  
; * the operator that the user fault input has been activated. *  
; *****  
DEL fault ; Delete a program before defining it (a precaution)  
DEF fault ; Begin definition of program fault  
IF(ER.7=b1) ; Check if error bit 7 equals 1  
; (which means the user fault input has been activated)  
WRITE"FAULT INPUT\10\13" ; Send the message FAULT INPUT  
T3 ; Wait 3 seconds  
NIF ; End IF command  
END ; End definition of program fault  
ERRORP fault ; Assign the program called fault as the error program  
  
; *****  
; * Enable the user fault error-checking bit by putting a "1" in *  
; * the seventh bit of the ERROR command. After enabling this *  
; * error-checking bit, the controller will branch to the error *  
; * program whenever the user fault input is activated. *  
; *****  
ERROR0000001 ; Branch to error program upon user fault input (As an  
; alternative to the ERROR0000001 command, you could also  
; enable bit 7 by issuing the ERROR.7-1 command.)
```

Step 2 Save the program file in the Editor module. Then, using the Terminal module, download the program file to the 6K controller.

Step 3 Test the error handling:

1. While in the terminal emulator, enter these four commands:


```

L                               ; Loop command
WRITE"IN LOOP\10\13"           ; Display the message "IN LOOP"
T2                               ; Wait 2 seconds
LN                               ; End the loop ("IN LOOP" will be displayed
                               ; once every 2 seconds)
      
```
2. While the loop (IN LOOP) is executing in the terminal emulator, enter the !INEN1 command. The !INEN1 command disables input 1 and forces it on for testing purposes. This simulates the physical activation of input 1. (Since the error program is called continuously until the branch to the error program is canceled, the message FAULT INPUT will be repeatedly displayed once every 3 seconds.)
3. While the FAULT INPUT loop is executing in the terminal emulator, enter the !INENE command. The !INENE command re-enables input 1. The message In loop will not be displayed again, because the user fault input error is a GOTO branch (not a GOSUB branch) to the error program.

Non-Volatile Memory

The items listed below are automatically stored in the 6K product's non-volatile memory (battery-backed RAM). Cycling power or issuing a RESET command will not affect these settings.

- Power-up program (STARTP)
- Programs (defined with DEF & END)
- Compiled profiles and PLC programs (PCOMP). Compiled contours and PLC programs are always saved in the Compiled portion of battery-backed RAM. However, compiled individual axis profiles (GOBUF profiles) are removed from Compiled memory if you run them with the PRUN command and later cycle power or reset the controller (you will have to re-compile them with the PCOMP command).
- Memory allocation (MEMORY)
- Axis type definition (AXSDEF)
- Variables: VAR, VARI, VARB, and VARS
- Scaling: SCALE, SCLA, SCLD, SCLV, SCLMAS
- Commanded direction polarity (CMDDIR)
- Encoder polarity (ENCPOL)
- Device address for RS-232 or RS-485 serial communication (ADDR)
- Baud rate for RS-232 or RS-485 serial communication (BAUD)
- Ethernet IP address (NTADDR)
- Ethernet network mask (NTMASK)
- RP240 check and serial port functionality (DRPCHK)
- RP240 password (DPASS)
- Servo gain sets (SGSET)

A checksum is calculated for the non-volatile memory area each time you power up or reset your 6K controller. A bad checksum indicates that the user memory has been corrupted (possibly due to electrical noise) or has been cleared (due to a spent battery). The controller will clear all user memory when a bad checksum is calculated on power up or reset, and bit 22 will be set in the TSS command response.

System Performance

Several commands (listed below), when enabled, will slow command processing. This degradation in performance will not be noticeable for most applications. But for some, it can be necessary to disable one or all of these commands.

- SCALE (enable/disable scaling)
- INDUSE (enable/disable user status updates)
- INFNC (trigger and extended input functions; excluding functions “A” and “H”)
- LIMFNC (limit input functions; excluding functions “A”, “R”, “S”, and “T”)
- OUTFNC (digital output functions; excluding function “A”)
- ONCOND (enable/disable ON conditions)

Servo Update Performance Slowdown

If the 6K displays the “SYSTEM UPDATE OVERRUN, USE SYSPER4” error message, execute the SYSPER4 command to change the 6K’s system update period. For details, see the SYSPER command description in the *Command Reference*.