SPiiPlus NT
Programmer's Guide

Version NT 1.0

SPiiPlus NT Programmer's Guide

Version NT 1.0, 30 December 2010

Web Site: **www.AcsMotionControl.com**
Information: **info@AcsMotionControl.com**
Tech Support: **support@AcsMotionControl.com**

ACS Motion Control, Ltd.
Ramat Gabriel Industrial Park
POB 5668
Migdal HaEmek, 10500
ISRAEL
Tel: (972) (4) 6546440
Fax: (972) (4) 6546443

ACS Motion Control, Inc.
6575 City West Parkway
Eden Prairie, MN 55344
USA
Tel: (1) (763) 559-7669
(800-545-2980 in USA)
Fax: (1) (763) 559-0110

ACS Motion Control (Korea)
Digital Empire Building D-191
980-3, Youngtong-dong, Youngtong-gu,
Suwon, Geonggi-do, 443-813, Korea
Tel: +82-31-202-3541
Fax: +82-31-202-3542

Changes in Version NT 1.0

Page	Change
	Entire document updated to reflect the new NT line of motion controllers.
192	Added procedure for setting PEG to General Purpose outputs.

Table of Contents

List of Figures

List of Tables

1 Introduction

This guide provides a general overview for programming the SPiiPlus™ motion controller products using the ACSPL+ programming language.

This guide applies to the SPiiPlus NT motion control product lines as detailed in **Section 1.1.2 - ACS Motion Control NT Product Lines**.

Note	
	*The term "controller" is used in this guide whenever information applies for both controllers and control modules. If information applies to only one of these product groups, the group is stated explicitly.*

1.1 About ACS Motion Control Motion Controllers

1.1.1 ACS Motion Control Company Profile

ACS Motion Control is a global manufacturer of high performance multi-axis motion and machine control systems that combine power and precision to deliver the most flexible, cost-effective and user-friendly control solutions. Established in 1985, ACS Motion Control has its international headquarters in Israel, with North American headquarters in Plymouth, Minnesota and an Asian support center in South Korea. Backed by an ISO9001-certified design and manufacturing capability with an ongoing commitment to quality control and reliability testing, ACS Motion Control delivers its products through an international distribution network that provides sales support and customer service worldwide.

1.1.2 ACS Motion Control NT Product Lines

ACS Motion Control produces the following lines of NT products:

- **MC4U advanced multi-axis control system**

  Intended for multi-axis motion control applications that require high performance, flexible drive configuration, PLC motion and logic control, the MC4U provides the flexibility of up to eight integral servo drives and 64 distributed axes via CANOpen.

- **SPiiPlus NT/DC Motion Controllers**

  The ACS Motion Control SPiiPlus NT/DC motion controllers product line is an extension of the company's SpiiPlus-3U-HP motion controllers that incorporates axes, ACS network element and 3rd party element expansion network-support. As such, it not only provides the capabilities of the SpiiPlus 3U-HP, but expands axis control to 24 and up to 64 (versus 8 axes controlled by the SpiiPlus 3U-HP).

- **SPiiPlus PDMnt Network Module**

  The SPiiPlus PDMnt is a network module designed for controlling external drives and I/Os.

- **SPiiPlus SDMnt Step Motor Drive Module**

  The SPiiPlus SDMnt Step Motor Drive module is a panel mounted, four or eight axis EtherCAT slave module designed for running step motors. It can run seven two-phase unipolar step motors and one two-phase bipolar motor.

  The SPiiPlus SDMnt is designed for slaving to either a SPiiPlusNTM stand alone network controller or an MC4U Control Module configured for NT.

- **SPiiPlus UDMnt (Universal Drive Module)**

  The SPiiPlus UDMnt (Universal Drive Module) is a dual-axis card designed for incorporation in the MC4U Control Module to enable control peripherals over the Ethernet.

  The SPiiPlus UDMnt supports axes in addition to the main axes of multi-axis machinery.

1.2 ACSPL+ Programming Language

ACSPL+ is a powerful programming language developed specifically for SPiiPlus motion controllers. ACSPL+ incorporates many advanced features, including: powerful programming elements such as arithmetical and logical expressions, user-defined variables with local and global scope, user-defined one- and two-dimensional arrays.

SPiiPlus ACSPL+ enables:

- Execution of up to 16 programs simultaneously
- Program isolation - each program resides in a separate buffer
- Rich set of motion types, providing a large degree of versatility
- Advanced implementation of master-slave motion
- Axis-independent programming
- On-condition autoroutines

Complete details of all ACSPL+ commands and variables are given in the *SPiiPlus Command & Variable Reference Guide*.

ACSPL+ libraries are provided for host programming in other high level languages. The library for C, C++, and Visual Basic are described in the *SPiiPlus C Library Reference*. Routines for synchronizing communication between the host program and the SPiiPlus motion controller are given in *SPiiPlus COM Library*.

1.3 About this Guide

This guide is designed to give you practical instruction in using ACSPL+ to program your motion controller. The guide assumes that you have prior experience with programming languages; therefore the guide emphasizes only specific ACSPL+ motion and input/output commands along with practical examples. For the complete ACSPL+ command set see *SPiiPlus Command & Variable Reference Guide*.

The guide is organized in eleven chapters as follows:

Chapter 1	This chapter, providing general information on the SPiiPlus motion controllers and the ACSPL+ programming language.
Chapter 2	**SPiiPlus Architecture** – Provides an overview of the SPiiPlus software architecture.
Chapter 3	**ACSPL+ Overview** – Provides an overview of ACSPL+ programmng, including how to enter code, compile and run.
Chapter 4	**ACSPL+ Motion Programming** – Provides a practical guide for using the ACSPL+ to program motion control.
Chapter 5	**Inputs and Outputs** – Provides details on analog and digital input and output.
Chapter 6	**Fault Handling** – Provides details on safety control and handling faults. Before you create your application make sure that you thoroughly read and understand this chapter.
Chapter 7	**Connection to the Plant** – Provides details on the connection between the motion controller and that which is controlled.
Chapter 8	**Advanced Features** – Provides details on specialized functions as well as certain SPiiPlus model-dependent features.
Chapter 9	**Generic EtherCAT Master** - Provides details of the generic interface of EtherCAT master functionality for the SpiiPlus NT family.
Chapter 10	**Errors & Diagnostics** – Provides procedures for performing error diagnostics.
Chapter 11	**Application Examples** – Provides practical examples for SPiiPlus applications.

1.4 Conventions Used in this Guide

1.4.1 Text Formats

Several text formats and fonts, illustrated in **Table 1**, are used in the text to convey information about the text.

Table 1 Text Format Conventions

Format	Description
Bold	ACSPL+ command names. Software tool menus, menu items, dialog box names, and dialog box elements.
Italic	Emphasis or an introduction to a key concept. In a command syntax, specifies a variable name or other information that the user provides.
`Monospace`	Code example.
`Monospace Bold`	Indicates system responses displayed on the monitor.
blue italic	Names of other documents.
blue bold	Cross references, web pages, and e-mail addresses.

1.4.2 Command Formats

Table 2 provides the symbols employed in sample ACSPL+ commands. The general format is:

COMMAND {+f | -f} arguments . . . [options]

Table 2 ACSPL+ Command Syntax Symbols

Element	Description
command	Language element that must be entered as shown (keyword, command, function, and the like).
{ }	Indicates a set of choices from which to choose one.
|	Separates two mutually exclusive choices. Only one of them is to be selected.
…	An argument that can appear more than once.
[ ]	Optional item(s)

1.4.3 Flagged Text

The following symbols are used in flagging text:

Note 	*Notes include additional information or programming tips.*

Caution 	***A Caution describes a condition that may result in damage to equipment.***

Warning 	***A Warning describes a condition that may result in serious bodily injury or death.***

Advanced 	*Indicates a topic for advanced users.*

Model 	*Highlights a specification, procedure, condition, or statement that depends on the product model.*

1.5 Related Documents

The following documents provide additional details relevant to this guide:

Table 3 Related Documentation

Document	Description
SPiiPlus Command & Variable Reference Guide	Complete description of all variables and commands in the ACSPL+ programming language.
SPiiPlus C Library Reference	C++ and Visual Basic® libraries for host PC applications. This guide is applicable for all the SPiiPlus motion control products.
SPiiPlus COM Library Reference	COM Methods, Properties, and Events for Communication with the Controller.
SPiiPlus MMI Application Studio User Guide	A complete guide for using the SPiiPlus MMI Application Studio and associated monitoring tools.
SPiiPlus Utilities User Guide	A guide for using the SPiiPlus User Mode Driver (UMD) for settingup communication with the SPiiPlus motion controller.
SPiiPlus NT/DC Hardware Guide	Technical description of the SPiiPlus NT/DC product line.
SPiiPlus PDMnt Hardware Guide	Technical description of the SPiiPlus PDMnt Network Interface.
SPiiPlus SDMnt Hardware Guide	Technical description of the SPiiPlus SDMnt Step Motor Drive Module.
SPiiPlus UDMnt Hardware Guide	Technical description of the SPiiPlus UDMnt Universal Drive Module.
MC4U-CS Control Module Hardware Guide	Technical description of the MC4U Control Module integrated motion control product line.
HSSI Expansion Modules Guide	High-Speed Synchronous Serial Interface (HSSI) for expanded I/O, distributed axes, and nonstandard devices.
SPiiPlus NT PEG and MARK Operations Application Notes	Provides details on using the PEG commands in NT systems.

1.6 Terms and Definitions

The following terms and acronyms are used in this guide.

ACSPL+	A programming language for multi-program motion control. Provides access to SPiiPlus resources and strict timing of program execution.
ACSPL+ command	The smallest executable unit of ACSPL+. Several commands can be combined in one ACSPL+ line to be executed in one controller cycle.
ACSPL+ line	A line of code, containing one or more ACSPL+ commands. Can be stored in a buffer as a part of an ACSPL+ program, or can be sent to the controller for immediate execution. By default, one ACSPL+ line is executed in one controller cycle.
ACSPL+ program	A sequence of ACSPL+ lines, loaded in one buffer.
ACSPL+ variable	Can be user variable (user-defined) or standard variable (predefined in firmware).

analog inputs/outputs	The controller's analog inputs accept a signal (voltage range is model-dependent) from an external source such as a sensor or a potentiometer. The analog outputs supply a signal (voltage range is model-dependent) to an external acceptor such as a scope or power amplifier. An ACSPL+ command can access the analog inputs/outputs through the standard variables AIN/AOUT.
application	In a broad sense, an application is any user installation that includes a SPiiPlus controller. In this document the application is considered in a narrow sense, as a set of files that tailors the controller to the specific controlled plant. The set includes the following: ❑ Configuration variables ❑ ACSPL+ program ❑ SP program An application can be stored in the controller's nonvolatile memory, so that the controller will be ready to control the plant immediately after power-up. An application can also be stored as a file in the host computer and downloaded to the controller as required. An application can also include an external host-based program, which is stored and executed on the a PC host computer..
application protection	Application protection does the following: ❑ Protects the user application from unintentional modification. ❑ Prevents harmful operator intervention while the application is running. ❑ Restricts erroneous changes to critical data and execution of potentially dangerous operations while the application is running. At any time the user can enable or disable application protection. When application protection is disabled, none of the protections specified above apply. When application protection is enabled, the controller is said to be in protected mode. When application protection is disabled, the controller is said to be in configuration mode.
array	A standard variable or user variable that contains more than one value (element), as opposed to a SCALAR. All values in an array belong to the same type, either integer or real. An array can be one-dimensional (vector) or two-dimensional (matrix).
autoroutine	A subroutine in an ACSPL+ program activated automatically once a specific condition is satisfied. The user specifies the activation condition and the action executed in autoroutine. A number of autoroutines can be defined in a user application.
axis	An abstraction of the controlled motor. Within the controller an axis is represented by a set of standard variables specifying axis position, velocity, etc. One or more axes serve as an environment for motion. The connection between the controller axes and the physical motor may range from a simple one-to-one correspondence (default), to sophisticated formulae that calculate the motor position using several controller axes.
axis group	A set of axes that act as a coordinated unit. An axis group provides an environment for multi-axis motion. The controller creates and destroys axis groups automatically as required by the executed motion. However, the user can create a permanent axis group that cannot be destroyed automatically.

buffer	A container for an ACSPL+ program. There are 16 program buffers for storing motion control programs plus one buffer (the D-Buffer) for storing definitions of axis names and global variables. The programs stored in thedifferent buffers can be executed concurrently.
closed loop	An element of servo control that causes the specific parameter (feedback) to follow the desired value (reference). For servo motors the controller provides position, velocity and, sometimes, current closed loop. Within the controller all closed loops are digital. The servo processor assigned to the axis provides all necessary calculations.
configuration mode	Mode of application protection where application and critical data can be modified by user. Opposite of protected mode.
configuration variable	A subset of standard variables that tailor the controller to a specific controlled plant. The user typically defines these variables when building an application, and thereafter does not change them. The values of the configuration variables are a part of the application. As a rest of the application the values can be stored in the nonvolatile memory. The controller automatically retrieves the values from the nonvolatile memory on power-up.
controlled plant	An object of the control that includes the following components: ❑ Mechanical part ❑ Motors ❑ Feedback sensors ❑ Additional sensors and actuators connected to the controller's digital or analog inputs/outputs
controller	One of the SPiiPlus NT line of compatible controllers. The models in the line differ by packaging characteristics, available communication channels, number of controlled axes, and internal resources. All models share the same basic architecture and programming language. Information in this guide applies to all SPiiPlus NT controllers.
controller cycle	A primary period within the controller time framework. The controller aligns all its operations to this period. The controller cycle is fixed at 1 millisecond.
data collection	The process of sampling the values of specified variables and storing them in a specified array.
DHCP	**Dynamic Host Configuration Protocol** - a protocol used by networked devices (clients) to obtain various parameters necessary for the clients to operate in an Internet Protocol (IP) network.
digital inputs/ outputs	Digital input accepts binary signal from an external source such as a switch or a relay. Digital output provides binary signal to an external acceptor such as an LED or actuator. An ACSPL+ command can access the digital inputs/outputs through the built-in **IN**/**OUT** variables.
DRA	**Disturbance Rejection Algorithm** - an ACS proprietary algorithm used to improve the disturbance rejection response of the servo, and helps to minimize the position error during the settling phase as well as shorten the settling time.
factory default	Set of controller configuration variables, SP programs and SP data written to the controller nonvolatile memory as a part of the firmware. The user application can replace one or more parts of the factory default. However, the controller can be returned to factory default at any time by using the **#RESET** command.

fault	An abnormal situation detected by the controller. The controller detects the faults by examining the fault conditions as a part of safety control process. The fault conditions include verification of the safety inputs as well as internal integrity.
firmware	A set of files factory-written in the nonvolatile memory. The firmware includes the MPU program, the default SP programs, and the default values of the configuration variables. The user cannot modify the firmware; however, using SPiiPlus MMI Appication Studio the user can upgrade the firmware version. If an application includes the values of the configuration variables, the controller on the power-up retrieves the application instead of the factory defaults. The user can delete the application at any time and to return to the factory defaults by using the **#RESET** command.
global variables	Variables that are common for all buffers. A variable can have either global or local scope. All ACSPL+ variables have global scope. A user variable's scope is specified in the variable declaration.
hardware	The physical components of the SPiiPlus Motion Controller. The principal hardware components include the MPU the Servo Processor, and the nonvolatile memory.
host	The user-supplied computer that communicates with the controller.
host-based program	A program written in C, C++ or any other programming language that runs on the host computer and communicates with the controller.
HSSI	**High-Speed Synchronous Serial Interface** – One of the standard features in the SPiiPlus controllers. HSSI provides an extension of the controller digital inputs/outputs (for HSSI details see *HSSI Modules Hardware Guide*).
immediate execution	A mode of command execution, when a command received via any communication channel is not stored in a buffer but is executed by the controller immediately. An Immediate command is always executed immediately. An ACSPL+ command can either be executed immediately or stored in a buffer.
index	1. An input signal from the encoder or similar sensor that defines an absolute origin of the motor. The signal cause latching of the current encoder position. 2. A syntax element of the ACSPL+ language that provides access to a specific element of array.
leading axis	The first specified axis in an axis group. The ACSPL+ variables, including the variables' parameters, tuning adjustments, and drive amplification, related to the leading axis define motionfor the entire group. The same variables of other axes in the group have no effect on the group motion.
local variables	Variables that are accessible only within the buffer that they are declared. A variable can have either global or local scope. All ACSPL+ built-in variables have global scope. A scope of user variable is assigned in the variable declaration.
mark	A dedicated controller input signal that latches the current encoder position (saves the position to a register). The register value is assigned to the ACSPL+ **MARK** variable. A similar mechanism supports secondary encoders, with the input signal called **MARK2** and the value assigned to the **M2ARK** variable.
master-slave motion	Motion that evolves according to some external or internal signal, as opposed to the time-based motion that evolves as a function of the time. In a simple case, an axis involved in a master-slave motion can reproduce the motion of another axis.
matrix	A two-dimensional array.

motion	Process in the controller that results in a change in internal variables of the controller, and can result in physical effects such as motor movement.
motion profile	Diagram of position or velocity against the time in a time-based motion. The controller provides a third-order motion profile using the velocity, acceleration, deceleration and jerk supplied by the ACSPL+ variables.
motor	The part of the controlled plant that performs physical movement on one axis.
MPU	**Main Processing Unit** – The hardware component that executes the MPU program.
MPU program	The principal part of the firmware. It implements most of the controller functions, and is stored in the nonvolatile (flash) memory. The controller automatically retrieves the MPU program on the power-up. It cannot be modified by the user.
nonvolatile memory	A type of memory that retains the values when the power is off. Referred to as the *flash* memory.
PEG	**Position Event Generator** – A SPiiPlus hardware-supported feature that enables creating events based on exact position.
protected mode	Mode of application protection where application and critical data are protected from user intervention. Opposite of configuration mode.
PTP	**Point-to-Point**
read-only variable	An ACSPL+ variable that can be read from the controller but cannot be assigned.
safety control	The function of the controller that ensures safe operation of the system. Consists of the verification of numerous fault conditions that the controller executes each controller cycle irrespective of any other activity. Once a fault condition is satisfied, the controller raises the corresponding fault.
safety inputs	Digital inputs that the controller examines in the process of safety control.
scalar	An ACSPL+ variable or user variable that contains one value, as opposed to an array.
servo tick	The period of SP program execution. This defines the sampling rate implemented in the servo processor for digital servo control and fine interpolation .
settling time	The time required for a motor to reside within a target envelope around the target point before the controller raises the in-position bit in the ACSPL+ **MST** variable.
simulator	An integrated part of each SPiiPlus tool that emulates the controller operations without producing actual movement. The Simulator is useful while developing the user application, for learning ACSPL+ and for demonstration purposes.
SPii	Servo Processor 2nd Generation
SPiiPlus NT	Family of NT controller products built around the SPii.
Terminal command	A Terminal command is executed immediately and is not stored in a buffer. The command is entered via the SPiiPlus MMI Application Studio Communication Terminal and can be sent to the controller through any communication channel. For Terminal commad details see the *SPiiPlus Command & Variable Reference Guide*, and *SPiiPlus MMI Application Studio User Guide*.
time-based motion	Motion that evolves as a function of time, as opposed to master-slave motion. The characteristic feature of a time-based motion is a motion profile that shows the motion progress against the time.

tools	A set of Windows applications provided with the controller to aid the user in developing and implementing an application. Each tool communicates with the controller in order to perform various functions. Each tool includes a controller simulator, which enables the tool to be used without requiring a physical connection to the controller.
user variable	User defined variable as opposed to ACSPL+ variables.
vector	A one-dimensional array.

2 SPiiPlus Architecture

This chapter provides an overview the SPiiPlus architecture.

2.1 Hardware Structure

The following diagram shows the principal parts of the SPiiPlus controller hardware:

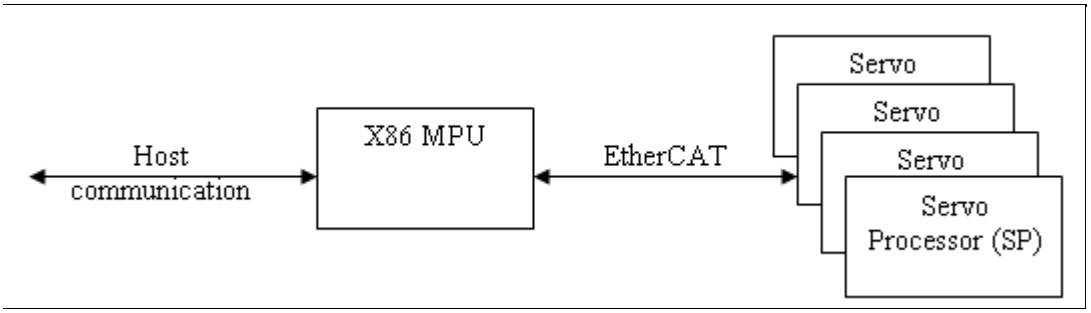


Figure 1 SPiiPlus Controller Hardware Structure

The Motion Processing Unit (MPU), which executes most of the controller tasks, is a powerful x86 processor. The MPU is an EtherCAT bus master and the Servo Processors (SP) are EtherCAT slaves. One master can control several slaves.

The master MPU runs compiled ACSPL+ programs and generates motion profiles to the SPs. Its principle tasks are:

- Communication with the SPs
- Motion profile generation (calculation of **APOS**)
- Calculation of Reference Position (**RPOS**)
- Safety control
- Data collection
- Position Event Generation (**PEG**)
- Processing of Index and Mark inputs
- Execution of ACSPL+ programs
- Communicating to Serial Link or Ethernet
- Execution of Immediate commands received from the Host
- Housekeeping

The MPU is equipped with a Flash (nonvolatile) memory that retains the stored data after the power to the controller is turned off.

The SP executes the realtime tasks, such as implementation of the realtime control algorithms. Each SP controls four axes. The SP includes all the necessary peripherals that are needed for a high performance axis control.

A single MPU module can manage several units over the EtherCAT bus thus expanding the number of controlled axes as shown in **Figure 2**:

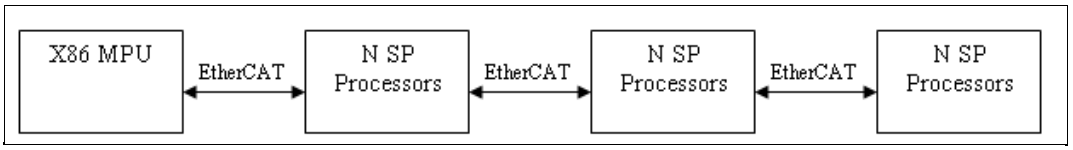


Figure 2 Multiple SPs Connected via EtherCAT

Note	*The part of the system that is connected to the rest of the system via the EtherCAT bus is called a* **unit**.

2.1.1 Firmware

The firmware consists of a set of files factory-written in the flash memory.

You cannot erase or modify the firmware; however, you are able to update the firmware version with a special tool that is part of the SPiiPlus MMI that is supplied with the controller.

The firmware files include the following:

- MPU program.
- Default values of the controller's configuration variables.

2.1.2 Controller Cycle and Servo Tick

The firmware operates in a rigid realtime framework. The two principal parts of the firmware, the MPU program and the SP programs are realtime programs operating in strict synchronism.

The SP interrupt has a fixed interval period (called a "servo tick") of 50 µSec (20kHz). Most SP tasks are executed each servo tick. Therefore, the SP executes the servo control algorithm at a constant rate of 20kHz irrespective of the number of axes and other factors.

The 20kHz SP clock is divided by a factor of 20, generating a 1kHz clock for MPU interrupts. The new clock, the controller cycle, is 1.0 µSec.

2.1.3 realtime and Background Tasks

MPU program tasks are divided into two categories:

- Realtime tasks

  The realtime tasks are executed in strict synchronism with the MPU interrupt. Each controller cycle, a required part of each realtime task is executed. The overall time of all realtime tasks executed in one controller cycle must be less than the one controller cycle.

  The following MPU tasks are the realtime tasks that are executed each controller cycle:

  - Communication with the SPs
  - Motion profile generation (calculation of **APOS**)
  - Calculation of Reference Position (**RPOS**)
  - Safety control
  - Data collection
  - Position Event Generation (**PEG**)
  - Processing of Index and Mark inputs
  - Execution of ACSPL+ programs

- Background tasks

  Background tasks are not synchronous with the MPU interrupt. Execution of a background task may overlap two or more controller cycles.

  The following MPU tasks are the background tasks and are asynchronous to the controller cycle:

  - Communicating to Serial Link or Ethernet
  - Execution of Immediate commands received from the Host
  - Housekeeping

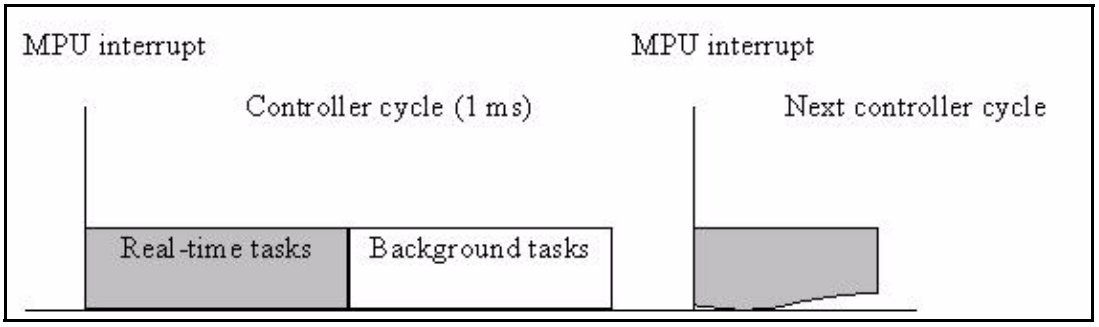


Figure 3 The Internal Structure of the Controller Cycle

The MPU interrupt invokes the realtime tasks. When all realtime tasks required in the current cycle are completed, the controller starts executing the background tasks. If the background tasks complete before the next MPU interrupt occurs, the controller remains idle for the rest of the cycle.

The exact time of the realtime and background tasks in each controller cycle depends on many factors and cannot be precisely specified. The following paragraphs explain different situations in controller cycle utilization.

If the background task execution does not finish in one controller cycle, the background execution is interrupted and continues after execution of the realtime tasks during the next cycle. Therefore, background tasks may overlap into the next MPU interrupt without causing a problem. However, overflow of realtime tasks into the next MPU interrupt is abnormal, and may cause problems with program execution. When this occurs, the controller latches the **Time Overuse** fault. This fault has no default response in the controller, but your application can monitor the fault and define a proper response.

Note 	*You can monitor if the usage of the controller cycle is close to critical. The MMI Application Studio Communication Terminal command:* **#U** *when entered, reports the usage as a ratio of realtime tasks execution time and the controller cycle. A maximum value of 90% is considered dangerous. If the usage limit is reached, you have to modify your application.*

2.2 User Application

This section provides a look at the elements that go into the construction of a user SPiiPlus application.

2.2.1 Firmware, User Application and Tools

The firmware is a program that is stored in the controller's nonvolatile (flash) memory, that defines the basic functionality of the controller. Among these functions are the preparing, storing and executing your applications.

Your application tailors the controller to your specific controlled plant. The controller can control various plants with different number of axes, mechanical construction, timing requirements, etc. Your application specifies the exact control and monitoring actions that must be executed in different conditions, including the exact sequences of motions, activation of outputs, response to inputs and interactions with the operator.

SPiiPlus Tools are Windows-based programs that provide you with support in different stages of the application such as initial set up and tuning, ACSPL+ application development, host application interaction with the controller, and manual control.

2.2.2 User Application Components

The following diagram shows the parts of a user application in the gray blocks and the relevant parts of firmware in the white blocks:

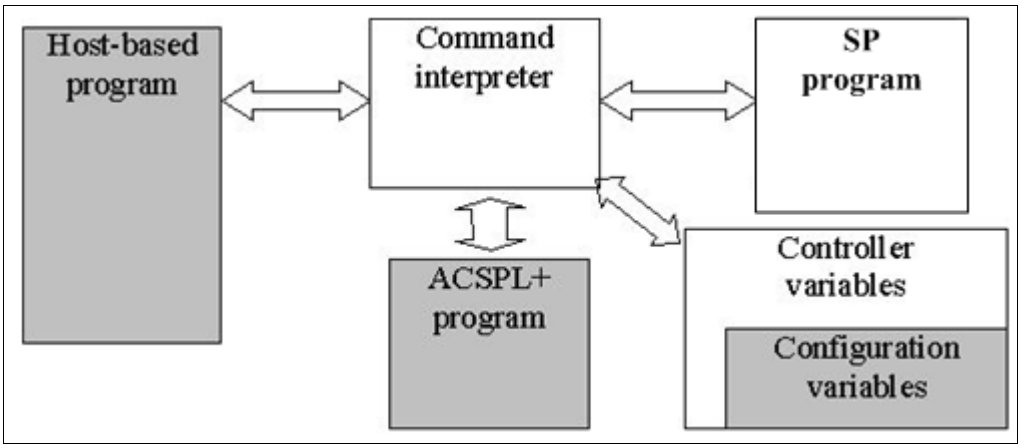


Figure 4 User Application Block Diagram

- Host-based program:

  A program written in C, C++ or any other programming language that runs on the host computer and communicates with the controller. The host-base program uses any communication channel provided by the controller; serial link, Ethernet, FIFO, dual port ram. The program issues commands to, and reads data from, the controller. The program can provide front-end user interfaces, motion sequencing, high-level decision-making and

other application specific functions. This part of user application can be absent if the controller works stand-alone without connection to the host.

The design of host-based programs is not the primary subject of this guide. For Windows programming, refer to the *SPiiPlus C Library Reference Guide*.

- ACSPL+ program:

  A sequence of ACSPL+ commands can be downloaded to the controller as an ACSPL+ program. There are 10 buffers for ACSPL+ programs. An ACSPL+ program is executed inside the controller with strict timing and with no communication delay.

  ACSPL+ programs are almost always present in user applications. Occasionally, the ACSPL+ programs are absent and the host commands all controller actions.

- Configuration variables:

  The firmware includes a set of predefined variables that can be used by ACSPL+ programs and by Immediate commands. The configuration variables are included in this set. The values of the configuration variables are defined by the user to tailor the controller operation to a specific task and plant control. For example, **ACC** defines the acceleration that is used for motion generation. The **SAFINI** variable defines the polarity of the input safety signals.

  The configuration variables must always be present in user applications.

- SP programs:

  The firmware includes SP (Servo Processor) real time control programs as a standard part of the controller.

2.2.3 User Applications Categories

You can use different strategies to build an application, including:

- Stand-alone application:

  No physical link to a host. No host-based program. ACSPL+ programs are stored in the controller flash memory. The ACSPL+ programs start running after power-up and implement all application functions.

- Host-driven application:

  No ACSPL+ programs. The host issues all commands to be executed by the controller. This approach is applicable only for non-time-critical applications.

- Hybrid application:

  A host-based program plus one or more ACSPL+ programs. Most user applications fall into this category.

2.2.4 SPiiPlus MMI Application Studio

SPiiPlus MMI Application Studio is a multipurpose user interface with the controller that provides the user with the means to fully control and monitor the performance of the motion controller.

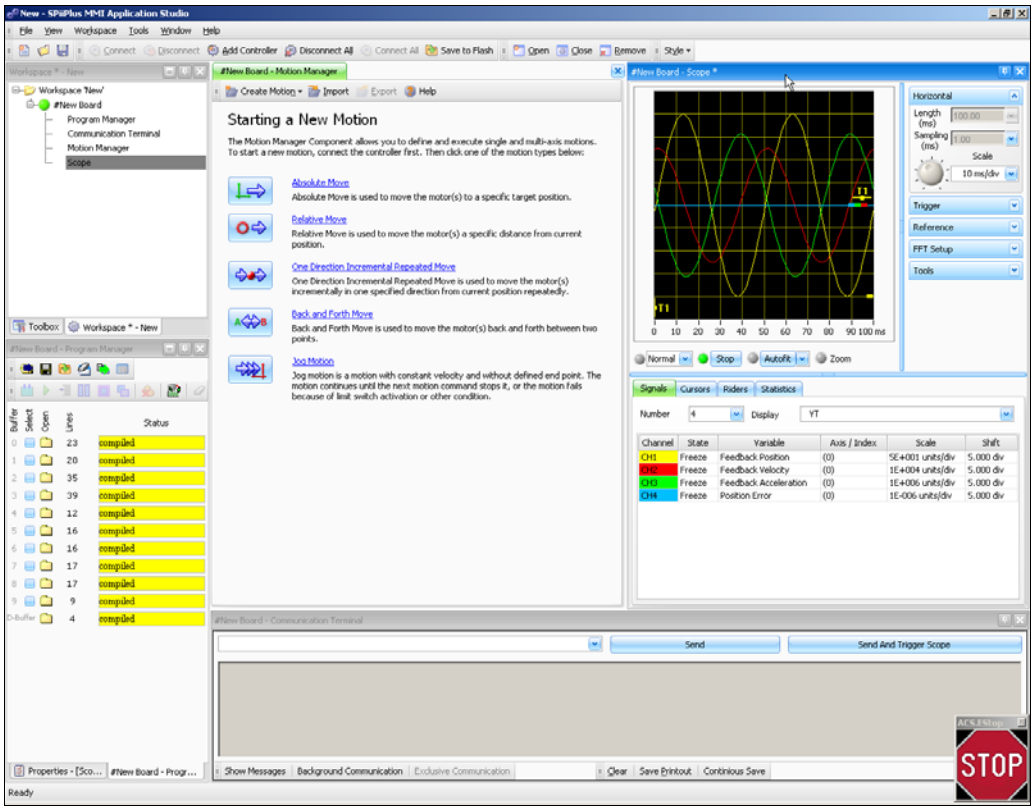


Figure 5 SPiiPlus MMI Application Studio Main Screen

The main features of the SPiiPlus MMI Application Studio are, amongst others:

- Program Manager
  Used for entering user-written programs into the motion controller's buffers.
- Motion Manager
  Used for completely defining the motion for all axes in the system.
- Communication Terminal
  Used for entering commands directly into the controller.
- Scope
  A digital oscilloscope providing a realtime graphic display of the motion.
- Variables Manager
  Enables the user to set watch windows for the values of critical variables.

For complete details see *SPiiPlus MMI Appication Studio Guide*.

2.2.5 File Extensions Used in SPiiPlus MMI Application Studio

Several file formats are used to store data and programs used with SPiiPlus MMI Application Studio as the following table shows:

Table 4 SPiiPlus MMI Application Studio Extensions

File Extension	Content	Associated SPiiPlus MMI Application Studio Component
.acsw	Workspace configuration	Workspace
.awd	Drive database	Adjuster Wizard
.awf	Feedback database	Adjuster Wizard
.awm	Motor database	Adjuster Wizard
.frf	FRF data	FRF Analyzer
.log	MMI log files	All components that generate logs
.par	Controller parameters	Configuration Wizard
.prg	ACSPL+ Program	Program Manager
.rtf	Print to file	All components that enable printouts
.sgn	Scope data	Scope
.spi	Application (includes controller parameters + adjustment parameters + ACSPL+ program + SP files)	Upgrade and Recovery Wizard

2.3 Programming Resources

The controller-based parts of the user application operate in the environment created by the firmware. The environment includes a set of resources that the user application can use. This section provides a short description of the available resources.

2.3.1 Commands

The controller supports a rich set of commands which are divided into two types of command sets:

- **Terminal Commands**

  Terminal commands are those that are sent directly to the controller. They are entered through the **Communication Terminal**. The general structure of the **Communication Terminal** windows is shown in **Figure 6.**

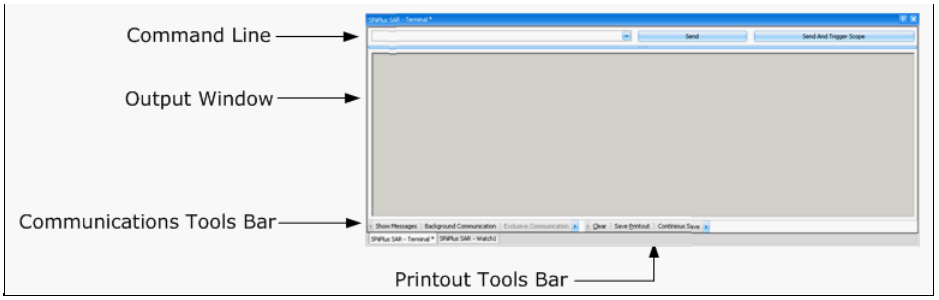


Figure 6 Communication Terminal Window

The Communication Terminal window is described in the *SPiiPlus MMI Appication Studio Guide*.

As soon as the command is received through one of the communication channels it is executed. Each Terminal command starts with **?** (query command) or **#** (program management command), for example:

?FPOS	Display the current position of all motors.
#0L	List the complete program in buffer 0.

A Terminal command cannot be stored in a buffer. Once a Terminal command is received via any communication channel, the controller executes it immediately or rejects the command if it cannot be executed.

- **Buffered Commands**

  The controller stores a sequence of commands in a buffer and executes them as a program. ACSPL+ commands can either be executed immediately or can be stored in a buffer. Examples of ACSPL+ commands that are stored in a buffer:

`enable X`	Enable motor X
`Var = 5*Var2`	Assign the result of expression to variable Var
`wait 50`	Delay the program for 50 μSec
`ptp X,3000`	Move the X motor to point 3000

2.3.2 Program Buffers

The controller provides up to 16 buffers for storing ACSPL+ programs. The controller defines the size of each buffer according to the required size of the program. For all practical purposes, you can consider the size of each buffer to be unlimited.

A program stored in a buffer can be edited, compiled and executed independently of the programs in other buffers. For example, a program in buffer 0 may be running while you edit the program in buffer 1.

Programs stored in different buffers can be executed concurrently. Each buffer defines an execution thread connected to this buffer. When you activate a program in a buffer, the program is executed in a separate thread. Therefore, up to 16 ACSPL+ programs can be executed concurrently.

A buffer also provides isolation between the programs. All labels and local variables defined in a program are isolated in their buffer and are inaccessible from any other buffer. For example, two programs can contain identical user-defined labels, but the controller considers each label as belonging only to the buffer in which it is contained, and relates to all references to the label appropriately.

2.3.3 Declaration Buffer (D-Buffer)

The D-Buffer is an additional special buffer that provides a place for the definitions of axis names and global variables.

Note	
	*Executing programs and autoroutines is not supported in D-Buffer.*

2.3.3.1 Defining Global Objects in D-Buffer

Axes and global variables defined in D-Buffer are not required to be defined in other buffers before use. However, such redefinition is not an error, given all attributes of the definitions are identical.

The **#SAVE** and **#SAVEPROG** commands store the D-Buffer in the flash along with other buffers.

Note	
	*The values are also stored using the SPiiPlus MMI Application Studio Program Manager Buffer Editor.*

At start-up, the controller loads and compiles the D-Buffer before any other buffers.

Note	
	*After any change in the D-Buffer, all other buffers should be recompiled.*

2.3.3.2 D-Buffer Default Contents

The default contents of D-Buffer differ from other buffers. The other buffers are initially empty by default; however, the D-Buffer contains a set of definitions that provides compatibility with previous FW versions. The default contents of the D-Buffer are:

```
!axisdef X=0,Y=1,Z=2,T=3,A=4,B=5,C=6,D=7
!axisdef x=0,y=1,z=2,t=3,a=4,b=5,c=6,d=7
global int
I(100),I0,I1,I2,I3,I4,I5,I6,I7,I8,I9,I90,I91,I92,I93,I94,I95,I96,I97,I98,I99
global real
V(100),V0,V1,V2,V3,V4,V5,V6,V7,V8,V9,V90,V91,V92,V93,V94,V95,V96,V97,V98,V99
```

This provides you with the means to define names for the axes in your system (axisdef). The two 100 element arrays, one an integer and one a real, are, however, for internal use.

The **#RESET** command restores the default content in the D-Buffer.

2.3.4 Command Execution

2.3.4.1 Terminal Commands

Once a Terminal command is received from the Communication Terminal, the controller executes it immediately or rejects the command if it cannot be executed.

The controller executes Teminal commands as a background task. One Terminal command cannot be interrupted by another. The controller finishes execution of a command, sends a reply, and only then continues to the next command. Therefore, a host-based process that sends commands is guaranteed to receive the replies in the same order as the commands were sent.

Typically, execution of a Terminal command takes 1-2 controller cycles. The commands that supply or request a great amount of data, such as query of a large array, may require a longer processing time.

Processing time can also be affected by a high MPU usage. The realtime tasks always have the greatest priority. If the usage (percentage of the realtime tasks in the controller cycle) reaches

90% or more, the response time of the controller deteriorates. If an application requires the fastest response to Terminal commands, you must keep the usage below 50%.

2.3.4.2 ACSPL+ Commands

There are two methods for executing ACSPL+ commands:

- Execute immediately - via the Communication Terminal.
- Store a sequence of commands in a buffer and then execute the sequence as an ACSPL+ program.

If the prompt is **:** (colon), no program buffer is open for editing, and an ACSPL+ command, transmitted to the controller through the Communication Terminal, is executed immediately.

Immediate execution of ACSPL+ commands is a background task. Therefore, the processing time can be affected by a high MPU usage. The realtime tasks always have the highest priority. If the usage (percentage of the realtime tasks in the controller cycle) reaches 90% or more, the response time of the controller deteriorates. If an application requires the fastest response to Immediate commands, you must keep the usage below 50%.

Executing an ACSPL+ program from a buffer is different because the controller executes a buffered ACSPL+ program as a realtime task. Up to 16 buffered ACSPL+ programs can be executed simultaneously, and in parallel.

2.3.5 ACSPL+ Standard Variables

ACSPL+ standard variables are a set of predefined variables provided by the controller. You can use ACSPL+ standard variables in any command, either immediate or buffered, without having to declare them.

The ACSPL+ standard variables are divided into two categories:

- State variables - Examples are **FPOS** (Feedback Position), which reports the immediate position of a motor and **MST** (Motor State), which indicates the motor status, including whether it is enabled and whether is involved in motion.
- Configuration variables - The values of the configuration variables tailor the controller to a specific control object. Examples are **ACC** (Acceleration), which specifies a tolerable acceleration of a motor and **SLLIMIT** (Software Left Limit), which specifies a lower limit for the area of motion, etc.

ACSPL+ variables are mentioned throughout this guide where they relate to a particular element or feature of ACSPL+.

2.3.6 User-Defined Variables

In addition to the set of ACSPL+ standard variables, you can declare variables with user-defined names as required in the application.

A user-defined variable can be declared as scalar, one-dimensional array (vector) or two-dimensional array (matrix).

Note	
	*A user array can contain up to 100,000 elements.*

A user-defined variable can be declared as local or global. Local variables are accessible only within the buffer that the declaration resides in. Global variables are common to all buffers and can be accessed from any buffer.

2.3.7 Nonvolatile Memory and Power Up Process

The on-board nonvolatile (flash) memory retains information while power is off.

The flash memory stores the following data:

- Firmware as well as SP programs
- User application

A new controller is supplied with only the firmware and SP programs stored in the nonvolatile (flash) memory. You cannot erase or modify the firmware or the SP programs. You can, however, store ACSPL+ programs and configuration variable values in the controller's flash memory.

During the power-up process the controller loads the firmware, the ACSPL+ programs, the configuration variables and the SP programs from the flash memory. If you did not store any component of configuration variables, the controller loads the default firmware component.

2.4 Executing ACSPL+ Programs

2.4.1 Program Buffers

The controller provides up to 16 program buffers.

Each buffer provides:

- Separate storage for each ACSPL+ program
- An isolated environment for program editing/execution
- Separate thread for concurrent execution
- Independent autoroutine execution

Each buffer is managed independently of the other buffers. For example, you can edit a program in one buffer while the program in another buffer is executing.

All labels and local variables in a program are local in the encapsulating buffer. Programs in other buffers do not have access to these label and variables. Even if two programs in different buffers both define a local variable with the same name, the variables are considered as two different variables, each in its corresponding buffer.

A program in a buffer can be executed independently of any other program. The program executed in a buffer does not affect the program executed in other buffers, unless you have provided for synchronization through the global variables or common resources.

If a program in a buffer includes one or more autoroutines, the buffer manages the autoroutines independently of other buffers. If an autoroutine condition is satisfied, only the program executed in the enclosing buffer is interrupted. No other buffer is affected.

The time allotted for processing non-executable lines, e.g., comment, a new line, a label, etc., is controlled by the **S_FLAGS.1** bit. If the bit is 0 (default), the non-executable line will be skipped during execution. If the bit is 1, the line will be allotted a controller cycle. executed as before taking a standard time for execution.

The bit affects the program compilation; therefore, if the bit is changed, the results will be visible only after a program is recompiled.

2.4.2 Execution of a Single Program

Assume a compiled ACSPL+ program containing no errors is stored in buffer 0, and you issue the following command in the Communication Terminal:

```
#0X
```
Execute the program in buffer 0

The controller starts execution from the first line.

The controller executes the program according to this simple model:

- One program line is executed each controller cycle. If a line contains several ACSPL+ commands, all them are executed in one controller cycle. See also Execution Rate below.
- If a program is linear, meaning that it contains no program flow commands, like goto, and no autoroutines, the program lines are executed sequentially: the first line in the first controller cycle, the second - in the second controller cycle, and so on. A program flow command redirects execution and defines another line to be executed in the next controller cycle.
- A number of commands can delay program execution. For example, the command

```
wait 50
```

will execute for 50 milliseconds instead of one controller cycle. The command:

```
till ^MST(0).#MOVE
```

provides a delay of execution time until the 0 axis motion ends.

Commands that use the controller resources, like motion commands, also can be delayed if the resource is busy. After a command that caused a delay has finished, the controller continues executing one line per one cycle.

2.4.3 Concurrent Execution

Assume the programs in buffers 0, 1, 2 are linear and include no commands that can delay execution. After starting simultaneously, the programs execute in a simple and predictable way: one line of each running program per one controller cycle.

In the first controller cycle the controller executes line number 1 from buffer 0, line number 1 from buffer 1 and line number 1 from buffer 2. In the second controller cycle the controller executes line number 2 from buffer 0, line number 2 from buffer 1 and line number 2 from buffer 2, in progression.

If a program contains a program flow command or a command that delays execution, this strict synchronization vanishes, but the general principle remains the same: one line of each running program per each controller cycle.

The rule does not depend on the number of concurrent programs. If all 16 programs run concurrently, the controller executes 10 ACSPL+ lines in each controller cycle - one from each running program.

Delay in one executed program has no affect on other executed programs. Each buffer provides an isolated thread for program execution.

Within a controller cycle, the order of executing the program lines follows the buffer numbers: first, a line from buffer 0 is executed, then a line from buffer 1, and so on.

2.4.4 Immediate Execution

What happens if while one or more programs are running at the same time and the controller receives a Communication Terminal ACSPL+ command through a communication channel?

The controller executes the Terminal command in an additional thread not connected with any buffer. Therefore, the controller provides 16 + 1 threads for ACSPL+ execution. Sixteen threads are assigned to the program buffers and one thread is reserved for immediate execution.

All rules of the execution model also apply to immediate execution. For example, if several commands are combined in one line for immediate execution, all of them are executed in one controller cycle in parallel with the lines from each running program.

Within a controller cycle, the immediate line is executed after all lines from the running programs. Therefore, in one cycle the controller executes a line from buffer 0, then a line from buffers 1, 2, up to 15, and then executes a CommunicationTerminal command line (if any has been received).

2.4.5 Immediate Execution vs. Stored Program

The controller provides three options for executing ACSPL+ commands:

- Execute a command immediately
- Store a sequence of commands in a buffer and then execute the sequence as an ACSPL+ program
- Execution in a dynamic buffer

If the controller prompt is: *No program buffer is open for editing*, and an ACSPL+ command, transmitted to the controller through any communication channel, is executed immediately.

If the prompt contains a line number like 2:00001>, a program buffer is open for editing, and an ACSPL+ command, transmitted to the controller through any communication channel, is stored in the open buffer. ACSPL+ commands stored in a buffer constitute an ACSPL+ program.

2.4.6 Autoroutine Execution

An autoroutine is a part of ACSPL+ program and can be placed in any program buffer. Each ACSPL+ program can contain from zero to any number of autoroutines. An autoroutine placed in a buffer shares the local variables and the same thread for execution with other autoroutines in the same buffer and with the rest of ACSPL+ program.

If a buffer contains one or more autoroutines, the execution model described above is slightly modified. In each controller cycle, the controller examines the conditions of all autoroutines in the buffer, before executing the next line in a buffer. If a condition is satisfied, the controller does not execute the next line but executes the first line of the body of the corresponding autoroutine. Therefore, the autoroutine interrupts the program executed in the same buffer. An autoroutine in one buffer has no affect on the program or autoroutines in another buffer.

Autoroutines provide an interrupt-like response to external or internal events. For more information about autoroutines see **Section 3.8.3 - Autoroutines**.

2.4.7 Synchronization and Mutual Exclusion

Though the controller provides independent execution of concurrent programs, applications can often require that their component programs be synchronized at certain points. The controller provides a simple and flexible approach to solving synchronization problems. In addition to an ACSPL+ line serving as a unit for concurrent execution, the same ACSPL+ line serves as a unit of mutual exclusion. Namely, execution of an ACSPL+ line cannot be interrupted by a concurrent program or by an autoroutine.

Therefore, an ACSPL+ line provides an atomic unit of execution. Given that a single ACSPL+ line can contain any number of ACSPL+ commands, various synchronization tasks can be resolved using global variables.

2.4.7.1 Mutual Exclusion

Mutual exclusion is the most frequently used synchronization task. As the execution of an ACSPL+ line is atomic, mutual exclusion of the lines in concurrent programs is automatic and does not require any intervention from the user. If a critical section requiring mutual exclusion comprises only a few commands, then you simply place these commands in one line.

However, if a critical section is long, or combining it in one line is undesirable for any reason, another solution must be found. The following construction implements a simple semaphore, which is ON (one) while the program is inside the critical section, and is OFF (zero) otherwise.

```
global int Mutex                Variable Mutex implements semaphore
. . . . .                       Any program actions
till ^Mutex; Mutex = 1          Enter critical section
. . . . .                       Critical section
Mutex = 0                       Exit critical section
```

The Enter and Exit lines enclose the critical section. If the program contains several critical sections, each section must be enclosed with the same Enter and Exit lines as shown in the example above.

All programs that require mutual exclusion must include the same declaration of the Mutex variable and the same embracing of each critical section.

This construction guarantees that only one of the concurrent programs may be inside a critical section. If the second program tries to enter the critical section, the command **till ^Mutex** delays the program execution until the first program zeroes **Mutex** on exit from critical section.

It should be noted that the solution is based on automatic one-line mutual exclusion. Therefore, the two commands:

```
till ^Mutex; Mutex = 1
```

must be in one line. If they are placed in two sequential lines, they cannot provide mutual exclusion.

2.4.7.2 Synchronization

Assume two programs that run mostly asynchronously must execute certain commands at the same time. There is a point in each program that whichever program comes to the point first, it must wait for the second program to come to its synchronization point. Then the next lines in both programs will be executed in the next controller cycle.

The problem is solved by the following construction:

```
global int Sem                          The Sem variable implements a general semaphore
. . . . .                               Asynchronous part of the program
Sem = Sem+1; till Sem = 2; Sem = 0      Synchronization point
. . . . .                               The line will be executed synchronously
```

The same definition of the **Sem** variable and the same line of synchronization point must be in the second program.

Whichever program comes to its synchronization point first, the command **till Sem = 2** provides waiting for the second program. The assignment **Sem = 0** provides reuse of the construction if necessary.

The solution can be also be extended to three or more concurrent programs.

2.4.7.3 Execution Rate

In the execution model described above, the controller executes one line from each running program per each controller cycle. The execution rate in each buffer is the same.

You can modify the execution model by specifying how many lines in each buffer the controller must execute in one controller cycle.

ACSPL+ variables **PRATE** and **ONRATE** (see *SPiiPlus Command & Variable Reference Guide*) contain one element per each program buffer. Each element of **PRATE** specifies how many lines in the corresponding buffer the controller must execute in one controller cycle, except the case if an autoroutine is executed. Each element of **ONRATE** specifies the same when an autoroutine is executed.

For both variables, the default value is 1 (one line per each controller cycle). The user can increase the value up to 10 (ten lines per each controller cycle).

In a typical case the user increases **PRATE** and **ONRATE** in one buffer, providing the higher execution rate in this buffer. The program in this buffer runs faster, than the programs in either buffers, as if the buffer has a higher priority.

You should, however, exercise caution when simultaneously increasing the execution rate in buffers. Increasing the execution rate increases the usage of the controller realtime cycle and may cause a Time Overuse fault. Use the **#U** Immediate command (see *SPiiPlus Command & Variable Reference Guide)* to monitor the present usage. If maximum usage value approaches 90%, the application places an excessive load on the controller. Decrease execution rates, simplify your application, or use a more powerful model of the controller, to solve the problem.

3 ACSPL+ Overview

SPiiPlus enables running up to 16 separate ACSPL+ programs. The programs are stored in what are referred to as "buffers". The programs are entered via the SPiiPlus MMI Application Studio Program Manager (see *SPiiPlus MMI Application Studio User Guide* for details).

This chapter provides a general overview of ACSPL+ programming.

For complete details of the ACSPL+ command set and variables refer to the *SPiiPlus Command & Variable Reference Guide*.

3.1 ACSPL+ Syntax

3.1.1 Commands, Lines, and Command Aggregates

A command is the smallest executable unit of ACSPL+.

One program line may contain one or several commands. If a line contains several commands, the commands must be separated by a semicolon (**;**). A semicolon after the last (or single) command in a line, however, is not required.

Examples:

The following is an example of a program line that contains one assignment command:

```
V0 = V1 + 2*(V2 - V3)
```

The following is an example of a program line that contains two assignment commands:

```
V0 = V1 + 2*(V2 - V3) ; V4 = 0
```

White spaces (spaces and tabs) may be inserted arbitrarily inside or between commands. However, white spaces must not be inserted within a keyword or variable name.

A command aggregate consists of several commands. It starts with a specific command and terminates with the **end** command. For example, a loop structure starts with the loop command followed by an arbitrary number of commands and terminates with the **end** command signalling the completion of the loop. The commands of a structure may reside in one or more program lines.

3.1.2 block..end Control Structure

The commands specified within the **block..end** structure are executed in one controller cycle.

The structure has the following syntax:

block command_list end

The structure provides an alternative to specifying **command_list** commands in one line. The commands within the structure can be specified in several lines. However, the controller executes all commands in one controller cycle, as if they were written in one line.

The following limitations are applied to the commands within the structure:

Commands and functions that may cause delay (**wait**, **till**, **getsp**, **while**, **loop**, etc.) provide delays even if they are used within the **block..end** structure.

3.1.3 ACSPL+ Keywords

ACSPL+ Keywords are reserved words that have specific meanings. They cannot be used as names in a program. **Table 5** lists the ACSPL+ keywords. For their meanings and use, see the *SPiiPlus Command & Variable Reference Guide*.

Table 5 ACSPL+ Keywords

abs	ecunmap	log	save
acos	edge	log10	send
all	else	map	set
arc1	elseif	mapby1	setsp
arc2	enable	mapby2	setdspini
asin	enableon	map2	setdsp
atan	end	map2free	sign
atan2	ends	max	sin
avg	extin	master	slave
binp	extout	min	split
break	exp	monsp	splitall
call	floor	mpoint	sqrt
coe	getsp	mptp	start
coeread	getdsp	mref	stop
coewrite	getdspini	mseg	stopall
ceil	global	on	stopdc
connect	go	path	stopper
cos	goto	pause	till
dc	imm	peg	tan
deadzone	int	point	vsp
disable	intgr	projection	wait
disableall	jog	pow	while
disableon	kill	ptp	write
disp	killall	read	zbal
do	lag	real	
dsign	ldexp	resume	
dynamic	let	ret	
ecin	line	roll	
ecout	local	sat	

3.1.4 Names: Variable and Label

A name is a sequence of alphanumeric characters used to denote one of the following:

- Variable (ACSPL+ or user-defined)
- Label

The first character of a name must be a letter.

The names of ACSPL+ variables are predefined and cannot be changed. For a full list of ACSPL+ variables see the *SPiiPlus Command & Variable Reference Guide*.

There is also an ACSPL+ label: **AUTOEXEC**. If this label is used in a program, then execution will start at this point when the controller is powered up.

You can declare user variables and user labels as required in the application. The number of user-defined names and the length of each name are essentially unlimited. However, you may not declare variable names that coincide with the following:

- Keywords (**if**, **wait**, **sin**, etc.)
- Names of ACSPL+ variables (**FPOS**, **MST**, etc.)
- **AUTOEXEC** (ACSPL+ label)
- Postfix- indexed form of an ACSPL+ variable

3.1.5 Case Sensitivity

All keywords are case-insensitive, e.g., the words **if**, **IF**, **If** or **iF** all have identical meaning in a program.

Variable and label names, however, are case-sensitive, e.g., the names **FPOS** and **Fpos** designate two different variables.

3.1.6 Axis Designations

Many ACSPL+ commands take one or more axes as arguments. This applies particularly for motion commands (see **Chapter 4 - ACSPL+ Motion Programming**).

Axes can be designated as a single letter or a sequence of letters, like **X**, **XYZ**, **ZAD**, or as a list of values enclosed within parentheses, for example, **(0, 2, 4)**. The values can also be represented by variables.

Some commands support the keyword: **all**, to designate all the axes supported by the controller.

The following examples are equivalent:

```
axisdef XY=0                    !Define axis variable
enable XY                       !Enable axis
ptp XY, 100, 200

enable (0,1)                    !Axes designated as list of values.
ptp (0,1), 100, 200
```

```
int first_axis, second_axis                !Variable declarations.
first_axis = 0                             !Variable assigned value.
second_axis = 1                            !Variable assigned value.
enable (first_axis, second_axis)           !Axes designated as list of variables.
ptp (first_axis, second_axis), 100, 200
```

3.1.7 Comments

A comment is text in a program that the controller stores along with the program but ignores while executing the program. Comments are normally used to annotate a program.

A comment starts with an exclamation mark (**!**). An exclamation mark encountered in a line designates all subsequent characters in the line as part of a comment. If the exclamation sign is the first character in a line, the whole line is a comment.

```
! This entire line is a comment.
V0 = V1 !This comment starts from the exclamation mark.
```

3.2 Variables

Variables have the following attributes:

- Name
- Class (standard or user variable)
- Scope (global or local)
- Lifetime
- Accessibility (read-write, read-only, protected)
- Type (integer or real)
- Size
- Value

Note	
	*The controller has a large set of ACSPL+ variables with predefined names. You cannot change these names. These ACSPL+ variables can be used in ACSPL+ commands without an explicit declaration.*

3.2.1 Variable Name

Variable names follow the general syntax rules given in **Section 3.1.4 - Names: Variable and Label**.

User-defined variable names must be specified by explicit declaration. For example:

```
int Var1, Var2              !Declare local variables Var1 and Var2
global real RealVar         !Declare global variable RealVar
```

3.2.2 Variable Class: ACSPL+ or User

ACSPL+ variables are predefined in the controller. ACSPL+ variables can be used in ACSPL+ commands without an explicit declaration.

ACSPL+ variables are mentioned throughout this guide where they relate to a particular element or feature of ACSPL+. For details of the ACSPL+ variables see the *SPiiPlus Command & Variable Reference Guide*.

User-defined variables are defined by explicit declarations. A declaration can appear as:

- A part of an ACSPL+ program. The declared variable becomes available when the program is inserted into one of the program buffers and is compiled. Any attempt to query a variable in a buffer window that has not been compiled will result in an error.
- An immediate ACSPL+ command. Only global variables can be used as a Terminal command. The declared variable becomes available immediately after the controller executes the command.

3.2.3 Variable Scope

Variables have either global or local scope.

3.2.3.1 Global Variable Scope

A variable with global scope can be used in any program buffer and also in Terminal commands.

All ACSPL+ variables have global scope. A user-defined variable can be declared as either global or local.

Declaration of user global variable starts with the keyword **global**. For example:

```
global real GlobVar     !Declare the GlobVar variable as a global variable.
```

This declaration may appear in several program buffers. However, in distinction to local variables, all these declarations are considered to be the same variable.

Using a ACSPL+ variable in a program does not require explicit declaration of the variable. However, a global user variable must be declared before it can be used in a program. Terminal commands can use any global variable without explicit declaration.

3.2.3.2 Local Variable Scope

A variable with local scope can be used only in the program buffer where it is declared. Only user variables can be local.

By default, if a declaration does not contain the keyword **global**, the variable is declared as local. However, it is recommended that you include the keyword **local** for clarity. For example, the declaration

```
real LocVar
```

is the same as

```
local real LocVar
```

Both of them define **LocVar** as a local variable.

A local variable is valid only within the buffer where it is declared. If a local variable with the same name is declared in another buffer, the controller considers them as two different variables.

The name of a local variable may also be the same as the name of a global variable. The controller considers them as different. The program where the local variable is declared has access to the local variable only.

3.2.4 Variable Lifetime

All ACSPL+ variables are valid as long as the controller firmware is active; from power-up to power-down.

A user local variable is valid as long as the program containing it is compiled. Therefore, a local variable becomes active when the program containing it is inserted into one of the buffers and is successfully compiled. After compilation, the variable is assigned its value and is available for the **#V** (list of variables) Communication Terminal command and queries.

Local variables are erased when the program they to which they belong returns to non-compiled state. Programs may enter a non-compiled state as a result of:

- Explicit **#SR** command (Stop and Reset)
- Editing or inserting another program in the buffer

When a program terminates normally, it remains in a compiled state and local variables therefore remain valid. In addition, if a controller-executed program causes an error, the controller terminates the program, but the buffer remains in a compiled state. Any user variable in the buffer therefore remains valid.

A user global variable declared in one or more ACSPL+ programs is valid as long as at least one of the programs that contains it is compiled. After compilation, the global variable is assigned its value and is available for the **#V** (list of variables) Terminal command and queries.

A user global variable disappears when the last program that it is contained in returns to a non-compiled state. The conditions when a program may return to non-compiled state are discussed above.

A special case of user global variable is the persistent global variable. A global variable is defined as persistent when the variable declaration is not a part of any program but is issued as an immediate ACSPL+ command.

Assume, that you execute the following commands via the **Communication Terminal** window:

```
global real PersistentVar    !Declaration
?PersistentVar               !Query
 0                           !Initial value
```

The controller immediately accepts the declaration and creates the persistent global variable: **PersistentVar**. The variable is now valid and can be queried as illustrated by the query command, **?PersistentVar,** that follows.

The lifetime of a persistent global variable is not connected with any program. A persistent variable survives any change in the program buffers and may be erased only by the explicit **#VGV** (Clear Global Variables) Terminal command.

3.2.5 Variable Accessibility

All user variables have read-write access and are not protected. Any ACSPL+ command located in the scope of a variable can use or assign the variable's value at any time.

An ACSPL+ variable can belong to any of three access classes:

- read-write

  Read-write ACSPL+ variables such as user variables can be used or assigned at any time. Examples of read-write ACSPL+ variables are **VEL** (Velocity) and **IST** (Index State).

- read-only

  Many ACSPL+ variables are read-only. These variables cannot be assigned. The read-only variables provide readouts of the controller state. Examples include **RPOS** (Reference Position), **FPOS** (Feedback Position), **MST** (Motor Status).

- protected

  Protected ACSPL+ variables define the configuration of the controller, and are therefore called configuration variables. The values of configuration variables can be read at any time. Assignment to configuration variables is allowed only in the Configuration Mode. Examples of configuration variables are **ENTIME** (Enable Time), and **XACC** (Maximal Acceleration).

In most applications, you set the values of configuration variables during the application development process by using the special tools provided for this purpose in the Protection wizard of the SPiiPlus MMI Application Studio (see the *SPiiPlus MMI Application Studio User Guide*).

3.2.6 Variable Type: Integer and Real

ACSPL+ supports integer and real variables.

Each ACSPL+ variable has a predefined type. For a user variable the type is specified in the declaration.

The controller provides automatic conversion from integer to real and from real to integer if required. Therefore you are not restricted in using variables of both types in ACSPL+ commands.

For example in the following fragment:

```
int Var1          !Declare Var1 integer variable
real Var2         !Declare Var2 real variable
Var1 = Var2       !Controller automatically converts real to integer
Var2 = Var1       !Controller automatically converts integer to real
```

The types differ by internal presentation and behavior in arithmetical operation.

- An integer value occupies 4 bytes, 32 bits, numbered from 0 to 31. The sign is located in bit 31, bit 0 is the least significant bit of the mantissa.
- A real value occupies 8 bytes, 52 bits of mantissa and 12 bits of exponent. The format corresponds to the standard double format of PC.

3.2.7 Variable Size

A variable can be a scalar, one-dimensional or two-dimensional array. A one-dimensional array is referred to as a vector, and a two-dimensional array is referred to as a matrix.

The size of each ACSPL+ variable is predefined in the controller. For example, the **S_FAULT** (System Faults) variable is a scalar variable, and the **FAULT** (Motor Faults) variable is an array of 8 elements.

Many ACSPL+ variables are sized according to the number of axes, with one element per each axis. For example, each element of the **FAULT** array displays faults for one axis.

The size of user variables is defined in the variable declaration. For example:

```
int ScalarVar              !Declare the ScalarVar variable as a scalar variable
global real Ar1(100)       !Declare Ar1 as a global array of 100 real numbers
int Ar2(10)(200)           !Declare Ar2 as a 10x200 matrix of integers
```

Note	*The maximum size of a user array is 100,000 elements.*

3.2.8 Variable Value

Value is the only variable attribute that can change during a variable lifetime. In most applications, a value is changed explicitly, as a result of assignment.

Read-only variables change their values implicitly. For example, the controller updates the elements of variable **FPOS** (Feedback Position) each controller cycle, so that each element displays the actual position of the corresponding motor.

3.3 Variable Declaration

ACSPL+ variables are predefined in the controller and do not require explicit declaration; however, each user variable must be declared in the program where it is used.

Syntax:

[scope-specification][type-specification] name [size-specification]

Where:

- **[scope-specification]** is one of:
  - **local**
  - **global**
- **[type-specification]** is one of:
  - **int**
  - **real**
- **name** - the name of the variable
- **[size-specification]** - required when defining an array, it can be one of:
  - **(integer)** - One dimensional array
  - **(integer) (integer)** - Two dimensional array

Any of the parts of either the scope-specification or the type-specification can be omitted, but not both. If scope-specification is omitted, the **local** specification is implied. If type-specification is omitted, the **int** specification is implied.

If no size-specification is specified, the variable is taken as scalar.

3.3.1 Declaration of Global Variables

A global variable is common for all programs in all program buffers.

To have access to a global variable, a program must declare it. Without explicit declaration a global variable is invalid in a program. Therefore declaration of a global variable may appear in several programs, but all these declarations refer to the same variable.

Declarations of a global variable in different programs must be identical. Declaring a global variable in two buffers with the same name, but different type or size, causes a compile error.

A global variable becomes active once the first program containing it is compiled in one of the program buffers. The variable remains valid as long as at least one of the programs containing it remains compiled.

Local and global variables are initialized to zero during compilation.

Examples of the global variable declarations:

`global int Var1`	Var1 is declared as a global integer variable
`global real Var2, Var3`	Var2 and Var3 are declard as global real variables
`global int Var4, Var5(2)(100)`	Var4 is declared as a global integer variable and Var5 is declared as a two-dimensional array of integers

3.3.2 Persistent Global Variables

A persistent global variable is a global variable that is declared by Terminal command. Terminal execution of ACSPL+ commands is described in **Section 2.4.4 - Immediate Execution**. A command executed immediately is not included in a program or stored in a buffer.

A global command executed immediately declares a persistent global variable, whose lifetime is not dependent on any buffer. A persistent variable becomes active immediately when the command executes. The variable remains valid throughout any buffer manipulations. The only way to erase a persistent variable is by executing the **#VGV** (Vanish Global Variables) command. The command erases all persistent global variables that are not referred in any compiled buffer.

A program in any buffer can access a persistent global variable. Like a regular global variable, a persistent variable must be re-declared in each buffer where it has to be used. The declaration in a buffer does not affect the persistency of the variable. However, once declared in an Terminal command, the variable remains persistent irrespective of all re-declarations. A local variable, on the other hand, cannot be declared by an Terminal command. For example, assuming that you are communicating with the controller through the **Communication Terminal** window and want to start data collection without preparing a special ACSPL+ program. The following dialog occurs:

`global Data(2)(1000)`	!Declare persistent global array of size 2x1000
`dc Data,1000,1,FPOS(0),PE(0)`	!Collect Feedback Position and Position Error of 0 axis motor, 1000 samples, 1 millisecond sampling period
`?S_ST`	!Query the System State variable
`3 ON Data Collection (#DC)`	Response to query: data collection is in progress.
`?S_ST`	!Query the System State variable
`3 OFF Data Collection`	Response to query: data collection is off.
`?Data`	!Query the Data array

3.4 Arrays and Indexing

3.4.1 Scalars and Arrays

A variable can be either scalar or array. A scalar variable contains a single value. An array contains a set of values. The arrays are subdivided into one-dimensional arrays and two-dimensional arrays or matrices.

Before an array is used, its size must be declared. The size of a one-dimensional array is the number of elements in the array. The size of a two-dimensional array is determined by multiplying each dimension, for example 2x2000 (for a total of 4000 elements).

Access to the value of a scalar variable is provided by its name. For example:

`int Scalar1`	Declare **Scalar1** as a local integer
`Scalar1 = 4`	Assign 4 to **Scalar1**
`disp Scalar1`	Display a value of **Scalar1**
`disp Ar2(1)`	Display value of column 1 of the **Ar2** matrix

A typical use of an array requires access to a specific element of the array. To specify an element of an array the array name must be followed by index specification. For example:

`int Ar1(100),Ar2(3)(100)`	Declare local integer **Ar1** as a vector of size 100 and local integer **Ar2** as a matrix of size 3x100
`Ar1(4) = 3000`	Assign 3000 to element 4of **Ar1**
`Ar2(0)(99) = 20`	Assign 20 to the element of **Ar2(0,99)**
`disp Ar2(0)(99)`	Display the element of **Ar2(0,99)**

Indexing of arrays starts from zero. In the example above the first element of **Ar1** has index 0, the last element has index 99.

For information on saving user arrays in the nonvolatile memory, see **Section 2.3.7 - Nonvolatile Memory and Power Up Process**.

3.4.2 ACSPL+ Array Variables

ACSPL+ variables include both scalar and one-dimensional arrays. For example, the variable **S_ST** (System State) is scalar, while the variable **FPOS** (Feedback Position) is a one-dimension array of size 8.

Many ACSPL+ arrays are sized according to the axis number (eight elements). Such arrays contain one element per each controlled axis or motor. For example, each element of read-only variable **FPOS** reads a feedback position of the corresponding motor. **FPOS(0)** provides the position of the 0 motor, **FPOS(1)** provides the position of 1 motor, and so on.

Other ACSPL+ variables are related to the program buffers and therefore are sized according to the number of buffers. For example, each element of **PRATE** specifies a program rate of the corresponding buffer.

3.4.3 Explicit Indexing

Explicit indexing is applicable to both standard and user arrays. To access a specific element of an array, the array name must be followed by one or two indexes. Each index must be enclosed in parenthesis. A one-dimensional array requires one index and two-dimensional array requires two indexes.

The following is the syntax of index specification:

- **(expression)** - One dimensional array
- **(expression) (expression)** - Two dimensional array

For example:

`real Ar1(100),Ar2(2)(8)`	Declare **Ar1** as a local real vector of size 100 and **Ar2** as a local real matrix of size 2x8
`int J`	Declare **J** as a local scalar integer
`Ar1(0) = 0`	Assign 0 to the first element of **Ar1**
`J = 0`	Assign 0 to **J**
`loop 99`	Repeat 99 times
`Ar1(J+1) = Ar1(J) + 1`	Fill **Ar1** with a sequence of numbers 0-99
`J = J + 1`	Increment
`end`	End of loop
`J = 0`	Assign 0 to **J**
`loop 8`	Repeat for each axis
`Ar1(0)(J) = FPOS(J)`	Store current **FPOS** in **Ar1(0,J)**
`Ar1(1)(J) = PE(J)`	Store current **PE** in **Ar1(1,J)**
`J = J + 1`	Increment
`end`	End of loop

3.4.4 Postfix Indexing of Standard Arrays

Postfix indexing consists of a number appended to a standard array name without parenthesis. For example, **FPOS0** is equivalent to **FPOS(0)**; both read the feedback position of the 0 motor. **ERRI1** is equivalent to **ERRI(1)** that specifies a tolerable position error for the 1 motor.

Postfix indexing is applicable to any standard array. However, user variables cannot accept postfix indexing.

Postfix indexing is convenient if an element of an ACSPL+ array is used in an application as a separate variable. For example, elements of ACSPL+ arrays **V** and **I** are often used as counters and temporary variables. Postfix indexes allow using the elements as separate variables such as **V0**, **V22**, **I99**.

Arrays **IN** and **OUT** (Digital Inputs and Digital Outputs) are also often used with postfix indexing. The typical access to a digital input looks like:

```
IN0.4
```

where the first number after name **IN** is a postfix index and selects a group of 32 inputs. The number after the period is a bit specifier used to select one of the 32 bits (in this case bit #4).

Note that explicit indexing can be either constant or expression, while postfix indexing is always constant.

Postfix indexing and explicit indexing can be used interchangeably in one program. However, it is recommended that you select one style and use it throughout the application. If an application requires non-constant indexing of axis-related variables, neither axis-like or post-fix indexing can be used.

3.4.5 Axis Indexing

Each controller axis has an index. The index is an integer number from 0 to *Number_Of_Axes-1*.

Note	
	Number_Of_Axes *is defined by the controller model and cannot be changed, for example, if there are 3 supported axes in a specific controller specification, 0 designates the first axis, 1 the next, and 2 the last.*

The index of an axis is fixed, for example, **index 2** always refers to a specific physical axis. For example:

VEL(1) – stores the velocity of the axis designated as 1.

ACC(5) – stores the acceleration of the axis designated as 5.

3.4.6 User-Defined Axis Names

You can assign names to the axes, and once assigned, the names are used as aliases throughout your program. You do this through the command: **axisdef**.

Axis names follow the general rules for ACSPL+ names: a valid name can be any sequence of letters and digits, but must start with a letter. However, some limitations on the axis names are recommended (see below).

The command can be used to define one or more axis names, for example, the command:

```
axisdef X=23, W12=9, T9=0
```

defines the name **X** for axis 23, **W12** for axis 9, and **T9** for axis 0.

axisdef can be repeated many times as you like to define all required names; however, the following restrictions apply:

- Only one name can be defined for the same axis
- The names must be unique, i.e., you cannot define two axes with the same name
- The name must not conflict with any other name of variable, label, keyword, etc.

A compilation error occurs if one of the above restrictions is not satisfied.

Axis names must be defined either in D-Buffer - see **Section 2.3.3 - Declaration Buffer (D-Buffer)**, or in a program, where it is used. In any case, the axis definition has global scope; therefore the definition of the same axis in a different program must be identical (similar rules apply to global variables). Axis defined in D-Buffer can be used in any other buffer without re-definition.

3.4.6.1 Axis Name as Symbolic Constant

The axis name can be used in expressions as a symbolic constant. For example, given a program that includes declaration:

```
axisdef Q=3
```

the following command

```
VEL(Q)=1000;
```

assigns 1000 to the required velocity of axis 3.

3.4.6.2 Axis Name in Indexing

In axis-related ACSPL+ standard array variables that contain *Number_Of_Axes* components, one component per each controller axis, where the Index of the array ranges from 0 to Number_Of_Axes-1. User axis names can be used for indexing, not only for explicit indexing, but also for prefix and postfix indexing, for example: given the program includes declaration:

```
axisdef Q=3, X1=12, X2=13
```

Table 6 provides examples of the possible index formats:

Table 6 Index Formats

Explicit Indexing	Postfix Indexing
VEL(3) or VEL(Q)	VEL3
ACC(12) or ACC(X1)	ACC12
SLVKI(13) or SLVKI(X2)	SLVKI13

3.4.6.3 Axis Specification in Commands

Another use of user-defined axes are arguments specifying a set of axes in ACSPL+ commands (like **enable**, **kill**, **ptp**, etc.) which are considered as integer arrays. You can declare an array, assign its elements, and then use it as a predefined axis group. For example, assume the application frequently uses axes 1, 12, and 15 as a group. The program may contain the following commands:

```
int AxisGroup(3)
AxisGroup(0)=1; AxisGroup(1)=12; AxisGroup(2)=15;
enable AxisGroup
ptp AxisGroup,1000,1500,1200
…
halt AxisGroup
disable AxisGroup
```

Other formats of axis specification are also supported (actually, they are considered as special forms of array specification):

1. Axis expression, like (0, 1, 2), (Ax1, Ax2, Ax3, Ax4).
2. The keyword: **all** that specifies all available axes.

3.4.7 Array Processing Functions

ACSPL+ provides the following functions for processing arrays:

- **min** – finds the minimum value in an arrary or any section of it
- **mini** – finds the minimum value in an arrary or any section of it and returns its index
- **max** – finds the maximum value in an arrary or any section of it
- **maxi** – finds the minimum value in an arrary or any section of it and returns its index
- **avg** – finds the average value in an arrary or any section of it
- **fill** – fills an array or section of it with the specified value

These functions are detailed in the *SPiiPlus Command & Variable Reference Guide*.

3.5 Using Variables

3.5.1 Querying Variables

The value of any variable can be displayed in response to a query command. See *SPiiPlus Command & Variable Reference Guide* for a description of query commands.

Examples:

`?FPOS`	Query feedback positions of all motors
`1003 4001 233000 1 0 -1 0 1`	
`?FPOS0`	Query feedback positions of the 0 axis motors
`1003`	
`?FPOS0,FPOS2`	Query feedback positions of the 0 and 2 axes motors
`1003`	
`233000`	
`?GlobVar`	Query global user variable **GlobVar**
`23.35`	
`?0:LocVar`	Query local user variable **LocVar** from buffer 0
`100`	
`?0:LocVar,5:LocVar`	Query two local user variables from different buffers
`100`	
`233.7`	

3.5.2 Variables as Operands in Expressions

Variables can be used as operands in expressions.

An array, as a unit, cannot be an operand in an expression. Only the array elements can be used. Therefore the array name must be fully indexed to provide access to a specific element.

In addition, a bit specifier can be added to an integer variable to provide access to a specific bit (see the fourth example below).

An expression is calculated each time that a command that includes the expression is executed. During calculation of the expression each variable is substituted with its current value.

Examples:

`RPOS(0) - FPOS(0)`	An arithmetical expression which produces the position error of the X motor.
`RPOS(0) < FPOS(0)`	A logical expression which produces: • 0 - if the 0 axis position error is positive • 1 - if the 0 axis position error is negative
`2*(VEL(3) - LocVar)`	An expression that combines standard and user variables.

`(IN0.3|IN0.4)&^IN0.6`	A logical expression that uses bit specifiers to access specific digital inputs.
`V0`	An expression that contains only one variable with no operations.

3.5.3 Variables as Arguments in Command or Function

Variables can be used as arguments in commands and functions.

Arguments used to specify commands and functions have specific requirements. In a typical example, an argument is required to be an expression. In this case a single variable can be used as a simplest case of expression. Use of variables in expressions is discussed in **Section 3.5.2 - Variables as Operands in Expressions** above.

A number of commands and functions require a variable or an array as one of their arguments. For example, the first argument of the **dc** command (Data Collection - see **Section 8.1.1 - dc Command**) is an array that accumulates the collected data. Other examples are statistical functions, such as **max**, **min**, and **avg**, that process the whole array specified as an argument. Unlike their use in expressions, an array name without indexes specifies the array as a whole.

For example, the following fragment collects data 1000 times on the **FPOS** for the 0 axis at intervals of **V0**:

```
dc Data,1000,V0,FPOS(0)
```

Where:

1. The first argument of the **dc** command is required to be an array, in this case it is the user array, **Data,** that is specified without indexes, as a whole.
2. The second argument is an expression that defines the number of samples to be collected, in this case it is a simple expression: the constant 1000.
3. The third argument is an expression that defines the sampling period. The **V0** variable is a simple expression. Using a variable instead of an integer provides changing the sampling period, based on **V0**, from one execution of this data collection to another.
4. The fourth argument must be a variable or an array element. The values of the variable (in this case **V0**) will be collected in the array. The syntax of **dc** requires that the fourth element be a variable or an array element. Neither a general expression nor an array without indexes can be specified. An array without indexes neither can be specified. **FPOS(0)** addresses element 0 of the array **FPOS** that corresponds to the 0 axis feedback position.

3.5.4 Variables in ACSPL+ Terminal Commands

The Communication Terminal execution of the ACSPL+ commands is described in **Section 2.4.4 - Immediate Execution**. A command immediately executed is not included in a program or stored in a buffer.

Use of variables in immediate ACSPL+ commands is limited to ACSPL+ and global variables. Local variables cannot be referenced in immediate ACSPL+ commands.

3.5.5 Accessing Variables by Tags

3.5.5.1 Variable Tags

The controller supports integer numerical tags for all standard ACSPL+ variables. You are also able to define tags for user-defined variables. The ACSPL+ program is able to access a variable by a *tag* as described in the next section. This feature enables you to assign your own names to ACSPL+ standard variables thereby making your coding more understandable.

- ACSPL+ Standard Variable Tags

  The tags of standard ACSPL+ variables are fixed in all firmware versions, and you cannot change the tag of a standard variable. For the tag of any given ACSPL+ variable see *SPiiPlus Command & Variable Reference Guide*.

- User-Defined Variable Tags

  A user-defined global variable can be declared with a tag.

The extended syntax of a variable declaration is:

global {int|real} tag tag_number variable_name [,variable_name, variable_name,...]

Where:

tag_number	A positive interger to be associated with the variable(s).
variable_name	A unique variable name, or list of names.

Only global variables can be declared with a tag. The following conditions also apply:

- The tag is not mandatory in the variable declaration. However, if the tag is not declared, the variable cannot be accessed by tag.
- The value of **tag_number** must be greater than 1000. Values below 1000 are reserved for the standard ACSPL+ variables.
- The **tag_number** value must be unique in the application.
- If more than one **variable_name** is included, when the program is compiled, the controller builds a sequence of tag numbers: the specified **tag_number** is attached to the first variable in the list, **tag_number**+1 value is attached to the second variable, **tag_number**+2 to the third, and so on.

3.5.5.2 getvar and setvar Functions

The controller implements two functions:

Syntax:

real getvar(tag_number[, index1, index2])

and

setvar(value, tag_number[, index1, index2])

Where:

tag_number	A positive interger associated with the variable.
index1, index2	Integer indexes. The indexes must be omitted if the variable is scalar. The second index must be omitted if the variable is one-dimensional array.

The **getvar** function reads the current value of the variable and returns it as a real value. The **setvar** function assigns the specified **value** to the variable designated by **tag_number**.

Though the **value** argument and the return value are defined as real, the functions can be used for integer variables as well. The controller implements all necessary transforms automatically.

The functions provide read/write access to all standard ACSPL+ variables and to those user-defined variables declared with **tag**.

3.6 ACSPL+ Functions

For a complete description of all ACSPL+ functions see the SPiiPlus Command & Variable Reference Guide.

3.7 Expressions

3.7.1 General

An expression is a sequence of operators and operands that specify a calculation.

Expressions serve as building blocks for many ACSPL+ commands. For example, assignment commands include expressions to the right of the equal sign:

```
V0 = V1 + 2*(V2 - V3)
```

When the controller executes a command that includes an expression, the expression is calculated and the result is used as required by the command.

Complexity of expression ranges from the simplest expressions that include only one constant or variable name, to extended formulae containing multiple operators and functions with several levels of brackets. For example:

`I99 = 5`	5 is a simple expression
`I98 = V0`	V0 is a simple expression
`I97 = ((I1-1)*sin(I2)+2)/5`	Complex expression

3.7.2 Calculation Order

In a complex expression the following factors, listed below in priority order, determine the order of calculation:

1. Brackets in the expression
2. Operator precedence
3. Order of operators in the expression (left-to-right order)

If the brackets do not unambiguously define the calculation order, then the operator precedence is taken into account, and if the calculation order is still ambiguous then the left-to-right calculation order is applied.

Table 7 summarizes the operator precedence, summarizing them in order of precedence from highest to lowest.

Table 7 Mathematical Operators

Operator	Operation
. (dot)	Bit selection
- ~ ^	Unary minus, Inversion, Logical not
* /	Multiplication, Division
+ -	Addition, Subtraction
= <> < > <= >=	Compare
& | ~	Logical and bitwise AND, OR, XOR

If several operators appear on the same line or in a group, they have equal precedence.

3.7.3 Expression Type

The controller uses two types of numerical values;

- Integer values that are 4-bytes (32-bits) long and range from -2147483648 to 2147483647
- Real values that are 8-bytes long, 53-bits of mantissa and 11-bits of exponent with a range of $\pm 10^{-1023}$

Expression calculations produce either integer or real values.

Within the group of expressions that produce integer values, there is a subset, called logical expressions, that consists of expressions that produce only two values: 0 or 1.

According to the type of result, an expression is defined as integer, real or logical. The controller provides automatic type conversion of the result so that expression type required does not restrict expression use. For example:

```
V0 = 0.01 * V1
```

In this assignment command, the integer variable **V0** is to the left of the equals sign, while a real expression is to the right of the equal sign. The controller automatically uses rounding type conversion of the result obtained on the right side, so that the real value can be converted to an integer variable and stored.

The types of operands and operators used in the expression define the expression type. Each operator has an associated rule that defines the type of result according to the types of operands. The rules are summarized in the following tables.

Note	*Integer (0,1) is specified for the operators that produce only two values: 0 or 1 (logical result).*

3.7.4 Operands

An operator requires one or two operands.

- An operator that requires one operand is called a unary operator. A unary operator is placed before its operand.
- An operator that requires two operands is called a binary operator. A binary operator is placed between its operands.

Unary operators:

Operator	Type of Operand	
	integer	real
- (unary minus)	integer	real
~ (inversion)	integer	integer
^ (logical not)	integer (0,1)	integer (0,1)

Binary operators:

Operator	Type of Operand			
	integer-integer	**integer-real**	**real-integer**	**real-real**
+ (addition)	integer	real	real	real
- (subtraction)	integer	real	real	real
* (multiplication)	integer	real	real	real
/ (division)	real	real	real	real
& (and), | (or), ~ (xor)	integer	integer	integer	integer
= , <>, <, >, <=, >= (compare)	integer (0,1)	integer (0,1)	integer (0,1)	integer (0,1)
. (bit selection)	integer (0,1)	integer (0,1)	integer (0,1)	integer (0,1)

Each operand can be either integer or real. If the operator requires, the controller automatically converts the operand to the required type.

Operands can be one of the following:

- Constant

  A constant can consist of any integer or real number. Integer constants can be presented in decimal, hexadecimal and binary notation. Real constants can include a decimal point and/or exponent.

- Symbolic constant

  Symbolic constants are predefined in the controller. Each symbolic constant presents an integer number.

- Scalar variable name or Array name with indexing

  A variable name is a name of any standard or user-defined variable. If a user-defined name is used, it must be declared before being used in the program. If the variable presents an array, the name must be fully indexed to specify one element of the array.

- Function call or Expression

  Any ACSPL+ function can be used in an expression. Using expression as an operand of other expression provides unlimited variety of expressions. In many cases expression used as operand must be enclosed in brackets. For example:

`(V0+5)*7`	Expression V0+5 is the left operand of multiplication
`(V0+5)*(I88+3)`	Both operands of multiplication are expressions
`V1*3 + I2*6`	Operands of addition are expressions V1*3 and I2*6

The following sections provide further elaborations on the operands and operators.

3.7.4.1 Arithmetical Operators

Arithmetical operators provide four arithmetical operations. The arithmetical operators are:

- + (addition)
- - (subtraction)
- * (multiplication)
- / (division)

Addition, subtraction and multiplication calculate an integer result if both operands are integers, and a real result if at least one operand is real.

Division always calculates real results, for example:

`real Var`	Declare real variable **Var**
`Var = 5/4`	Assigns 1.25 to variable **Var**.

3.7.4.2 Compare Operators

Compare operators are:

- = (equal to)
- <> (not equal to)
- > (greater than)
- >= (greater than or equal to)
- < (less than)
- <= (less than or equal to).

Compare operators work with any combination of integer and real operands. Compare operator results are always the integers 0 or 1. A positive result of a comparison provides value 1, while a negative result provides value 0.

Compare operators are typically used in a variety of contexts, for example:

```
if V0 > 5 ...
while I99 <> 0 ...
on IN0.5 = 1 ...
```

Because they always produce an integer result, compare operators can be used in arithmetical calculations, for example:

```
V1 = (V0 = 5) * V4
```

The expression in parenthesis, **V0 = 5**, results in 1 if **V0** is equal to 5 and results in 0 if **V0** is not equal to 5. Therefore, this command assigns **V1** with the current value of **V4** if **V0** equals 5, and assigns **V1** with 0 if **V0** has any other value.

3.7.4.3 Bitwise and Logical Operators

Bitwise and logical operators are:

- **&** (and)
- | (or)
- ~ (xor - exclusive or)

The result of a bitwise operator is always an integer. If an operand of a bitwise operator is real, the controller automatically converts it to an integer before the operation.

Bitwise means that the operation is executed separately on each bit of the operand. Each integer operand is considered as a set of 32-bits. Bit 0 of the left operand is combined with bit 0 of the right operand to produce bit 0 of the result.

The following example illustrates the AND operator. Both operands and the result are considered as sets of 32-bits.

First operand	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0
Second operand	0	0	1	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0	0	0	1	1	1	1	1	0	0
Result	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0

If both operands are logical, i.e., have only values of 0 or 1, the result is also logical. In this case the operators can be treated as logical AND, OR and XOR. For example, the following condition:

```
(V0 = 5) & (I30 = 0)
```

is satisfied only if **V0** is 5 and **I30** is 0 simultaneously.

3.7.4.4 Unary Operators

The unary operators are applied only to the operand. The unary operators are:

- **-** (unary minus) – Negates its operand, either integer or real. The type of result follows the type of operand. For example:

```
V0 = -V1
```

V0 is assigned with the negative value of **V1**

- **~** (inversion) – Provides bitwise inversion of its operand. If the operand is real, the controller provides automatic conversion to integer before the operation. The result is always an integer.
- **^** (logical not) – Accepts either an integer or real operand and calculates an integer result. The result is 1 if the operand equals to 0, and is 0 if the operand is non-zero.

The following two examples illustrate the difference between the **~** and **^** operations:

X	0	0	1	1	1	1	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	1	1	1	1	1	0	0
~X	1	1	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	1	1
^X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
~X	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
^X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Note	*Inversion is a bitwise operator, while the logical not applies to the entire value.*

3.7.4.5 Bit Selection Operator (Dot)

Bit selection operator (dot) extracts one bit from an integer number. The result is always an integer and can yield only two values: 0 or 1.

The operands can be integer or real. The controller converts a real operand to integer before the operation.

The left operand supplies a number from which a bit will be extracted. The number is treated as a set of 32 bits numbered from 0 to 31. Bit 0 is the least significant bit.

The right operand supplies the ordinal number of the bit to be extracted. The value of the right operand must be in the range from 0 to 31.

Typical use of the bit selection is for the flag variables, such as **MST**, which are treated as a collection of one-bit flags. For example, the following command

```
till ^MST(0).#MOVE
```

provides waiting for the end of the 0 axis motion. The symbolic constant **#MOVE** provides selection of bit 5 of the **MST** (Motor State) variable. The fifth bit of **MST** reflects the state of the motion: it is 1 while the axis is in motion, and is 0 while the axis is idle.

3.7.5 Character Constants

An ASCII character enclosed in single quotation marks is interpreted as integer constant and can be used in assignment and expressions as a normal integer constant. For example:

```
int Char
Char = 'A'
```

assigns value 65 (or 0x41, which is the numerical equivalent of A) to the variable: **Char**.

3.8 ACSPL+ Commands

The complete ACSPL+ command set is detailed in the *SPiiPlus Command & Variable Reference Guide*. This section addresses certain ACSPL+ commands with expanded information on their use.

3.8.1 Assignment Command

The Assignment (=) command is used to give a variable a value.

Syntax:

left-term = expression

left-term can be:

- Standard or user variable
- An element of a ACSPL+ or user array
- One bit of integer variable or integer array element

Assigning to an ACSPL+ variable is limited by the following rules:

- Assignment to read-only variable (for example, **FPOS**) is prohibited
- Assignment to a protected variable (for example, **ERRI**) is allowed in only in the Configuration mode.

expression can be of integer or real type. By using different operators and parenthesis, an unlimited number of expressions can be constructed.

After assignment, the previous value of the variable is replaced by the new value.

The controller executes assignment commands in the following order:

1. Calculate **expression**
2. Convert the type of calculated value to the type of **left-term** (if the types differ)

3. Assign the result to **left-term**

The following sections explain assignment for specific types of **left-term**.

3.8.1.1 ACSPL+ Variable Assignment

ACSPL+ variables can be used in the left side with the following restrictions:

1. The variable must not be read-only. Using read-only ACSPL+ variable in the left side causes a compile-time error.
2. If it is a protected variable, protection is checked when the command is executed. In protected mode, the assignment fails, producing a run-time error (3077).

Examples:

The following command tries to assign a read-only variable, and will cause a compilation error:

```
FPOS(0) = 0
```

The following command assigns the protected variable **FMASK**.

```
FMASK(0) = 0
```

is a legal command; however, when the command executes, if the controller is in the Protected mode, the assignment fails and produces a run-time error.

- ❑ If the ACSPL+ variable is scalar, no indexing is required.
- ❑ If a ACSPL+ variable is an array, explicit or implicit indexing is required. For indexing of ACSPL+ variables see **Section 3.4.3 - Explicit Indexing** through **Section 3.4.4 - Postfix Indexing of Standard Arrays**.

Examples:

`VEL(0) = 1000`	Assign 1000 to 0 axis default velocity - explicit indexing
`VEL0 = 1000`	The same as above - postfix indexing
`Var1 = FPOS(0)`	Assign to user variable
`Var2(0)(5) = 200`	Assign to element of user array
`OUT0.5 = 1`	Assign to digital output 5

3.8.1.2 User Variable Assignment

User local and global variables must be declared before they can be used in an assignment command.

- ❑ Explicit indexing only is allowed for user array variables.
- ❑ If a user variable is scalar, no indexing is required.
- ❑ If a user variable is one-dimensional array, it requires one index. Two-dimensional arrays require two indexes.

Examples:

`global int Int_Scalar`	**Int_Scalar** is declared as a global integer variable
`local real Real_Array1(20), Real_Array2(10)(10)`	**Real_Array1** is a local real array of 20 elements. **Real_Array2** is a local real array of size 10x10.
`Int_Scalar = 5`	Assign 5 to **Int_Scalar**
`Real_Array1(2) = 5*Int_Scalar`	Calculate **Int_Scalar** multiplied by 5 and assign the result to the third element of **Real_Array1**
`Real_Array2(Int_Scalar)(Int_Scalar+2) = 1000`	Assign 1000 to the element of **Real_Array2** with first index equal to **Int_Scalar** and the second index equal to **Int_Scalar+2**

3.8.1.3 Bit Assignment

You use a bit specifier, added to an integer variable or integer array element, to provide assignment to a single bit. The syntax of bit specifier is

VAR.bit_specifier = expression

The **expression** must calculate to an integer number that provides an ordinal number. Typically the **expression** is simply a constant that specifies the number. However, an arbitrary expression can be specified that provides calculation of the bit value in run time.

In the controller, an integer number is presented by 32 bits. The bits are numbered from 0 to 31. The least significant bit is bit 0; therefore, **bit_specifier** must be an integer number in the range 0-31.

A bit can have only two possible values: 0 (false) or 1 (true), while the **expression** result, which defines the bit value, can be any value. Assignments convert the value as follows:

- If the value is zero, the bit is set to zero
- If the value is non-zero, the bit is set to one

Although bit assignments are applicable to any integer variable or array element, they are mainly used for changing flag variables and output bits.

Examples:

`OUT0.13 = 1`	Set output 13 to one.
`IST(0).#IND = 0`	Reset index flag of 0 axis.
`FMASK(0).#DRIVE = 1`	Enable Drive Fault exception. The command is allowed only in the Configuration mode.

3.8.1.4 Type Conversion

Left-side terms and right-side expressions may be of different types: integer or real in any combination. (The special case of Bit assignment is handled differently, as is described in Bit Assignment, **Section 3.8.1.3 - Bit Assignment**).

If the types differ, the type of calculated right-side expression is automatically converted to the type of left-side term. There are two possible conversions:

- Integer to real; conversion is exact.
- Real to integer; conversion is not always exact. A real number is rounded to the closest integer.

3.8.2 Synchronization Commands

Synchronization commands provide delay in program execution for a specified number of milliseconds or until a specified condition is satisfied. The following synchronization commands are available:

- **wait**
- **till**

3.8.2.1 wait Command

The **wait** command delays program execution for a specified number of milliseconds.

Syntax:

wait expression

The **wait** command executes in the following order:

1. Calculate expression. The result is the required delay time in milliseconds.
2. Delay the program for the calculated amount of time.

Typically the expression is specified as a constant that provides a constant delay time. However, in some cases a variable delay time may be needed as shown in the example below.

If the wait command is located in a separate line, the total execution time of this line is the delay time plus the standard one line execution time as defined by the **PRATE** variable, which defines the program execution rate (see *SPiiPlus Command & Variable Reference Guide* for details on the **PRATE** variable).

Example:

The following example is a program that tests the **wait** command:

```
V0 = 0                          Assign 0 to V0
loop 100
 V1 = TIME
 wait V0
 disp TIME - V1
 V0 = V0 + 1
end                             End loop
stop                            End of program
```

If the controller cycle is 1 millisecond and **PRATE** is 1, the program displays a list of numbers from 2 to 101. The first number, 2, corresponds to the standard execution time of two lines,

because the first time the additional delay provided by the **wait** command is zero. Each loop executed adds one to the requested delay, therefore the displayed time grows correspondingly.

3.8.2.2 till Command

The **till** command delays program execution until a specified expression produces a non-zero (true) result.

Syntax:

till expression [, timeout]

The optional **timeout** argument specifies a timeout for the **till** command in milliseconds.

The **till** command executes in the following order:

1. Calculate expression.
2. If the expression result is non-zero, go to the next command.
3. If the expression result is zero, wait one controller cycle and repeat the **till** command execution.

Examples:

The following fragment demonstrates a typical use of till command that provides waiting for a specific state before the execution of the next command:

`ptp 0, 2000`	Start positioning of the 0 axis to absolute point 2000
`till ^AST(0).#MOVE`	Wait until the 0 axis motion finishes

The bit: **AST(0).#MOVE** is raised as long as the 0 axis is involved in a motion. Inversion of the bit (^**AST(0).#MOVE**), causing the bit to become non-zero, occurs when the motion ends for any reason. Therefore the above **till** command provides a delay of execution of the next command until the motion is over.

In the following example, the program starts a data collection and then a motion. The feedback position is sampled with a period of 1 millisecond and stored in the data array. After the data collection finishes, the data array contains a transient process of ptp motion. Synchronous data collection used in the example displays its state in the **AST(1).#DC** bit which is raised as long as the data collection is in progress. The collected data can be safely used only after the data collection process has terminated. The **till** command below validates that both the motion and the data collection are over:

`global real Data(1000)`	Declare global real array Data of 1000 elements
`dc/s 1, Data, 1000, 1, FPOS(1)`	Start data collection of **FPOS(1)** to array **Data**, 1000 samples, 1ms period
`ptp 1, 2000`	Start positioning of the 1 axis to absolute point 2000
`till ^AST(1).#MOVE & ^AST(1).#DC`	Wait until both the 1 axis motion and the data collection finish

The following example provides the 3 axis motion in negative direction until a general purpose input becomes active and then terminates the motion:

`jog 3, -`	Start jog motion of the 3 axis in negative direction
`till  IN0.5`	Wait until input 5 is activated
`halt 3`	Terminate the 3 axis motion

In the following example a general purpose output must be activated 25 millisecond before the motion end. The ACSPL+ **GRTIME** variable (for details on the **GRTIME** variable, see **Section 4.2.3 - The GRTIME Variable**) contains the estimated time that remains to the motion end.

`ptp 0, 10000`	Start positioning of the 0 axis to absolute point 10000
`till GRTIME(0) <= 25 ; OUT0.4 = 1`	Activate output 4 25 milliseconds before the motion ends.

The output activation, **OUT0.4 = 1**, is placed in the same line as the **till** command in order to avoid one controller cycle delay between program lines.

3.8.3 Autoroutines

The technique of autoroutines is similar to hardware interrupts. In distinction to routines that must be explicitly executed (by way of the **call** command), the autoroutine is automatically executed when a specific condition is satisfied. The routine interrupts the currently executing program, executes the commands specified in the autoroutine body, and then returns control to the interrupted program.

3.8.3.1 on Command

The **on** command flags the routine as an autoroutine and specifies the condition upon which the execution of the routine is based.

Syntax:

on expression

The value of **expression** defines the condition. The condition is considered true if the expression calculates to a non-zero result. A zero result corresponds to a false condition, and the routine is not executed.

The controller never executes the **on** command directly.

Note	
	*If the program execution flow hits an* **on** *command, the controller asserts a run time error and aborts the program. Therefore you must either end the program before the* **on** *command, or use an unconditional* **goto** *command to skip over the routine.*

Instead of direct execution, the controller registers an autoroutine when the program containing the routine is compiled. Then the controller calculates the expression each controller cycle in parallel with executing ACSPL+ programs. If the expression calculates to a non-zero value, the controller interrupts the ACSPL+ program being executed in the same buffer where the autoroutine is located, and transfers the execution point to the autoroutine. If no ACSPL+ program is executed in the buffer, the controller does not interrupt any program and simply starts the autoroutine execution.

The controller implements edge-detection in autoroutine condition verification. If a condition becomes true, the controller activates the autoroutine only once. If the condition remains true afterwards, the controller does not activate the autoroutine again. The condition must become false and then become true again in order to activate the autoroutine again.

3.8.3.2 Autoroutine Body and Execution

The autoroutine body is a sequence of ACSPL+ commands that starts from the command following the autoroutine header. The body continues until it reaches a **ret** command.

The **ret** command terminates the autoroutine execution and transfers execution control back to the interrupted program. If no program was interrupted, the **ret** command simply terminates the program.

As explained above, an autoroutine interrupts the program in the host buffer, but is executed in parallel with programs that are executed in other buffers.

You can specify different execution rates (number of lines executed per one controller cycle) for regular programs and for autoroutines in the same buffer. The ACSPL+ **PRATE** array contains elements for each program buffer and specifies the execution rate for regular programs. The ACSPL+ **ONRATE** array specifies the execution rate for autoroutines.

For example, if you have configured the controller so that **PRATE(2)** is one, but **ONRATE(2)** is four, the program in buffer 2 will be executed one line per one controller cycle, and any autoroutine specified in buffer 2 that interrupts the program will be executed four lines per one controller cycle. When the **ret** command that terminates the autoroutine is executed, the controller switches back to the rate of one line per one cycle.

3.8.3.3 Autoroutine and the Host Buffer Interactions

An autoroutine can reside in any program buffer. The controller examines the conditions each controller cycles for all compiled autoroutines in all buffers.

There are, however, specific autoroutine-host buffer interactions:

- The buffer's local variables can be used in the autoroutine condition as well as in the autoroutine body only in an autoroutine defined in the host buffer. However, all ACSPL+ and user global variables can be used in any buffer.
- When the condition is satisfied, the autoroutine interrupts only the program executed in the host buffer. Programs that are concurrently executing in other buffers continue executing in parallel with the autoroutine. When activated, the autoroutine prevents activation of other autoroutines in the same buffer. A program that is executed in any other buffer can be interrupted by an autoroutine specified in its own host buffer.

The following approaches are available to you for defining a set of autoroutines and assigning them to one or more buffers:

- A specific autoroutine occupies a separate buffer with no other program or autoroutine in the buffer. Activating and executing the autoroutine has no direct affect on other programs or autoroutines. This approach is the most suitable for an autoroutine that takes a long time to execute, because a large autoroutine that shares a buffer with another program or autoroutines would prevent the activity of other programs or autoroutines during its execution.
- Several autoroutines are specified in one buffer with no regular program in the same buffer. In this case the activation of an autoroutine does not interrupt any program, all programs executed in the other buffers continue executing concurrently. An activated autoroutine prevents the activation of another autoroutine in the same buffer until its termination.
- One or more autoroutines are specified in a buffer along with a regular program. In this case the activation of the autoroutine interrupts the program execution. This approach is the most suitable if the program and the autoroutine are closely related and must use the same local variables. For example, the autoroutine processes the failure conditions for the program, and must interrupt the program if a failure occurs.

3.8.3.4 Examples

The following fragment demonstrates a typical use of autoroutine for processing the controller faults. The autoroutine provides an error message when the Drive Alarm of 0 axis occurs:

`on FAULT(0).#DRIVE`	Activate autoroutine when bit **FAULT(0).#DRIVE** changes from 0 to 1
`disp "X Drive Alarm"`	Display an error message
`ret`	End of autoroutine

The following autoroutine responds when either the Left Limit or Right Limit are activated:

`on FAULT(1).#LL | FAULT(1).#RL`	Activate autoroutine when the right or left limit bit is activated on the 1 axis.
`disp "1 axis Limit Switch activated"`	Display an error message
`ret`	End of autoroutine

The following example assumes that an extra ventilator must be activated when the motor overheat input signal is activated for the Z axis. The ventilator is controlled by the output bit: **OUT0.4**. The ventilator must be disabled when the signal returns to inactive state.

`on FAULT(2).#HOT`	Activate autoroutine when bit **FAULT(2).#HOT** changes from 0 to 1.
`OUT0.4 = 1`	Set output 4 to 1
`ret`	End of autoroutine.
`on FAULT(2).#HOT`	Activate autoroutine when bit **FAULT(2).#HOT** changes from 1 to 0.

`OUT0.4 = 0`	Set output 4 to 0
`ret`	End of autoroutine.

All bits, not only faults, can be used in autoroutine conditions. Assuming that output **OUT0.6** (of the 0 axis) is connected to a LED indicator, the following autoroutines signals the motion state bit to activate the indicator, and deactivate it when the 0 axis is no longer in motion:

`on MST(0).#MOVE`	When the **MST(0).#MOVE** bit changes from 0 to 1 (signaling that the X axis is moving)
`OUT0.6 = 1`	Set output 6 to 1 (turns on the LED)
`ret`	End of autoroutine.
`on ^MST(0).#MOVE`	When the **MST(0).#MOVE** bit changes from 1to 0 (signaling that the X axis is no longer moving)
`OUT0.6 = 0`	Set output 6 to 0 (turns off the LED)
`ret`	End of autoroutine

The condition of an autoroutine can be any type of expression, not only bit verification. The following autoroutine provides an alarm message if a fault occurs in the controller:

`on S_FAULT`	When a fault occurs
`disp "Something happened"`	Display an error message
`ret`	End of autoroutine

The above autoroutine displays the alarm message only on the first fault. If one fault bit is already raised, and another fault occurs, the second fault does not cause the alarm message.

The ACSPL+ **MERR** (motor error) array can be used for motor failure processing. While a motor is enabled, the corresponding element of **MERR** is zero. If a motor is disabled, the element stores the reason why the motor was disabled. Codes greater than or equal to 5010 correspond to fault conditions. The following autoroutine displays a message when the controller disables the X motor due to any fault.

`on MERR(0) >= 5010`	When the 0 axis motor is disabled
`disp "Motor 0 was disabled. Error code: ", MERR(0)`	Display a message stating that the motor was disabled, and the error code of the fault
`ret`	End of autoroutine

The ACSPL+ **AERR** array can be used to detect abnormal motion termination.

The ACSPL+ **MERR** and **AERR** variables expose only those faults that cause motor disable or abnormal motion termination.

3.8.4 Program Management Commands

Program management commands are used for controlling the execution of a program. As any other command, a program management command can be either executed immediately as an Terminal Command, or stored in a buffer. Using program management commands within a

program provides the ability to create a master program that manages execution of other programs.

3.8.4.1 start Command

The **start** command activates program execution in a buffer.

Syntax:

start buffer_number, label_name

The command specifies a target buffer (**buffer_number**) that contains the program that must be activated. The **buffer_number** argument can be a constant or expression that calculates to integer number. The specified or calculated buffer number must fall into the range 0 to 9. If the number is out of range, error 3052 is generated.

The **label_name** argument is a label in the program (see **Section 3.1.4 - Names: Variable and Label**). Execution starts from that label.

If the **start** command is executed from a program, the specified **buffer_number** must be different from the buffer that contains the current program because a program cannot start itself. It will be aborted, generating error 3044.

The **start** command executes successfully if the target buffer is loaded with a program, compiled, but not running. Otherwise, the **start** command causes a run-time error and aborts the current program.

The program activated by the **start** command executes concurrently with the program containing the **start** command, and other active programs.

Examples:

The following fragment starts the program in buffer 2 from label **PStart**:

```
start  2, Pstart
```
Start executing buffer #2 at the line labeled **Pstart**.

The following Terminal command displays change in buffer state after the **start** command was executed:

```
?2
```
Querying status of buffer #2.

```
Buffer 2: 192 lines, running in line 153
```
Response to query.

3.8.4.2 stop and stopall Commands

The **stop** command terminates program execution in a buffer. The **stopall** command terminates all currently executing programs except the program that issued the command. The syntax is:

stop [buffer_number]

stopall

Where **buffer_number** is the buffer designator (an integer between 0-16)

The **stop** command without **buffer_number** affects the currently executing program in the buffer and is the normal method of program termination.

The **stop** command with **buffer_number** terminates a program in the specified buffer. A master program that manages the whole application can use this command in order to terminate a certain activity.

The **stopall** command executed by a program terminates all other concurrently executed programs, but the program itself continue executing.

After termination by the **stop** or **stopall** command, a program remains in the compiled state. Therefore, if the program contains autoroutines, the autoroutines can be activated after the program termination whenever its condition is satisfied.

Examples:

The following command terminates the current program:

```
stop
```

The following command terminates the program only if the 0 axis is disabled:

```
if  ^MST(0).#ENABLED  stop;  end
```

The following command terminates the program executed in buffer 3:

```
stop  3
```

The following command executed in buffer 0 terminates the programs currently executed in all buffers except buffer 0:

```
stopall
```

The following Terminal command displays change in buffer state after executing the **stop** command:

`?3`	Querying status of buffer #3.
`Buffer 3: 35 lines, compiled, not running`	Response to query.

3.8.4.3 pause and resume Commands

The **pause** command suspends program execution in a buffer. The **resume** command resumes execution of a suspended program.

Syntax:

pause buffer_number

resume buffer_number

The **pause** command suspends the program executed in the specified buffer (**buffer_number**). If no program is executed in the buffer, the command has no effect.

The **resume** command resumes execution of the program suspended in the specified buffer. If the program was not suspended, the command has no effect.

Examples:

The following command suspends the program currently executed buffer 0:

```
pause 0
```

The following Terminal command displays change in buffer state after executing the pause command:

`?0`	Querying status of buffer #0.
`Buffer 0: 97 lines, suspended in line 69`	Response to query.

The following command resumes program execution in buffer 0:

```
resume 0
```

The following Terminal command displays change in buffer state after executing the resume command:

`?0`	Querying status of buffer #0.
`Buffer  0: 97 lines, running in line 83`	Response to query.

3.8.4.4 enableon and disableon Commands

The **enableon** command enables the activation of an autoroutine in a buffer. The **disableon** command disables the autoroutine activation in a buffer.

Syntax:

enableon buffer_number

disableon buffer_number

The commands alter the **NOAUTO** bit in the ACSPL+ **PFLAGS** variable that controls autoroutine activation (see *SPiiPlus Command & Variable Reference Guide* for details on the **PFLAGS** variable).

If the bit is reset, the controller starts verifying the condition of an autoroutine and can activate the autoroutine as soon as the buffer is compiled. Setting the bit prevents the autoroutine activation even if the buffer is compiled and the condition is true.

Examples:

The following dialog shows the effect of the commands on the buffer state:

`disableon  0`	Disabling autoroutines in buffer #0
`?0`	Querying status of buffer #0.
`Buffer 0: 97 lines, compiled, not running, autoroutines disabled`	Response to query
`enableon 0`	Enabling autoroutines in buffer #0
`?0`	Querying status of buffer #0.
`Buffer 0: 97 lines, compiled, not running`	Response to query.

4 ACSPL+ Motion Programming

This chapter provides practical details for using ACSPL+ to program motion. It covers the specific commands for programming motion. It should be used in conjunction with the *SPiiPlus Command & Variable Reference Guide*.

4.1 Axis/Motor Management Commands

Axis/Motor Management commands comprise various operations that change the state of the motors and the axes, establish relations between the motors and the axes, and manage executed motion.

4.1.1 enable & disable Commands

The **enable** command activates one or more motors and drives. After the **enable** command, the motor starts following the reference and physical motion is available.

The **disable** command shuts off one or more drives and motors. After the disable command the motor cannot follow the reference and remains idle.

Syntax:

enable axis_specification

disable axis_specification [, reason]

In simple cases **axis_specification** is a single axis like 0 or 13, or a string consisting of axis enclosed in parenthese and separated by commas, for example: (0, 2, 13), or the keyword **all** (specifying all available non-dummy axes).

The **enable** and **disable** commands affect all specified axes.

The optional second parameter of the **disable** command (**reason**) must be an integer constant or expression and specifies a reason why the motor was disabled. If the parameter is specified, its value is stored in the **MERR** variable. If the parameter is omitted, **MERR** stores zero after the disable operation.

A reason stored in the **MERR** variable is cleared by the **fclear** (see **Section 4.1.4 - fclear Command**) or **enable** command.

As long as the motor is enabled, the controller provides the following:

- Holds output **ENA** (enable drive) in active state.
- Calculates **PE** (non-critical position error).
- Performs closed loop control (for servo motors).
- Examines conditions of **PE** and other faults and raises the corresponding fault bits if required.

The following variables/bits can modify execution of the enable command:

ENTIME	Defines the time (or maximum time) of **enable** execution
MFLAGS.#ENMOD	Defines the mode of **enable** execution
FMASK.#DRIVE	Defines if the drive alarm fault is processed
SAFINI.#DRIVE	Defines an active level of the drive alarm safety signal

If the **MFLAGS.#ENMOD** bit is 1, the **ENTIME** value defines the time of enable execution. In executing the **enable** command, an ACSPL+ program always waits for **ENTIME** milliseconds. If then the drive alarm fault is zero, the **enable** command is considered successful; otherwise the **enable** command fails.

If the **MFLAGS.#ENMOD** bit is 0, the **ENTIME** value defines the maximum time allotted for **enable** execution. Executing **enable**, an ACSPL+ program monitors the drive alarm input signal. As soon as the drive alarm becomes inactive, the **enable** command finishes execution with success. If the drive alarm signal does not change to inactive state within **ENTIME** milliseconds, the **enable** command fails.

Examples:

`enable 0`	Enable axis 0
`enable (2,3)`	Enable axes 2 and 3
`disable 2,5011`	Disable axis 2, store 5011 as a disable reason. Code 5011 corresponds to left limit error, therefore the 2 axis motor will be reported as disabled due to fault involving left limit.
`disable (2,3)`	Disable motors of axes 0 and 3

4.1.2 commut Command

The **commut** command performs autocommutation and may be used when the following conditions hold true:

- ❑ The motor is DC brushless (AC servo)
- ❑ The motor is enabled
- ❑ The motor is idle (not moving)

The **commut** command is used in autocommutation-based startup programs.

The command will operate properly only after the SPiiPlus MMI Adjuster has been used to:

- ❑ Perform initial commutation adjustment
- ❑ Adjust the motor properly
- ❑ Save the adjustment parameters to the controller's flash memory.

Commutation using the SPiiPlus MMI Adjuster is described in depth in the *SPiiPlus MMI Application Studio User Guide*.

Motor movement during commutation very much depends on the motor drive settings. The **commut** command will not operate properly if the **SLPKP** variable is set to zero, or the integrator is very low.

Syntax:

commut axis, [excitation_current,] [settle_time,] [slope_time]

Where:

axis	Specifies the motor to start commutation.
excitation_current	Specifies the motor current used during autocommutation. The current is specified in a percentage of the maximal value. The controller restricts the actual current value by the **XCURI** value. The argument can be omitted in which case the default value is 0.98***XRMS**. You may wish to specify a greater value if the axis static friction is high, or a lower value if the axis static friction is low.
settle_time	Specifies the settling time in the autocommutation process in milliseconds. The argument can be omitted in which case the default value is 500 milliseconds. You may wish to specify a greater value in case of low-bandwidth or slow damping systems.
slope_time	Specifies the time that the excitation current rises from zero to the desired value. The argument can be omitted in which case the default value is 0 providing an immediate build-up of the excitation current. Slope time is required only in special cases and it is usually recommended to omit this argument in which case the excitation current is built instantly.

The **commut** command executes the autocommutation algorithm three times for verification and elimination of unstable equilibrium. The approximate execution time of the command is therefore 3*(**settle_time** + **slope_time**).

It should be noted that:

- In air bearing systems a lower **excitation_current** may be required.
- In high friction systems a higher **excitation_current** value is required.

The **excitation_current** should be the same as that which you determined in the initial commutation adjustment process.

The **settle_time** parameter determines the settling time for the autocommutation process initiated by the **commut** command. The entire autocommutation process lasts approximately three times longer, since the command executes the algorithm three times for verification.

Note	
	*In low-bandwidth systems (high inertia, etc.) a higher value may be required.*

The **settling_time** should be the same as that you have determined in the initial commutation adjustment process.

4.1.3 kill and killall Commands

The **kill** command causes one or more motors to terminate motion using a second-order deceleration profile. The deceleration value is specified by the **KDEC** variable (see *SPiiPlus Command & Variable Reference Guide*).

The **killall** command provides kill operation for all motors.

The commands have the following syntax:

kill axis_specification [, cause]

killall [, cause]

In simple cases **axis_specification** is a single axis like 0 or 13, or a string consisting of axis enclosed in parenthese and separated by commas, for example: (0, 2, 13), or the keyword **all** (specifying all available non-dummy axes).

The optional **cause** argument, specifying a cause why the motor was killed, must be an integer constant or expression that results in an integer. If the parameter is specified, its value is stored in the **MERR** variable. If the parameter is omitted, the **MERR** stores zero after the kill operation.

If several sequential kill operations specify different causes for the same motor, only the first **cause** will be stored in **MERR** and all subsequent causes will be ignored.

A **cause** stored in the **MERR** variable is cleared by the **fclear** (see **Section 4.1.4 - fclear Command**) or **enable** command.

Each motor specified in a kill operation decelerates individually using its **KDEC** deceleration value.

Any motion related to the killed motors is terminated. If a motion involves several motors and only some of the motors are specified in a kill command, all other motors decelerate synchronously using a third-order profile and the **DEC** deceleration value (same behavior as with the **halt** command, see **Section 4.1.8 - halt Command**).

The following examples illustrate **kill** execution under different conditions. (Some of the examples involve a default connection. This is a condition where a motor depends only on the corresponding axis and the difference between motor and axis can be ignored. For more information about default and non-default connections, see **Section 8.12.3 - connect Command** and **Section 8.12.4 - depends Command**.)

- **kill command with motor idle**

  Assume, none of the currently executed motions involves the 2 axis motor.

  The command **kill 2** does not affect the motor in any way.

  The command **kill 2,6100** does not affect the motor, but stores code 6100 (user-defined cause) in the **MERR(2)** variable. The code is stored only if at this moment the variable is zero, otherwise the command does not overwrite the previously stored code and the cause specified in the command is ignored.

- **kill command with single-axis motion, default connection**

  Assume, axis 1 executes motion **ptp/v 1,6000,20000**.

  Once **kill 1** is executed, the motor starts decelerating from its instant velocity using constant deceleration value specified by the **KDEC(1)** variable. Deceleration time is given by:

  V_1 / KDEC(1)

  where V_1 is the instant velocity at the moment of the **kill** command. **V_1** is not necessarily 20000 as specified in the motion command, it can be lower if the **kill** command is executed in acceleration or deceleration phases of the motion.

  Typically, the motor finishes the **kill** process and stops before it reaches the final motion point of 6000. However, if **KDEC(1)** < **DEC(1)** (not recommended in most applications), the motor can overrun the final point.

  The motion is considered to continue execution as long as the kill process is executed. Bit **AST(1).#MOVE** remains 1 while the motor is decelerating and drops to 0 once the motor reaches zero reference velocity. Bit **MST(1).#MOVE** also remains 1 while the motor is decelerating but drops to 0 only when the motor reference velocity is zero and the motor position error **PE(1)** remains less than **TARGRAD(1)** for more than **SETTLE(1)** milliseconds.

- **kill command with several single-axis motions, default connection**

  Assume, each of the axes 0, 2, 4 executes independent single-axis motion.

  The command **kill (0,2,4),6088** is equivalent to **kill 0,6088; kill 2,6088; kill 4,6088** and acts on each motion independently. Each motor uses its own component of **KDEC** and the time of the **kill** process is different for the motors.

  The reason for the **kill**, 6088 (user-defined code), is stored in **MERR(0)** (for the 0 axis), **MERR(2)** (for the 2 axis) and **MERR(4)** (for the 4 axis). However, if at the moment one of these variables contains a non-zero value, the value is not overwritten and the previously stored cause is retained.

  The command **kill 4** (again with default connection) kills the axis 4 motor and terminates the axis 4 motion, but does not affect motors and motions of axes 0 and 2.

- **kill command with multi-axis motion, default connection**

  Assume, motion **mptp (0,1,4)** is executed.

  The command **kill 1** causes the axis 1 motor to start decelerating from its instant velocity using the constant deceleration value specified by the **KDEC(1)** variable. The deceleration continues until the motor reaches zero velocity.

  The behavior of 0 and 4 axes is different. Once the **kill 1** is executed, the motion starts a third-order deceleration process just as if a **halt** command was executed. The 0 and 1 axes continue moving in the common motion. The vector deceleration of the motion is **DEC(0)** and the vector jerk is **JERK(0)**.

  If command **kill (1,4)** is executed, the **kill** process applies to 1 and 4 axes motors. Each motor decelerates independently from its instant velocity to zero using the constant decelerations **KDEC(1)** and **KDEC(4)**. At the same time the 0 axis motor decelerates using the third-order profile and the **DEC(0)** deceleration value, just as if a **halt** command was executed.

If command **kill (0,1,4)** is executed, each motor decelerates independently from its instant velocity to zero using the constant decelerations **KDEC(0)**, **KDEC(1)** and **KDEC(4)**.

In all cases bits **AST.#MOVE** and **MST.#MOVE** of axes 0,1, and 4 remain 1 as long as any of the motor continues decelerating. Once all motors reach zero velocity, bits **AST(0).#MOVE**, **AST(1).#MOVE** and **AST(4).#MOVE** drop to zero. Bit **MST(0).#MOVE** drops to zero as soon as position error **PE(0)** remains less than **TARGRAD(0)** for more than **SETTLE(0)** milleseconds. So do bits **MST(1).#MOVE** and **MST(4).#MOVE** (for the 1 and 4 axes respectively).

- **kill command with non-default connection**

  If a motor is in non-default connection but depends only on the corresponding axis the effect of the **kill** command is similar to the case of default connection. For example, if the connection was specified as

```
connect RPOS(0) = 0.5*APOS(0)*APOS(0)
depends 0,0
```

  (in this case the **depends** command is not necessary), the **kill 0** command starts the same kill process on the 0 axismotor and the halt process on the motion that involves the 0 axis. All above considerations about the idle motor, single-axis motion and multi-axis motion remain the same.

  The result is a little different if a motor depends on another axis or on several axes, for example:

```
connect RPOS(2) = APOS(0) + APOS(2) - APOS(4)
depends 2,(0,2,4)
```

  (in this case the **depends** command is required). The difference is that the **kill 2** command applies the halt operation to all executed motions involving any of the axes 0, 2, or 4. Correspondingly, bits **AST(2).#MOVE** and **MST(2).#MOVE** remain 1 as long as any of these motions continues its termination process.

  Note that a **killall** command always terminates all executed motions and therefore makes no difference between the default and non-default connection.

4.1.4 fclear Command

The **fclear** command clears the current faults and the result of the previous fault stored in the **MERR** variable. The command syntax is:

fclear [axis_specification]

In simple cases **axis_specification** is single axis like 0 or 13, a string consisting of axis enclosed in parenthese and separated by commas, for example: (0, 2, 13), or keyword **all** for all axes.

If **axis_specification** is omitted, the command clears the system faults. If **axis_specification** is specified, the command clears the **FAULT** and **MERR** components for the specified axes.

However, if a reason for a fault is still active, the controller will set the fault again immediately after the **fclear** command.

If one of the cleared faults is an encoder error, the command also resets the feedback position to zero.

4.1.5 set Command

The **set** command determines a current value of the feedback, reference or master position. The command syntax is:

set axis_VAR=expression

Only the following variables can be specified in **axes_VAR**:

FPOS	Feedback Position
F2POS	Secondary Feedback Position
RPOS	Reference Position
APOS	Axis Reference Position

Although the **set** command resembles the **assignment** command, execution of the **set** command is different from **assignment**. The **set** command induces a complex operation in the controller instead of a simple assignment.

Regardless of the left-side variable, execution of the **set** command starts with calculation of **expression**. The result of the calculation provides the right-side value. Then the execution depends on the variable specified on the left side.

The following are examples of the use of **set**.

- **set RPOS** and **set FPOS**

  The **set** command that contains **RPOS** or **FPOS**, shifts the origin of an axis. For example, command

  ```
  set FPOS(0) = 0
  ```

  places the origin of the 0 axis to the point where the motor is located this moment.

  FPOS and **RPOS** provide a reference and a feedback value for a motor. If a control loop works properly, **FPOS** follows **RPOS** with small or zero error.

  If the error is zero, both **set FPOS** and **set RPOS** provide the same result: both **FPOS** and **RPOS** become equal to the right-side value. This is not a simple assignment, and the command adjusts the controller offsets so that the periodic calculation of **FPOS** and **RPOS** will provide the required results.

  If the error is non-zero, the result of **set FPOS** and **set RPOS** may differ slightly. Consider the following example:

`?RPOS(0), FPOS(0)`	Query RPOS and FPOS for the X axis
`6000`	The RPOS and FPOS differ by 2 counts due to, for instance, the bias in the amplifier
`6002`	
`set RPOS(0) = 0`	Set RPOS to 0 for the 0 axis
`?RPOS(0), FPOS(0)`	Query RPOS and FPOS for the 0 axis
`0`	RPOS is set to exact zero
`2`	FPOS is set to 2 in the current point in order to retain the offset between RPOS and FPOS
`set FPOS(0) = 0`	Set FPOS to 0 for the 0 axis

`?FPOS(0), RPOS(0)`	Query RPOS and FPOS for the 0 axis
`0`	FPOS is set to zero in the current point
`-2`	RPOS is set to -2 in order to retain the offset between RPOS and FPOS

Note that in both **set** commands no physical motion occurs. The 0 axis remains in the same position, only the internal offsets in the controller are adjusted to shift the origin as required.

Note further that even if a motor is idle, several identical **set FPOS** commands may place the origin at slightly different points due to the jitter in feedback.

If a motor is flagged by the Default Connection bit (**MFLAGS.#DEFCON**), the **RPOS** and **APOS** variables are conjugate. Therefore, any command that changes **RPOS**, also changes the corresponding **APOS** to the same value.

- **set F2POS**

  The command **set F2POS** shifts the origin of the secondary axis feedback. For example, command

  ```
  set F2POS(0) = 0
  ```

  places the origin of the 0 axis secondary feedback to the point where the motor is currently located.

  As a result of the command execution, **F2POS** becomes equal to the right-side value. This is not a simple assignment, as the command adjusts the controller offsets so that the periodic calculation of **F2POS** will provide the required result (the specified value in the current point).

  Note that even if a motor is idle, several identical **set F2POS** commands may place the origin in slightly different points due to the jitter in feedback.

- **set APOS**

  If a motor is flagged by the Default Connection bit (**MFLAGS.#DEFCON**), variables **RPOS** and **APOS** are conjugate, and always keep the same value. In this case, the **set APOS** command is identical to the **set RPOS** command for the same axis.

  For non-default connection a motor and the corresponding axis are separated. Variables **RPOS** and **APOS** may have different values. In this case, command **set APOS** shifts the origin of the axis but has no effect on the origin of the motor. The controller adjusts offsets so that the command causes no jerk in the motor.

Note	
	*In the case of non-default connection the controller adjusts offsets only for the motors that depend on the specified axis. Therefore, the depends command is significant in a connection specification. If dependence is specified incorrectly, one or more motors can jump once* **set APOS=…** *is executed.*

4.1.6 group, split & splitall Commands

The **group**, **split** and **splitall** commands manage grouping the axes in coordinate systems for multi-axis motion.

- The **group** command creates a coordinate system for multi-axis motion.

  For most applications there is no need for the **group** commans in that the controller automatically creates and splits groups. Mainly you would include the command in order to keep a check that you have included all of the relevant axes in the subsequent motion commands. If you include a motion command that does not relate to all of the axes in the group (without a previous **split** command), the controller issues an error.

- The **split** command breaks down an axis group previously created with a **group** command.

- The **splitall** command breaks down all axis groups previously created with a **group** command.

Syntax:

group axes_specification

split axes_specification

splitall

In simple cases **axes_specification** consists of axis like (0,1,2,4). After power-up, each controller axis is a single axis, no axis group exists. One-axis motion does not require any axis group. One-axis motion can be activated immediately after power-up, assuming that the motor is enabled. Several one-axis motions can be activated in parallel, and do not require any axis group definition.

An axis can belong to only one group at a time. If the application requires restructuring the axes, it must split the existing group and only then create the new one.

For example, the command:

```
group  (0,2,3)
```

creates an axis group that includes axes 0, 2 and 3.

The first axis in the **axes_specification** (0 in the above command) is the leading axis. The motion parameters of the leading axis become the default motion parameters for the group. For example, for the above (0,2,3) group, the values of ACSPL+ variables **VEL(0)**, **ACC(0)**, **DEC(0)**, **JERK(0)**, **KDEC(0)** become the default values of velocity, acceleration, deceleration, jerk and kill deceleration for all motions executed in this group. If, for example, a group was defined as (3,2,0), the 3 axis is leading and the values of 3 will be used as the default motion parameters for 0 and 2.

In all other aspects the leading axis has no preference, and the order of axis letters in group definition is not important. The motion commands referencing a group are not required to specify all axes of the group or in the same order. However, an axis can belong to only one group, so all specified axes must belong to the same group. A motion command that references axes from different groups will fail.

The **split** command must specify the same axes as the **group** command that created the group. After the **split** command the group no longer exists.

Note	
	*If the* **split** *command specifying an axis that is currently in motion is executed within the buffer, the buffer execution is suspended until the motion is completed. However, if the* **split** *command is sent from the host or as a Terminal command, it returns error 3087: "Command cannot be executed while the axis is in motion.*

The **splitall** command breaks up all existing groups. An ACSPL+ program that starts in an unknown environment (not just after power-up) can execute the **splitall** command in order to ensure that no axes are grouped.

4.1.7 go Command

The **go** command starts a motion that was created using the /**w** switch (see **Section 4.6.2 - slave Command**). A motion that has been created without this switch starts automatically after creation and does not require the **go** command.

Syntax:

go axes_specification

In simple cases **axes_specification** is a single axis like 0 or 13, or a string consisting of axis enclosed in parenthese and separated by commas, for example: (0, 2, 13), or the keyword: **all** (specifying all of the axes)**.**

There following possibilities are available:

- **Starting single-axis motion**

  A **go** command specifies one axis that is not included in any group. The command starts the last created motion for the same axis. If the motion was not created, or has been started before, the command has no effect. For example:

`ptp/w 0, 1000`	Create the motion, but do not start it
`go 0`	Start the motion

- **Starting common motion**

  A **go** command specifies a leading axis in a group. The command starts the last created motion for the same axis group. If the motion was not created, or has been started before, the command has no effect. For example:

`ptp/w 0, 1000`	Create the motion, but do not start it
`till IN0.1`	Wait until input 1 is activated
`go 0`	Start the motion

- **Synchronous start of several motions**

  A **go** command can specify several axes. Each axis in the specification must be either a single axis not included in any group or a leading axis in a group. The command synchronously starts the last created motions for all specified axes and groups. If any of

referenced motions was not created, or has been started before, the command does not affect this axis/group but does affect all other specified axes/groups. For example:

`ptp/w (0,1), 1000,1000`	Create the motion, but do not start it
`ptp/w 2, 8000`	Create the motion, but do not start it
`go (0,1)`	Start both motions synchronously

4.1.8 halt Command

The **halt** command terminates a motion using a deceleration profile. The deceleration value is specified by the **DEC** variable (see *SPiiPlus Command & Variable Reference Guide*).

Syntax:

halt axes_specification

In simple cases **axes_specification** is a single axis like 0 or 13, or a string consisting of axis enclosed in parenthese and separated by commas, for example: (0, 2, 13), or the keyword: **all** for all axes.

The following possibilities are supported:

- **Terminating single-axis motion**

  A **halt** command specifies one axis that is not included in any group. The command terminates the currently executed motion for the same axis. If no motion is executed, the command has no effect.

- **Terminating common motion**

  A **halt** command specifies a leading axis in a group. The command terminates the currently executed motion in the same axis group. If no motion is executed, the command has no effect.

- **Terminating several motions**

  A **halt** command specifies several axes. Each axis in the specification must be either a single axis not included in any group or a leading axis in a group. The command terminates currently executed motions in all specified axes and groups. If any of referenced axes are idle, the command does not affect this axis/group but does affect all other specified axes/groups.

4.1.9 break Command

The **break** command provides premature termination of a motion with smooth transition to the next motion. The command executes differently in the following two cases:

1. The next motion already waits in the motion queue. The **break** command terminates the current motion and starts the next motion immediately. The controller provides smooth velocity profile of transition from motion to motion.
2. There is no next motion in the motion queue. The **break** command has no immediate effect. The current motion continues until the next motion appears in the motion queue. At that moment the controller breaks the current motion and starts the next as described above.

If the current motion finishes before the next motion comes to the queue, the command has no effect.

Syntax:

break axes_specification

In simple cases **axes_specification** is a single axis like 0 or 13, a string consisting of axis enclosed in parenthese and separated by commas, for example: (0, 2, 13), or the keyword: **all** (specifies all available non-dummy axes).

Note	*The* **break** *command is not supported in path or master-slave motion.*

The following possibilities exist:

- **Terminating single-axis motion**

  A **break** command specifies one axis that is not included in any group. The command terminates the currently executed motion for the same axis. If no motion is executed, the command has no effect.

- **Terminating multi-axis motion**

  A break command specifies a leading axis in a group. The command terminates the currently executed motion in the axis group. If no motion is executed, the command has no effect.

Caution	***In multi-axis motion, smooth vector velocity profiles do not always assure smooth motion of the coordinates. The user application must provide nearly tangent motion trajectories in the junction point to avoid jumps in coordinate velocity, which may cause damage to equipment.***

- **Terminating several motions**

  A **break** command specifies several axes. Each axis in the specification must be either a single axis not included in any group or a leading axis in a group. The command terminates currently executed motions in all specified axes and groups. If any of referenced axes are idle, the command does not affect this axis/group but does affect all other specified axes/groups.

4.1.10 imm Command

The **imm** command provides on-the-fly changes of the motion parameters: velocity, acceleration, deceleration, jerk, and kill deceleration.

Syntax:

imm motion_var=command

Only the following variables can be specified as **motion_var**:

VEL	Velocity
ACC	Acceleration
DEC	Deceleration
JERK	Jerk
KDEC	Kill deceleration

Although the **imm** command resembles the **assignment** command, execution of the **imm** command differs from normal assignment to the same variables.

As in conventional assignment, execution of the **imm** command starts from calculation of the right-side expression. The calculated right-side value is assigned to the left-side variable. Execution of flat assignment finishes at this point.

The difference between the conventional **assignment** and the **imm** commands becomes apparent when the command executes while a motion is in progress. The **assignment** command does not affect the motion in progress or any motion that was already created and is waiting in a motion queue. Only the motions created after the **assignment** command is executed will use the motion parameters changed by the command. The **imm** command, on the other hand, not only changes the specified variable, but also affects the motion in progress and all motions waiting in the corresponding motion queue. To change a motion on-the-fly, the **imm** command must change a variable of the axis that is a single axis of the motion or a leading axis if the motion is in axis group.

4.2 Point-to-Point Motion

This section covers the commands and relevant ACSPL+ standard variables for Point-to-Point (PTP) motion.

4.2.1 ptp Command

The **ptp** command provides for positioning to a specified target point.

Syntax:

ptp[/switch] axis_designation, target_point [,velocity]

Where **switch** can be one or a combination of:

e	Wait for motion termination before executing next command.
f	Allow specification of non-zero final velocity.
m	Use the maximum motion profile values in the axis group as a whole rather than those of the leading axis.
r	Consider the target point value as relative to the start point.
v	Use the specified velocity instead of the default velocity.
w	Create the motion, but do not start until the **go** command is issued.

In the simplest case the ptp command looks like this:

```
ptp  X, 1000
```

If the axis is moving when the command is issued, the controller creates the motion and inserts it into the axis motion queue. The motion waits in the queue until all motions before it finish, and only then starts.

This command creates a motion of the X-axis to the absolute target point of 1000. If the axis is idle when the command is issued, the motion starts immediately.

If the **e** switch is specified, the controller will wait until the motion terminates before executing the next command. The **e** switch is a convenient substitute for following the **ptp** command with another command that waits for motion termination, for example, the command:

```
ptp/e Y,1000
```

is equivalent to:

```
ptp Y,1000
till ^AST(1).#MOVE
```

Appending the **w** switch to the **ptp** command prevents the motion from starting immediately even if the axis is idle. The command:

```
ptp/w X, 1000
```

creates a motion to the absolute target point 1000, but the motion will not start until the **go X** command is issued.

In the two examples above the value 1000 is an absolute target point. To specify a relative value you use the **r** switch:

```
ptp/r X, 1000
```

This command creates a motion to a relative target point, which will be defined exactly only when the motion starts. When the motion starts, the target point is calculated as the instant axis position plus the specified value. For example, the following two commands

```
ptp X, 1000
ptp/r X, 1000
```

are equivalent to

```
ptp X, 1000
ptp X, 2000
```

In the previous examples the motion executed using the default velocity **VEL(0)** for the specified axis. To override the default velocity you use the **v** switch, as shown in the following example:

```
ptp/v X, 1000, 15000
```

The motion created will ignore the default velocity **VEL(0)** and execute at a velocity of 15000. The default value **VEL(0)** remains unchanged.

You can combine several switches in one command. For example, the command

```
ptp/rv X, 1000, 15000
```

creates a motion to relative target point 1000 using velocity 15000.

The examples shown above specified a single-axis motion. However, the axis, as specified in the **ptp** command, can be either a single axis or an axis group, which means that it can define a multi-axis move. The command

```
ptp  XYT, 1000,2000,3000
```

creates a temporary axis group that includes axes XYT and executes motion in one group. Temporary axis groups are automatically created and split-up by the controller. A group is automatically split-up if the following motion does not address all the axes in the group.

The controller normally takes the motion parameter values from the leading axis. However this can be overridden by using the /**m** switch, which causes the controller to calculate the maximum allowed common motion velocity, acceleration, deceleration and jerk, for example:

```
ptp/m XY, 1000, 2000
```

The calculation examines the **VEL**, **ACC**, **DEC**, **JERK** parameters of the axes involved and the projections of the motion vector to the axes.

Note	
	*If the* **m** *switch is combined with the* **v** *switch, the* **m** *switch is ignored for the velocity and the velocity specified in the command is used. However, common acceleration, deceleration, and jerk are still calculated.*

4.2.2 mptp, point, mpoint, and ends Commands

These commands are used for programming a sequence of multi-point motion.

Syntax:

mptp[/switch] axis_designators [,dwell_time]

point axis_designators, coordinate [,coordinate] [,velocity]

mpoint axis_designators, point_matrix, number_of_points

ends

Where **switch** can be one or a combination of:

w	Create the motion, but do not start until the **go** command has been issued.
v	Use the velocity specified in the command instead of the default velocity.
r	Treat points as relative.
c	Use the point sequence as a cyclic array: after positioning to the last point do positioning to the first point and continue.

4.2.2.1 mptp Command

Use the **mptp** command to specify axis and dwell time:

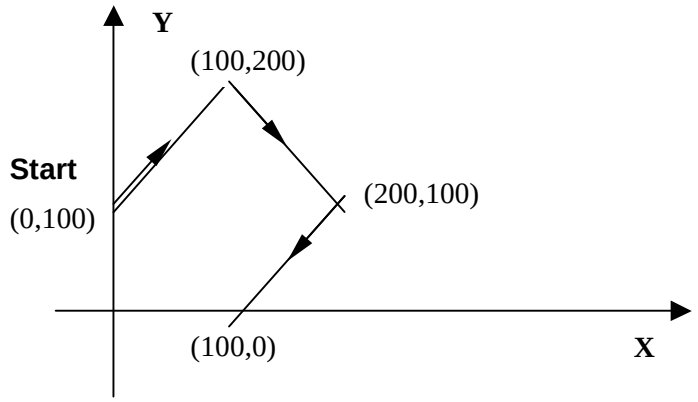
```
mptp X, 1000
```

This command creates a multi-point motion of the X axis and specifies a dwell time of 1000 msec at each point. If dwell is not required, **dwell_time** may be omitted. The **mptp** command itself does not specify any point, so the created motion starts only after the first point is specified. The points of motion are specified by the **point** or **mpoint** commands that follow the **mptp** command.

Consider the following program fragment:

`mptp  XY`	Create multipoint motion in group XY with no dwell time.
`point  XY, 0, 100`	Add first point.
`point  XY, 100, 200`	Add second point.
`point  XY, 200, 100`	Add third point.
`point  XY, 100, 0`	Add fourth point.
`ends XY`	End the point sequence.

The **mptp** command creates the multipoint motion. However, the motion does not start until a point is defined. After the first **point** command the motion starts if all involved axes are idle (not involved in some previous motion), or waits until a motion that is in progress ends, and then starts. The four **point** commands specify the following sequence:



The controller performs sequential positioning to each point. The **ends** command informs the controller that no more points will be specified for the current motion. The motion cannot finish until the **ends** command executes. If the **ends** command is omitted, the motion will stop at the last point of the sequence and wait for the next point. No transition to the next motion in the queue will occur until the **ends** command executes.

Normally, multi-point motion starts with the first **point** command, and the next **point** command executes while the motion is already in progress. However, sometimes you may need to delay starting the motion until all points are defined. You use the **w** switch to prevent the motion from starting until a **go** command executes. The motion created by the command:

```
mptp/w X, 1000
```

will not start until a **go X** command is issued.

Adding the **r** switch to the **mptp** command causes all points to be treated as relative. The first point is relative to the position when the motion starts, the second point is relative to the previous, and so on. The previous example, using the **mptp/r** command, will look like this:

`ptp  XY, 0, 100`	Create PTP motion to the first point (this serves as the reference point).
`mptp/r XY`	Create multipoint motion in group XY with no dwell time.
`point  XY, 100, 100`	Add point.
`point  XY, 100, -100`	Add point.
`point  XY, -100, -100`	Add point.
`ends XY`	End the point sequence.

The **mptp** command uses the default velocity **VEL** for positioning to each point. The **v** switch allows using a specific velocity for each positioning. The desired velocity must be specified in the **point** command after the point coordinates. The previous example is modified for using different velocities:

`mptp/v XY`	Create multipoint motion in group XY with no dwell time.
`point  XY, 0, 100, 30000`	Move to first point at velocity 30000.
`point  XY, 100, 200, 10000`	Move to second point at velocity 10000.
`point  XY, 200, 100, 5000`	Move to third point at velocity 5000.
`point  XY, 100, 0, 10000`	Move to fourth point at velocity 10000.
`ends XY`	End the point sequence.

Several suffixes can be appended to one command. For example, the command:

```
mptp/rv X, 1000
```

creates a multi-point motion with dwell tiime of 1000msec. The points will be specified by relative coordinates, and velocity will be specified for each point.

4.2.2.2 point Command

The **point** command adds a destination point to multi-point or arbitrary motion paths.

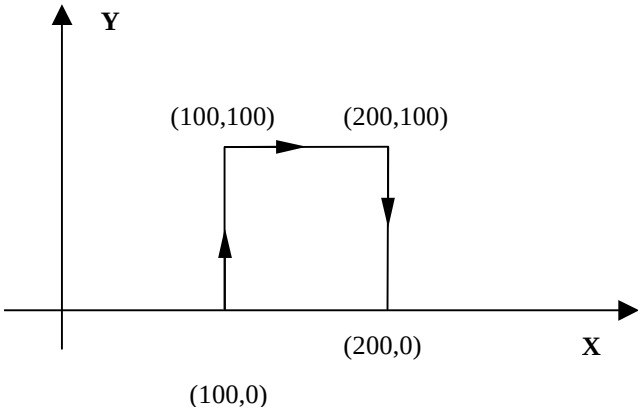
The **point** command does not require a specific value for all axes involved in a multi-point motion. If an axis is not specified in a **point** command, the axis remains idle and retains the previous value. Similarly, if a multi-point motion is created with the **v** switch, the velocity argument in a **point** command can be omitted, and the velocity of the previous segment will be used. If velocity is omitted for the first point, the default velocity **VEL** will be used.

Consider the following example:

`mptp/v  XY`	Create multipoint motion in group XY with no dwell time.
`point  XY, 100, 0, 30000`	Move to first point at velocity 30000.
`point  Y, 100`	Move to second point at velocity 30000.

`point  X, 200`	Move to third point at velocity 30000.
`point  Y, 0, 10000`	Move to fourth point at velocity 10000.
`ends  XY`	End the point sequence.

The four **point** commands specify the following sequence:



4.2.2.3 mpoint Command

The **mpoint** command adds an array of points to either multi-point or arbitrary path motion.

The arguments of the **mpoint** command are:

- **axis_designators** – must specify the same axes in the same order as in the axes-specification of the corresponding mptp or path command.
- **point_matrix** – name of declared two dimensional array.
- **number_of_points** – specifies how many points are added to the motion by the command.

Before the **mpoint** command can be executed, an array must be declared and filled with the point coordinates. Each row of the array contains coordinates of one point.

Note	
	**point_matrix** *must be a two-dimensional array, each column of which containing the specification of one point. The matrix must contain at least* **number_of_points** *columns. If the matrix contains more columns, the extra columns are ignored.*

If the corresponding motion command is **mptp** without the **v** switch or **path** without the **t** switch (see **Section 4.7 - Arbitrary Motion**), a column of the matrix must contain the coordinates of the point. Therefore, if the **axis_designators** includes M axes, the matrix must contain exactly M rows.

If the corresponding motion command is **mptp/v**, the matrix must contain M+1 rows. An additional value in each column specifies the desired velocity for transition from the previous to the current point. The velocity is specified in position units per second.

If the corresponding motion command is **path/t**, the matrix must contain M+1 rows. An additional value in each column specifies the time interval between the previous and the current point. The time is specified in milliseconds.

The following example illustrates how the **mpoint** command can be used for adding points on-the-fly. The example also shows a simple approach to synchronization between the host computer that calculates the points and the controller that executes the motion.

The host computer calculates the desired points and transmits the coordinates via the Ethernet link. The motion involves 6 axes. Therefore, each point specification contains 6 real numbers.

Because transmitting each point separately would be very ineffective in the Ethernet, the host calculates the desired points in advance and transmits them in batches of 50 points. The controller executes the motion. As soon as the controller is ready to accept the next batch of points and the host is ready to send the batch, the next transmission occurs, and so on. The pace of the host and the controller may be very different. However, the host is assumed fast enough to calculate the next 50 points before the controller has moved through the previous 50 points.

The controller executes the following program:

`real Points(50)(6)`	Declare an array of 50 points for each of six axes. The host will write the coordinates to the array.
`int Sync`	Declare synchronization variable.
`mptp XYZTAB`	Create multi-point motion for axes XYZTAB.
`while Sync >= 0`	Continue until the host writes negative number to Sync.
`till Sync`	Wait until the points are received. Once the host has filled the Points array, it writes the Sync variable with a number of points written to the Points array.
`if Sync > 0`	Sync < 0 indicates that the host has finished the point generation.
`mpoint XYZTAB, Points, Sync`	Add points from the Points matrix.
`Sync = 0`	The controller informs the host that the next batch is expected. At this moment the motion through the accepted points has not finished, but the controller is ready to receive more points.
`end`	End if.
`end`	End while.
`ends XYZTAB`	End mptp.
`stop`	End program.

The program running on the host in pseudo-code looks like this:

```
double HPoints(50)(6);
int N, HSync, NBuf;
HANDLE Com;

open communication, start program in buffer NBuf of the controller;

while (Continue)
calculate N (<= 50) points in array HPoints;

acsc_WriteReal(Com, NBuf, "Points", 0, N-1, 0, 6, HPoints, 1000) ;
acsc_WriteInteger(Com, NBuf, "Sync", -1, -1, -1, -1, &N, 0);
do
acsc_ReadInteger(Com, NBuf, "Sync", -1, -1, -1, -1, &HSync, 0);
while HSync;
reset Continue to zero if all points have been calculated;
end;

N = -1
acsc_WriteReal(Com, NBuf, "Points", 0, N-1, 0, 6, HPoints, 0) ;
```

Synchronization between the host and the controller is provided by the **Sync** variable. When the host has finished transmitting the next batch of points to the controller, it writes to **Sync** the number of points in the batch. The controller waits for non-zero **Sync** and then adds the points to the motion. When the controller has added the points, it writes zero to **Sync**, which signals to the host to transmit the next batch of points.

When the host comes to the end, it writes -1 to **Sync** to indicate the end of the motion.

4.2.3 The GRTIME Variable

The ACSPL+ **GRTIME** variable is an 8 element real, read-only array. **GRTIME** displays the time in milliseconds remaining until the end of motion. The **GRTIME** value is valid in **ptp** and **track** motion. For other motion types, the **GRTIME** value is not valid. **GRTIME** behaviour is as follows:

- Each element of **GRTIME** refers to one axis. In a multi-axis motion, only the **GRTIME** element of the leading axis is updated; the elements of other involved axes are zero.
- If an axis is idle, its **GRTIME** element is zero.
- At the beginning of motion, **GRTIME** is not valid and is assigned a large value.
- In Firmware versions previous to Version 4.50, **GRTIME** was invalid during motion phases 1 and 2 (see **Figure 7**). In Firmware Version 4.50, the invalid period is shorter, but its exact duration is not guaranteed. Normally, the period of invalid **GRTIME** is from one to five milliseconds. In the worst case, the period may span phases 1 and 2.

Figure 7 illustrates the **GRTIME** behavior in **ptp** or **track** motion.

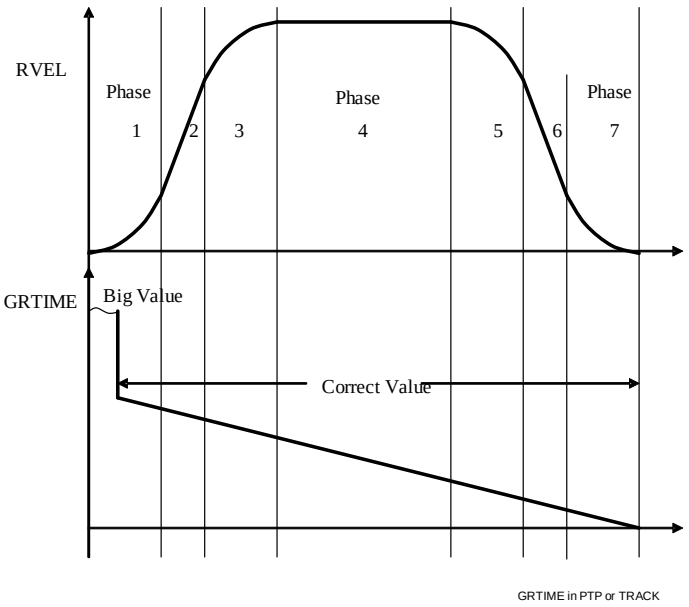


Figure 7 GRTIME Behavior in ptp or track Motion

4.2.4 Modulo Axis

Bit 29 (#**MODULO**) of the **MFLAGS** variable specifies modulo axis. If the bit is one, the axis feedback changes between the specified minimal and maximal positions as specified below:

- The ACSPL+ **SLPMIN** variable specifies the lower limit of modulo axis.
- The ACSPL+ **SLPMAX** variable specifies the upper limit of modulo axis.

The reference position **RPOS** of the modulo axis changes in the range from **SLPMIN** to **SLPMAX** inclusively.

Physically, the motion of the modulo axis is not limited, but each time when the **RPOS** comes out from range **SLPMIN..SLPMAX**, the controller brings **RPOS** into the range by changing the internal offset **EOFFS**. Note the following conditions:

- If the axis goes down and crosses the **SLPMIN** value, the controller adds value **SLPMAX**-**SLPMIN** to **EOFFS**. Assume the axis comes down to value **SLPMIN**-Δ. Correcting EOFFS, the controller brings **RPOS** to value **SLPMAX**-Δ.
- If the axis goes up and crosses **SLPMAX** value, the controller subtracts value **SLPMAX**-**SLPMIN** from **EOFFS**. Assume the axis comes up to value **SLPMAX**+Δ. Correcting **EOFFS**, the controller brings **RPOS** to value **SLPMIN**+Δ.
- Changing **EOFFS** immediately affects also feedback position **FPOS**. However, there is a slight difference between **FPOS** and **RPOS** behavior. **RPOS** always remains within the **SLPMIN**..**SLPMAX** interval. As **FPOS** differs from **RPOS** by position error, the corresponding **FPOS** may occur beyond the interval.

In the case of a default connection, the modulo operation also affects the **APOS** value (**APOS**=**RPOS**).

Note	
	**SLPMIN** *and* **SLPMAX** *variables can be changed only when the motor is disabled.*

4.3 Jog Motion

Jog motion is a motion with constant velocity and no defined end point. The motion continues until the next motion command stops it, or the motion fails because of limit switch activation or other condition.

Syntax:

jog[/switch] axis_designator [,direction] [,velocity]

Where **switch** can be one or a combination of:

w	Create the motion, but do not start until the **go** command has been issued.
v	Use the velocity specified in the command instead of the default velocity.

and **direction** is indicated by:

+	Motion is in the positive direction.
-	Motion is in the negative direction.

The simplest **jog** command is:

```
jog X
```

This command creates a jog motion of the X axis in positive direction using the default velocity **VEL(0)**.

Motion direction may be specified in the command:

```
jog X, -
```

This command creates a jog motion of the X axis in negative direction using the default velocity **VEL(0)**.

The command:

```
jog X, +
```

is the same as:

```
jog X
```

The **v** switch allows a specific velocity to be used instead of the default velocity **VEL**. The command:

```
jog/v X, 30000
```

ignores the default velocity and creates a jog motion with a velocity of 30000.

As with other types of motion, jog motion may be terminated by the **halt**, **kill**, or **break** commands. Unlike any other motion, jog motion also terminates when the next motion command for the same axis executes. For example, the following program fragment:

```
jog X, +
wait 500
jog X, -
```

provides jogging in the positive direction for 500msec and then switches to the negative direction. The controller automatically ensures a smooth transition from motion to motion.

Jogging can also occur in an axis group. For example, the following program fragment

```
jog XYT, -++
```

creates jogging in three axes: X in the negative direction, and Y and T in the positive direction. The motion uses the default velocity **VEL(0)** as a vector velocity for the three-axis motion.

4.4 Track Motion

Track motion enhances throughput by generating a move automatically if the target position is changed.

The **track** command initiates a track motion. In a track motion, a new move is generated to a new target position whenever the **TPOS** (target position) variable changes.

Syntax:

track[/w] axis_designator

The **w** switch causes the command to create the motion, but wait to start until a **go** command is issued.

The following command creates tracking motion of the X axis:

```
track X                                    Create track motion of X axis
```

If the axis is idle, the track motion is activated immediately. If the axis is moving, the controller creates the motion and inserts it into the axis motion queue. The motion waits in the queue until all previous motions in the queue are executed, and then starts.

When the track motion starts, the controller copies the current value of the reference position (**RPOS**) element to target position (**TPOS**) element. For example, when the command is executed, **RPOS(0)** is copied to **TPOS(0)**. No change of **RPOS** and no physical motion occur at this time.

Afterwards, the axis waits until the **TPOS** element is assigned a different new value. As soon as the program executes:

```
TPOS(0) = NewTarget          Assign a value to the TPOS element
```

the controller generates a PTP motion to the point designated by the value of the **NewTarget** user variable. After the X axis reaches **NewTarget**, the axis waits for the next change of **TPOS**. The next assignment to **TPOS(0)** automatically activates the next PTP motion and so on. Therefore, track motion is executed as a sequence of PTP motions.

The motion state bits **AST.#MOVE**, **AST.#ACC**, **MST.#MOVE**, and **MST.#ACC** reflect the state of each PTP motion in the track sequence exactly as they do for ptp motion. Between PTP motions, while the axis waits for the next **TPOS** assignment, the motion bits are zero (with the exception of the **MST.#MOVE** bit, which can be 1 if the position error exceeds the **TARGRAD** limit). The following ACSPL+ program fragment defines sequential positioning to points 1000, 2000, 10000, 11000:

```
track  Z                     Create track motion of Z axis
TPOS(2) = 1000               Move to point 1000
till ^AST(2).#MOVE           Wait till the motion ends
TPOS2 = 2000                 Move to point 2000
till ^AST(2).#MOVE           Wait till the motion ends
TPOS2 = 10000                Move to point 10000
till ^AST(2).#MOVE           Wait till the motion ends
TPOS2 = 11000                Move to point 11000
till ^AST(2).#MOVE           Wait till the motion ends
halt Z                       Terminate track motion
```

The result is similar to the following fragment:

```
ptp Z,1000                   Move to point 1000
ptp Z,2000                   Move to point 2000
ptp Z,10000                  Move to point 10000
ptp Z,11000                  Move to point 11000
```

While the code with the **ptp** commands looks shorter and simpler, there are applications where track motion is preferable to PTP motion.

Track motion is not terminated automatically. If **TPOS** is not changed, the axis track motion remains at the last target point until **TPOS** is assigned a new value, and then the motion continues.

Use a **halt** or **kill** command to terminate track motion.

Any subsequent motion command (except another **track** command) for the same axis also terminates the tracking motion. In this respect track motion is similar to the jog motion.

The motion profile while in Track mode, like in a standard PTP motion, is defined by the ACSPL+ variables **VEL**, **ACC**, **DEC** and **JERK**. The track command accepts the values of

these variables at the beginning of each component PTP motion within the track motion. Therefore, if an application assigns a new value to **VEL**, **ACC**, **DEC** or **JERK**, while track mode is in effect, then the new value will be used the next time that the application initiates a motion (by assigning a new value to **TPOS**).

The following ACSPL+ program fragment sets a specific velocity for each PTP motion:

`track  Y`	Create track motion of Y axis
`VEL(1) = 20000`	Set motion velocity 20000 units/sec
`TPOS(1) = 1000`	Move to point 1000
`till ^AST(1).#MOVE`	Wait till the motion ends
`VEL1 = 5000`	Set motion velocity 5000 units/sec
`TPOS1 = 2000`	Move to point 2000
`till ^AST(1).#MOVE`	Wait till the motion ends
`VEL1 = 10000`	Set motion velocity 10000 units/sec
`TPOS1 = 110000`	Move to point 11000
`till ^AST(1).#MOVE`	Wait till the motion ends
`halt Y`	Terminate tracking motion

In the example above the application updates **TPOS** only after the previous PTP motion ends.

In the following example the application updates **TPOS** while the motion is still in progress:

`track  X`	Create tracking motion of X axis
`TPOS0 = 2000`	Move to point 2000
`till GPHASE(0) >= 6`	Wait till the motion comes to phase 6 (deceleration to final point)
`TPOS0 = 2500`	Correct the final point
`till ^AST(0).#MOVE`	Wait till the motion ends
`halt X`	Terminate tracking motion

In this case, the controller does not execute two separate motions. As soon as **TPOS** is changed to 2500 (before the controller reaches 2000), the controller changes the move on-the-fly to the new target position of 2500. The on-the-fly change is done smoothly, similar to end-point correction on-the-fly.

The same result is provided by the following fragment:

`ptp X, 2000`	Move to point 2000
`till GPHASE0 >= 6`	Wait till the motion comes to phase 6 (deceleration to final point)
`break X`	Terminate the current motion and provide smooth transition to the next motion
`ptp X,2500`	Move to point 2500

The **track** command may also be used for programming multi-axes motion, for example, the command

```
track XZT
```

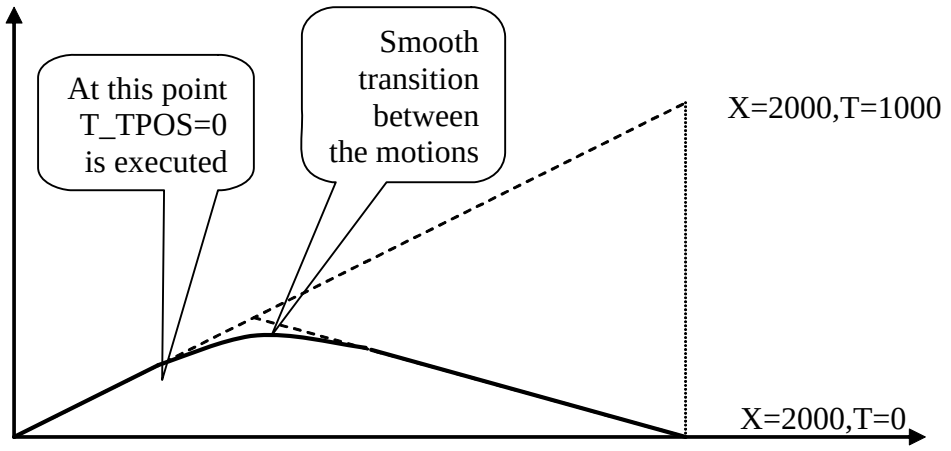
creates track motion of X, Z and T axes. The multi-axis track motion is executed as a sequence of PTP motions. A new PTP motion starts each time when one or more elements of **TPOS** that correspond to the involved axes are changed. Consider the following example:

`track X; track Z; track T`	Create track motion of X, Z, T axes
`TPOS(0)=0; TPOS(2)=0; TPOS(3)=0`	Move to point X=0, Z=0, T=0
`till ^AST(0).#MOVE`	Wait till the motion ends
`TPOS(2)=1000`	Move to point X=0, Z=1000, T=0
`till ^AST(0).#MOVE`	Wait till the X motion ends
`TPOS(0)=100;TPOS(3)=200`	Move to point X=100, Z=1000, T=200
`halt XZT`	Terminate track motion

In the following example **TPOS** is updated while the previous motion is still in progress:

`track XT`	Create track motion of XT axis
`TPOS(0)=2000; TPOS(2)=1000`	Move to point X=2000, T=1000
`till GPHASE(0) = 4`	Wait till the motion reaches constant velocity (phase 4)
`TPOS(3)=0`	Set a new final point X=2000, T=0
`till ^AST(0).#MOVE`	Wait till the motion ends
`halt X`	Terminate tracking motion

In the above case, the controller does not execute two separate motions. As soon as **TPOS** is updated, the controller changes on-the-fly from PTP motion towards X=2000, T=1000 to PTP motion towards X=2000, T=1000. The transition from the first motion to the second is done smoothly. While each PTP motion follows a straight trajectory, the transition between the motion is not straight, as shown in the following diagram:



4.5 Segmented Motion

Segmented motion moves axes along a continuous path. The path is defined as a sequence of linear and arc segments on the plane. Although segmented motion follows a flat path, it may involve any number of axes because the motion plane can be connected to the axes at any projection transformation.

4.5.1 Understanding Slaved Segmented Motion

Motion generation for segmented motion may be considered as a two-stage process. In the first stage the controller generates a smooth motion diagram as a function of time:

S = F(T)

where **S** is a distance along the segmented path, **T** stands for time, and **F** is a function independent of the specified segments.

In the second stage the controller separates the S path into the involved axes:

X = F_X(S)

Y = F_Y(S)

The second stage supplies the current values of the involved axes. The functions **F_X**, **F_Y** depend only on the specified segments. Only the second stage builds the shape of the path in the XY plane. The first stage provides the motion progress along the path. If the function F of the first stage is modified, this affects the motion velocity and time, but does not alter the final shape of the path.

The including the **s** or **p** switch with the **mseg** command affects the first stage of the motion generation process by causing the distance **S** to follow the value **MPOS** (Axis Master) of the leading axis in the group. For position lock (**p** switch), following is strict:

```
S = MPOS
```

For velocity lock (**s** switch), following allows a constant offset:

```
S = MPOS + C
```

In both cases the second stage of the motion generation remains unchanged and depends only on the specified segment sequence.

Segment commands specify a path on the plane, and the **MPOS** value of the leading axis defines motion progress along the path.

Formulas that calculate the **MPOS** value must be defined before using the master command .

4.5.2 mseg, projection, line, arc1, arc2, stopper Commands

Syntax:

mseg[/switch] axis_group, initial_start_point [,initial_start_point, initial_start_point]

[,projection matrix_designator]

line[/switch] axis_group, final_point [,final_point, final_point]

arc1[/switch] axis_group, center_point, final_point, rotation_direction [,velocity]

arc2[/switch] axis_group, center_point, rotation_angle, rotation_direction [,velocity]

stopper axis_group

ends axis_group

Where **switch** can be one or a combination of:

w	Create the motion, but do not start until the **go** command
v	Use the velocity specified for each segment instead of the default velocity
c	Use the segment sequence as a cyclic array: after the last segment return to the first segment and so on.
s	Slaved motion - the motion advances in accordance to the master value of the leading axis (velocity lock).
p	Position lock - slaved motion, strictly conforming to the master value.
e	Extrapolated - if a master value travels beyond the specified path, the last or the first segment is extrapolated.
t	Stalled - if a master value travels beyond the specified path, the motion stalls at the last or first point.

The **e** and **t** switches are relevant only for slaved motion and must be used with **s** or **p** suffix. For discussion of slaved motion see **Section 4.6.2 - slave Command**.

Segmented motion can be executed in an axis group with any number of controller axes.

The **mseg** command specifies axis group and the initial starting point:

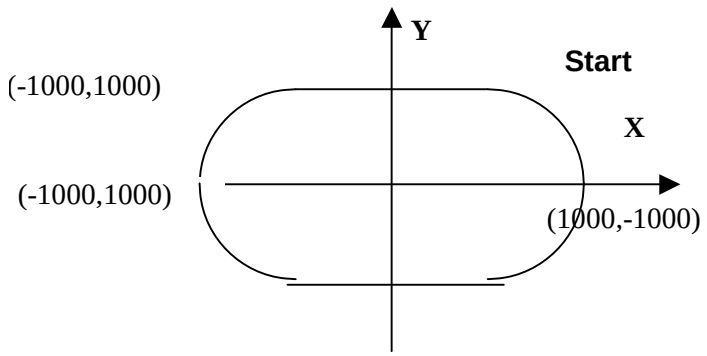
```
mseg XY, 1000, 1000
```

This command creates a segmented motion of the X axis group and specifies the coordinates of initial point on the plane. The **mseg** command itself does not specify any segment, so the created motion does not start immediately. A **line** or **arc** command must follow the **mseg** command to specify the segment sequence.

Consider the following program fragment:

`mseg XY,1000,1000`	Create segmented motion in group X, coordinates of the initial point are (1000,1000).
`arc1 XY, 1000,0,1000,-1000,-`	Add arc segment with center (1000,0), final point (1000,-1000), clockwise rotation.
`line XY,-1000,-1000`	Add line segment with final point (-1000,-1000).
`arc2 XY,-1000,0,-3.141529`	Add arc segment with center (-1000,0) and rotation angle -π radians.
`line XY,1000,1000,50000`	Add line segment with final point (1000,1000).
`ends XY`	End the segment sequence

The mseg command creates the segmented motion. The motion does not start, because no segment is defined yet. After the first arc1 command the motion starts if the axis group is idle (not involved in some previous motion). If the group is not idle, motion will start when the previous motion stops. The four segment commands specify the following path:



arc1 and **arc2** differ only by the required arguments. **arc1** requires the coordinates of the center point, final point, and the direction of rotation. **arc2** requires the coordinates of the center point and the rotation angle (in radians). Each command produces the same result, so selection of either **arc1** or **arc2** depends on the available data. If you know the coordinates of the center point, coordinates of the final point and the direction of rotation, **arc1** is preferable. If you know the coordinates of the center point and rotation angle, **arc2** is preferable.

The **rotation_direction** argument can be:

- **+** (plus) – for counter clockwise
- **-** (minus) – for clockwise rotation

The entire sequence of segmented motion must be terminated with an **ends** command. The **ends** command informs the controller that no more segments will be specified for the specified motion. The motion cannot finish until the **ends** command executes. If the **ends** command is

omitted, the motion will stop in the last point of the sequence and wait for the next point. No transition to the next motion in the queue will occur until the **ends** command is executed.

Segmented motion usually starts with the first segment command (**line**, **arc1**, or **arc2**). The next segment command therefore executes while the motion is already in progress. This is generally not a problem, because the program execution rate is higher than typical motions time. However, sometimes you may need to delay starting the motion until all points are defined. In this case you append the **w** switch to the command to prevent the motion from starting until the **go** command is issued. The motion, created by the command

```
mseg/w XY,1000,1000
```

will not start until the **go XY** command is issued.

The **r** switch is not allowed with segmented motion. All coordinates in the line, **arc1** and **arc2** commands are absolute in the plane. However, the whole path is relative to the point where the axes are located when the segmented motion starts. In other words, all coordinates are absolute in the plane, but the plane is relative to the starting point. Note that the initial point, specified in the **mseg** command, is also absolute in the plane. Therefore, the initial point does not cause any motion to that point, but only supplies starting coordinates for the first segment. The motion program should provide a motion to the desired initial point before executing the **mseg** command. The following fragment illustrates a typical activation of the segmented motion:

`ptp XY, 0, 100`	This command causes a physical motion to the point (0,100) that will be an absolute starting point for the following segmented motion.
`mseg XY, 100, 100`	Defines the starting coordinates for the first segment, which are absolute in the plane. The whole plane is located in such a way that the starting point in the plane (100,100) coincide with the present position of the motors (0,100).

The **mseg** command uses the default velocity **VEL** for each segment. The **v** switch overrides the default velocity and allows you to define a specific velocity for each segment. The desired velocity must be specified in the **line**, **arc1** and **arc2** segment commands after all other arguments. If the velocity argument is omitted in a segment command, the velocity from the previous segment is used. If the velocity argument is omitted in the first segment, the default **VEL** is used. The previous example is modified for using individual velocities:

`mseg/v XY,1000,1000`	Create segmented motion in group X, coordinates of the initial point are (1000,1000).
`arc1 XY, 1000,0,1000,-1000,-,30000`	Add arc segment with center (1000,0), final point (1000,-1000), clockwise rotation, vector velocity 30000.
`line XY,-1000,-1000`	Add line segment with final point (-1000,-1000). Vector velocity is not specified, previous value 30000 will be used.
`arc2 XY,-1000,0,-3.141529,10000`	Add arc segment with center (-1000,0), rotation angle of -π, vector velocity 10000.

`line XY,1000,1000,5000`	Add line segment with final point (1000,1000), vector velocity 50000.
`ends XY`	End the segment sequence

Several suffixes can be attached to one command. For example, the command

```
mseg/vw XY, 1000, 1000
```

creates a segmented motion with individual velocity for each segment. The motion does not start until the **go XY** command is issued.

4.5.3 projection Command

The **projection** command is an expansion to the **mseg …ends** set of commands that allows the controller to perform a three dimensional segmented motion such as creating arcs and lines on a user-defined plane. The method for this 3D segmented motion is to set a transformation matrix that defines a new plane for the segmented motion.

Syntax:

projection axes, matrix_table

Where:

axes	List of axes.
matrix_table	Coordinates defining the new plane.

As mentioned above, all coordinates, specified in the segment commands, are absolute in the working plane. Projection is a matrix that connects the plane coordinates and the axis values as specified in the mseg command. If the axis group contains two axes, and no **projection** command is specified, the controller provides a default projection that corresponds to a 2x2 matrix:

1	**0**
0	**1**

The matrix directly connects the first coordinate of the working plane to the first axis of the axis group and the second coordinate to the second axis.

The matrix can also define rotation and scaling. The full transform also includes an implicit offset. The controller calculates the offset automatically in such a way that the initial coordinates specified in the mseg command match the desired axis values at the moment when the motion starts. The offset provides the full path to be relative to the starting point.

If an axis group contains N axes, the controller extends the default matrix to N lines. The additional lines are filled by zeros:

1	**0**
0	**1**
0	**0**
…	**…**
0	**0**

The matrix connects only the first two axes to the plane coordinates. Therefore the segmented motion will involve only the first two axes in the group.

If N = 1, the **mseg** command applies to a single axis, and the matrix contains the first line only:

1	**0**

In this case the axis will follow the first coordinate of the working plane.

You can replace the default matrix with the **projection** command.

Example:

```
real M(3)(2)                                    !Define Matrix

M(0)(0)=1;M(0)(1)=0
M(1)(0)=0;M(1)(1)=1
M(2)(0)=0;M(2)(1)=2.74                          !Set the transformation
                                                !matrix values.
VEL(0)=1000;ACC(0)=10000;DEC(0)=10000           !Axis motion parameters
enable XAB                                      !Required command.
group XAB                                       !Required command.
set FPOS(0)=0;set FPOS(4)=0;set FPOS(5)=0       !Set axes' FPOS=0
mseg XA,0,0                                     !Define original plane.
projection XAB,M                                !PROJECTION of axes XAB by
                                                !matrix M
arc2 XA,750,0,6,24                              !ARC2 performed on new plane.
ends XA                                         !Concludes MSEG.
stop                                            !End Program
```

If the group contains N axis, the matrix in the projection command must be of size Nx2.

4.5.4 Arguments as Expression

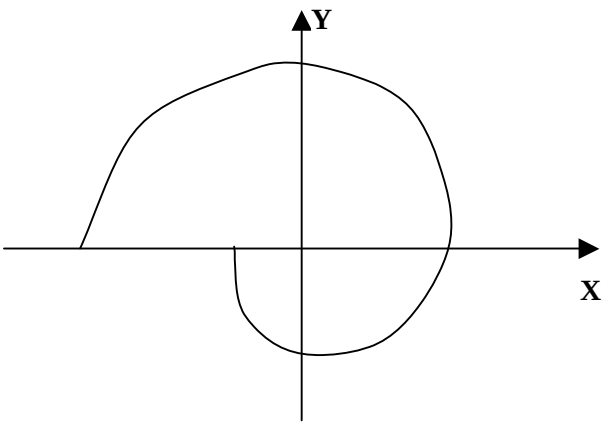
Any argument in the **line**, **arc1**, **arc2** commands can be specified by expression.

Using an expression instead of axis specification allows the involved axes to be calculated in the execution time as opposed to the programming time (axis-independent programming). The calculation must result in an integer between 0 and 7, corresponding to the eight axes. Value 0 corresponds to the X axis, 1 to Y, and so on.

Expression in place of coordinate allows calculating the segments on the fly. Consider the following program fragment:

`real P, K, S`	Declare the real variables P, K and S.
`P = 3.14159; K = 100 / P; S = P / 1000`	Calculate P K and S.
`ptp XY, -100, 0`	Perform a point-to-point motion in the XY plane to the point: (-100,0).
`mseg XY, -100, 0`	Create a segmented motion.
`loop 2000`	Perform the loop 2000 times.
`P = P + S`	
`line XY, P*K*cos(P), P*K*sin(P)`	Execute a linear segment based on a calculation.
`end`	End loop.
`ends XY`	End segment motion.

The program executes the line command 2000 times. The line segments build up a curve close to the Archimedean spiral:



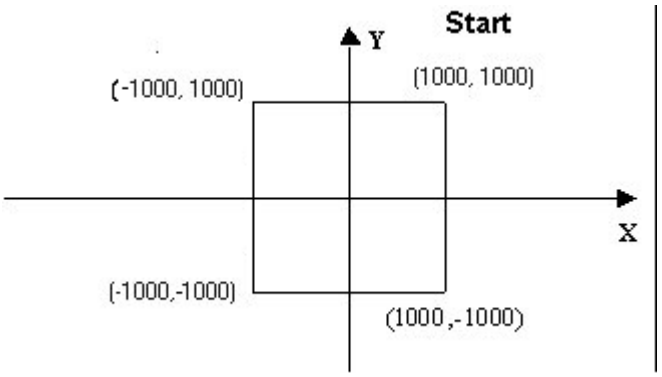
4.5.5 stopper Command

The controller builds the motion so that the vector velocity follows a smooth velocity diagram. If all segments are connected smoothly, axis velocity is also smooth. However, if you have defined a path with an inflection point, axis velocity jumps at this point. The jump might cause a motion failure due to the acceleration limit. Even if a failure does not occur, the abrupt change in velocity impairs accuracy and may be harmful to the machine.

The **stopper** command is used to avoid velocity jump in the inflection points. If a **stopper** command is specified between two segments, the controller provides smooth deceleration to zero before the stopper and smooth acceleration to specified velocity after the stopper. Consider the following program fragment:

`ptp XY, 1000, 1000`	Go to initial point.
`mseg XY, 1000, 1000`	Create a segmented motion.
`line XY, 1000, -1000`	Execute linear segment.
`stopper XY`	Slow down to zero.
`line XY, -1000, -1000`	Execute linear segment.
`stopper XY`	Slow down to zero.
`line XY, -1000, 1000`	Execute linear segment.
`stopper XY`	Slow down to zero.
`line XY, 1000, 1000`	Execute linear segment.
`ends XY`	End the segment sequence.

The program provides a rectangular path without velocity jumps:



4.5.6 Cyclic Motion

The **c** switch provides cyclic execution of segmented motion. The **mseg/c** command creates a motion that after executing the last segment, the motion begins again continues with the same sequence. The final point of the last segment is therefore the starting point of the first segment.

Note	
	*Cyclic segmented motion does not automatically finish. You must use one of the commands* **halt**, **kill**, **break** *to stop cyclic motion.*

The following is an example of the coding for cyclic motion:

```
enable XY
mseg/C XY,1000,1000    ! Create segmented motion in group X, coordinates of
                       ! the initial point are (1000,1000)

arc1 XY, 1000,0,1000,–1000,–   ! Add arc segment with center (1000,0), final
                               ! point (1000,-1000), clockwise rotation

line XY,–1000,–1000    ! Add line segment with final point (-1000,- 1000)

arc2 XY,–1000,0,–3.141529     ! Add arc segment with center (-1000,0) and
                              ! a rotation angle of -π

line XY,1000,1000     ! Add line segment with final point (1000,1000)

ends XY     ! End the segment sequence
```

4.5.7 Slaved Motion at Extreme Points

Additional switches define behavior of slaved motion at extreme points, when the motion approaches the final point of the last segment or the starting point of the first segment. This occurs, when the **MPOS** master value falls out of the interval (0, L), where L is the overall length of the path. There are four possibilities, depending on the switch employed in the command:

- **c** – Cyclic motion

  The path is closed. The motion passes from the last segment to the first or from the first to the last as if they were adjacent.

- **e** – Extrapolated motion

  If the distance S is slaved to the **MPOS** master value becomes greater than the overall length of the path, the motion continues along the extrapolated last segment. If the distance S slaved to the **MPOS** master value becomes less than zero, the motion continues along the extrapolated first segment. If the extrapolated segment is a circular arc, the motion will follow along the extrapolated circle.

- **t** – Stalled motion

  When the motion approaches the extreme point, the slave comes out from synchronism and stalls in the point until the **MPOS** master value allows to regain synchronism. For velocity lock synchronism is achieved when the **MPOS** changes its direction; after regaining the offset **C** may have a different value than before approaching the extreme point. For position lock synchronism regains when the **MPOS** falls into the interval (0, L) again. The controller ensures smooth approaching the extreme points and smooth return to synchronism.

- No swtich – Bounded motion

  The motion finishes when the slave approaches any extreme point. The controller activates the next motion in the queue (if any). If the **MPOS** master value changes only in positive direction, the behavior is very close to non-slaved motion. The difference is that non-slaved

motion is based upon the time value and slaved motion is based upon the **MPOS** master value.

4.6 Master/Slave Motion

4.6.1 master Command

The **master** command defines a formula for calculating the axis master value (**MPOS** – see *SPiiPlus Command & Variable Reference Guide*).

Syntax:

master MPOS(axis_index)=value

Only one component of the **MPOS** (master position) variable is allowed as **MPOS(axis_index)**.

In the simplest case the master value follows the feedback of another axis:

```
master MPOS(0) = FPOS(1)
```

When the command executes, the controller stores the formula specified to the right of the equals sign, and then calculates the master value **MPOS(0)** according to this formula (in the example above, it simply assigns the current **FPOS(1)** value to **MPOS(0)**). The controller does this calculation every controller period, independent of program execution. Even if the program that executed the **master** command terminates, the controller continues calculating the last specified master expression, until the another master command for the same axis executes.

The master value can be redefined at any time by the application. If the program that executed the above command then executes the command.

A more sophisticated use of the **master** command connects the axis to another axis feedback with a scale factor:

```
master MPOS(0) = 2.3 * FPOS(1)
```

The following example defines axis Z to follow the **RPOS** (reference position) of axis X translated through a conversion table (cam motion):

```
master MPOS(2) = mapby1(RPOS(0), Table)
```

In this example **Table** is a user-defined array that contains a table for conversion.

Similarly, the **master** value may be connected to other sources such as the sum of two or more axes, or to an analog input.

4.6.2 slave Command

The **slave** command creates a motion slaved to the master value of the specified axis.

Syntax:

slave[/switch] axis_designation [,start_interval, end_interval]

Where **switch** can be:

w	Create the motion, but do not start until the **go** command has been issued.
p	Use position lock instead of velocity lock (see **Section 4.6.2.2 - Velocity Lock vs. Position Lock**).
t	Stall when approaching interval boundary (see **Section 4.6.2.3 - Stalled Motion**).

Slave motion is governed by the variables of the slaved axis. These include:

XSACC	Maximal allowed acceleration of the synchronous motion. If the master acceleration exceeds this value, the slave comes out from synchronism.
SYNV	Allowed difference in master and slave velocities. Used in asynchronous motion to determine if the synchronism can be re-established.
JERK	Default jerk. The slave uses this variable only in asynchronous motion to overtake the master.
ACC	Default acceleration. The slave uses this variable only in asynchronous motion to overtake the master.
VEL	Default velocity. The slave uses this variable only in asynchronous motion to overtake the master.

Once started, slaved motion terminates only if a failure occurs, or one of the commands **halt**, **kill**, or **break** is executed. The **halt** and **kill** commands provide deceleration to zero and then the next motion starts. If no next motion was created, the axis becomes idle. The **break** command provides smooth transition to the next motion without stopping, if a next motion is waiting in the queue.

4.6.2.1 Synchronization

In slaved motion the slave is usually synchronized to the master, meaning that the **APOS** axis reference follows the **MPOS** master value strictly or with a constant offset. However, there are two cases when synchronism is not attainable:

- The slaved motion starts, and positions (position lock) or velocities (velocity lock) of the master and slave differ. The motion starts as asynchronous.
- The motion was synchronized, but the acceleration of the master exceeds the allowed limit (the **XSACC** variable of the axis) for the slave. The slave comes out of synchronization.

In both cases, the motion continues asynchronously, and the correspondence between **APOS** and **MPOS** appears broken. The controller tries to regain synchronization by having the slave pursue the master within the maximal allowed motion parameters. When the slave overtakes the master, synchronization is re-established and the motion continues as synchronous.

Only individual axes are allowed to be used in a **slave** command. Groups are not allowed. The created motion starts immediately if the axis is idle; otherwise the motion waits in the motion queue until all motions created before for the axis finish. The following command creates a slaved motion of the X axis:

```
slave X
```

Note	
	*The slave axis in a master-slave motion may show its state (through the* **AST** *variable) as accelerating and in motion even when the master axis is motionless. This reflects the fact that the axis is set to follow the motion of the other axis and is not following a motion profile of its own.*

4.6.2.2 Velocity Lock vs. Position Lock

In velocity lock the slave velocity follows the master velocity. A constant offset between the master and slave position is allowed. In position lock the slave position strictly follows the master position.

The slave command without the **p** switch activates a velocity-lock mode of slaved motion. When synchronized, the **APOS** axis reference follows the MPOS with a constant offset:

```
APOS(0) = MPOS(1) + C
```

Where **C** is constant in velocity lock mode and is zero in position lock mode.

When the **mseg** command includes the **p** switch (see **Section 4.5.2 - mseg, projection, line, arc1, arc2, stopper Commands**), this activates the position lock mode of slaved motion. When synchronized, the **APOS** axis reference follows the **MPOS** strictly:

```
APOS(0) = MPOS(1)
```

When the motion is asynchronous for any reason (see above), the controller tries to regain synchronism by having the slave pursue the master with the maximal allowed motion parameters. The difference between position lock and velocity lock manifests itself at the moment of regaining synchronization:

- Velocity lock motion switches to synchronized when the slave velocity reaches the master velocity (with allowed error defined by the **SYNV** variable of the slaved axis). At this moment the difference between the master position and the slave position is latched as the constant offset C, which then remains unchanged as long as the motion is synchronous.
- Position lock motion switches to synchronized when the slave position overtakes the master position, i.e., when **APOS** = **MPOS**.

Note that each time the motion loses and regains synchronization, the velocity lock offset C may latch a different value. Under the same conditions, the position lock motion each time re-establishes the strict equality **APOS** = **MPOS**.

4.6.2.3 Stalled Motion

When the **slave** command does not include the **t** switch, the command applies no limits to the slaved axis. The axis follows the master everywhere, unless a failure, such as limit switch activation, occurs.

The **slave** command with the **t** switch requires two additional parameters that define a permitted interval for the slaved axis motion. For example, the command:

```
slave/t X, -1000, 2000
```

allows X axis motion only within the interval (-1000, 2000).

When the **APOS** axis reference approaches either of the two interval limit points, the slave comes out from synchronism and stalls at that point until the **MPOS** master value allows restoration of synchronism. In velocity lock, synchronization is regained when the **MPOS** changes its direction. After regaining the offset, **C** may have a different value than before approaching the extreme point. For position lock, synchronization is restored when the **MPOS** comes back into the permitted interval. The axis stalls when the master leaves the permitted range and regains synchronization when the master returns into the permitted range. The controller ensures a smooth approach to the extreme points and a smooth return to synchronization.

4.7 Arbitrary Motion

The **path** command is similar to **mptp** in that it generates multi-point motion, but in this case it creates an arbitrary path motion.

Syntax:

path[/switch] axis_designators [,time_interval]

point axis_designators, coordinate [,coordinate] [,velocity] [,time_interval]

mpoint axis_designators, point_matrix, number_of_points [,time_interval]

ends

Where **switch** can be one or a combination of:

r	The point coordinates are relative to the previous point.
w	Create the motion, but do not start until the **go** command.
c	Use the point sequence as a cyclic array, that is, after positioning to the last point return to the first point and repeat.
t	Non-uniform time interval; the time interval is specified for each point along with the point coordinates.

Points for arbitrary path motion are defined by **point** and **mpoint** commands (see **Section 4.2.2 - mptp, point, mpoint, and ends Commands**).

The **ends** command terminates the point sequence. After the **ends** command, no **point** or **mpoint** commands for this motion are allowed.

The trajectory of the motion follows through the defined points. Each point presents the instant desired position at a specific moment. Time intervals between the points are uniform, or non-uniform as defined by the **t** switch.

Motion generated by the **path** command does not use the standard motion profile. Typically, the time intervals between the points are short, so that the array of the points implicitly specifies the desired velocity in each point. For this reason, variables **VEL**, **ACC**, **DEC**, **JERK** have no affect on this motion.

If the time interval does not coincide with the controller cycle, the controller provides linear interpolation of the points.

Commands **halt**, **kill**, **killall** (see **Section 4.1.8 - halt Command** and **Section 4.1.3 - kill and killall Commands**) are executed in a specific way with this type of motion; as with other motion types, the controller provides a smooth deceleration profile using **DEC** (**halt** command) or **KDEC** (**kill**, **killall** commands) for the leading axis. However, unlike other motions types, the controller does not provide following the motion trajectory during deceleration.

Arbitrary path motion created without the **t** switch implies uniform intervals between the points. If **path** is not specified with the **t** switch, the **time_interval** argument has to be included. The argument defines time interval between the motion points in milliseconds.

If **path** is specified with the **t** switch, it must not have **time_interval** specification. Instead, the **time_interval** must be specified for each point as an additional argument for the **point** command or as additional array column in the **mpoint** command.

Note	*The* **break** *command (see* ***Section 4.1.9 - break Command****) is not supported when using the* **path** *command.*

4.8 Spline Motion

4.8.1 Spline Motion Theory

General theory of spline interpolation is a topic of numerous books and articles. A classical introduction is *A Practical Guide to Splines* by Carl De Boor.

This section contains only basic facts required for understanding spline implementation in the SpiiPlus controller.

4.8.1.1 Main Definitions

PV (position-velocity) and PVT (position-velocity-time) refer to a motion mode that constructs the motion trajectory from spline segments. You specify the end position (P) and the end velocity (V) for each segment of motion. The difference between PV and PVT is that PVT motion also requires specification of the time interval for each segment, whereas PV motion uses a predefined constant time interval.

In PV/PVT mode the controller provides cubic spline interpolation between the specified points. As a spline mode, it minimizes the amount of data that the host-based program (or ACSPL+ program) needs to generate to produce the multi-axis arbitrary profile and provides precise trajectory generation.

A spline is a special function defined piecewise by polynomials. The spline is a piecewise polynomial function spread over an interval [a,b] consists of polynomial pieces, such that:

$a = t_0 < t_1 < t_2 < ... < t_{k-2} < t_{k-1} = b$

The points: t_i are called **knots**. The vector is called a **knot vector** for the spline. If the knots are equidistantly distributed in the interval [*a*,*b*], we say the spline is **uniform**, otherwise we say it is **non-uniform**.

In many cases functional dependence between two or more values cannot be expressed as an analytic formula. The most common presentation of those functions is a table of function values in specific points.

For example, a machine axis was graduated with an external laser interferometer. The result of graduation is a table like the following:

Commanded position (x)	100	200	300	400	500	600	700	…
Actual position (p)	103	199	294	402	500	598	705	…

The table defines a functional dependence *p=f(x)* that cannot be expressed analytically.

The argument values for *x* in the definition table are knots, and the function values for *p* are control points.

A 3rd order polynomial spline provides an approximation of the table-driven function that can provide the function value not only in the knots, but at any point. Between each two knots the spline is expressed as:

$$p = a_0 + a_1 x + a_2 x^2 + a_3 x^3 = \sum_{i=0}^{3} a_i x^i$$

where coefficients a_0, a_1, a_2, a_3 have different values at different intervals.

The SPiiPlus controller also supports two-dimensional splines. In this case, the definition table is a two-dimensional matrix. Knot points are defined for two arguments *x* and *y*, and the matrix contains corresponding *p* values. Knot values divide the XY plane into rectangular cells. The matrix defines the function values in the cell vertices. Within each cell, the interpolating spline is expressed as:

$$p = \sum_{i,j=0}^{3} a_{ij} x^i y^j$$

Many different spline approximations can be provided for one definition table. The SPiiPlus controller supports two kinds of splines: Catmull-Rom and B-Splines (see description below).

If the distance between the knots in the table is constant, the spline is called uniform. On the contrary, a non-uniform spline corresponds to a table that contains function values in arbitrary points. However, the definition table always arranges the knot values in ascending order, so that $x_i \leq x_{i+1}$.

All knot points constitute the *definition range* of the spline. **Figure 8** illustrates definition range of a function defined with six non-uniform knots:

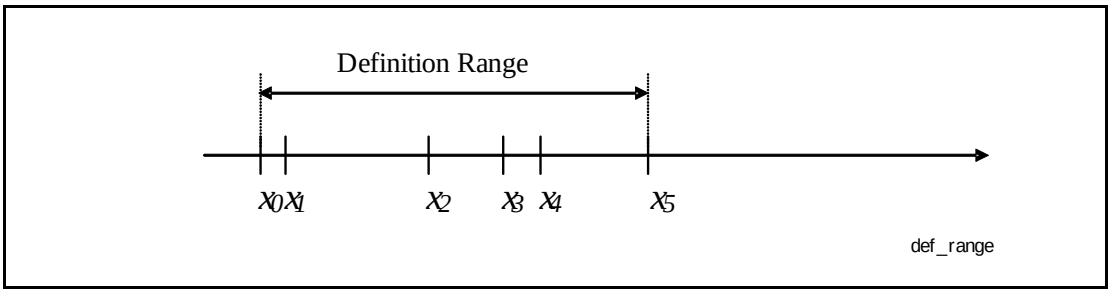


Figure 8 Spline Definition Range

In a two-dimensional case, definition range is a rectangular area, as illustrated in **Figure 9**:

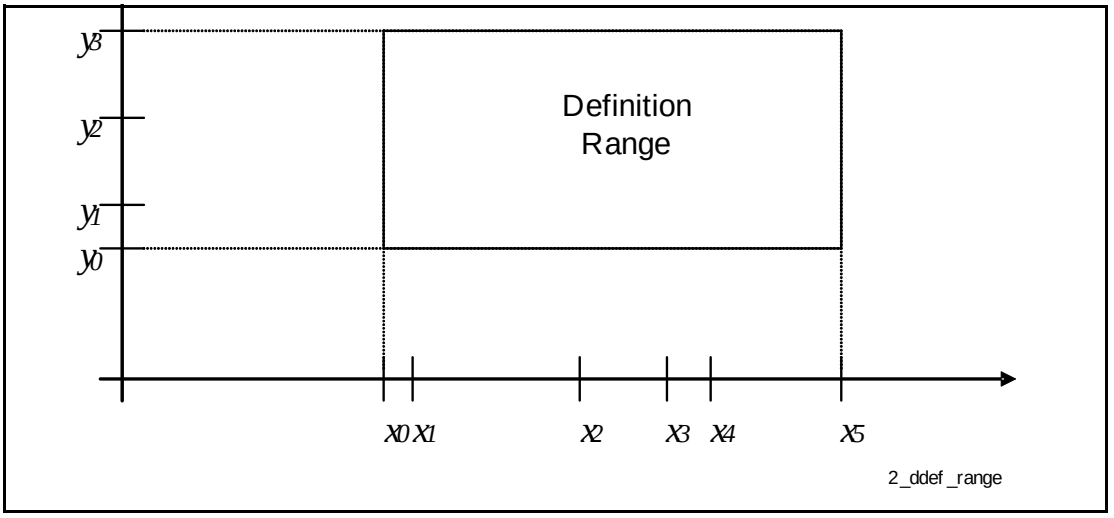


Figure 9 Two-Dimensional Spline Definition Range

4.8.1.2 One-Dimensional Catmull-Rom Spline

Assume a definition table provides N control points p_0, p_1, p_2… p_{N-1} in knots x_0, x_1, x_2… x_{N-1}.

The Catmull-Rom construction process is described as follows:

- At each internal knot x_i ($1 \le i \le N-2$) calculate the derivative:
  $v_i = (p_{i+1} - p_{i-1})/(x_{i+1} - x_{i-1})$
- At the first and last knots assume zero derivative: $v_0 = v_{N-1} = 0$.
- In interval i ($1 \le i \le N-2$), build a 3rd order polynomial $p=f(x)$ that satisfies four bound conditions p_i, v_i, p_{i+1}, v_{i+1}.
- Beyond the definition range the spline is defined as follows:
  $p = p_0$ if $x < x_0$
  $p = p_{N-1}$ if $x > x_{N-1}$

Figure 10 illustrates a Catmull-Rom spline that interpolates a 5-component definition table.

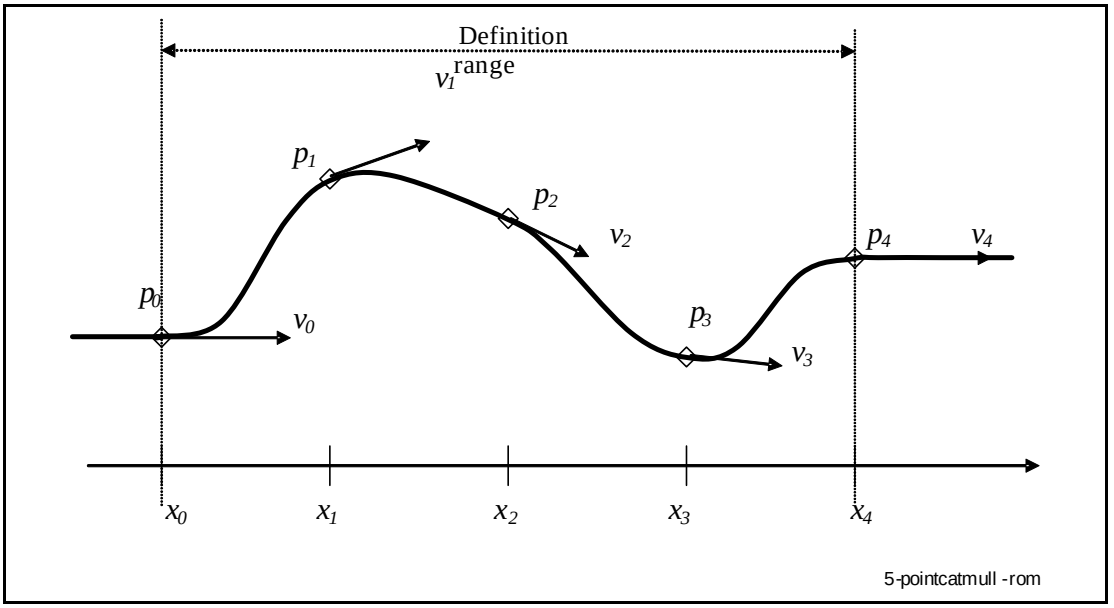


Figure 10 5-Point Catmull-Rom Spline

Features of the Catmull-Rom spline:

- The spline is $\mathbf{C^1}$-continuous, meaning the curve and its first derivative are continuous functions. The second derivative has discontinuities in the knot points.
- The curve interpolates all control points; meaning that the curve goes through each control point.
- At internal control point number *i*, the first derivative vector is parallel to the line connecting control points *i-1* and *i+1*.
- At the first and the last control points, the first derivative is zero.
- The spline yields a constant value equal to p_0 on the interval from $-\infty$ to x_0.
- The spline yields constant value equal to p_{N-1} on the interval from x_{N-1} to $+\infty$.

4.8.1.3 One-Dimensional B-spline

Assume a definition table provides N control points p_0, p_1, p_2… p_{N-1} in knots x_0, x_1, x_2… x_{N-1}.

Unlike the Catmull-Rom spline, a B-Spline does not go through the control points. Actually, it approximates the control points as illustrated in **Figure 11**.

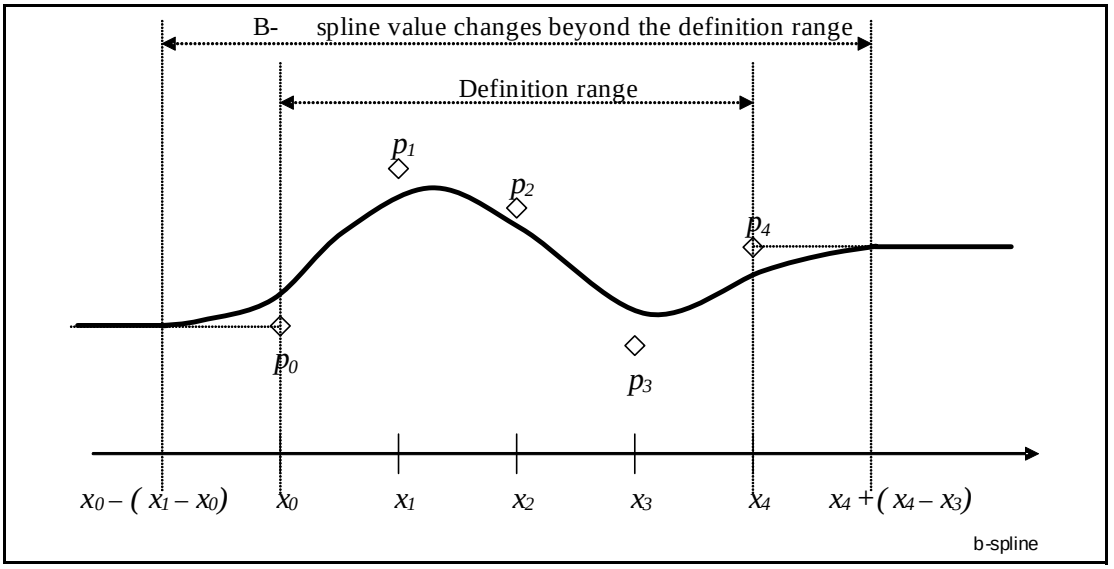


Figure 11 B-Spline - Approximation of Points

Compared to a Catmull-Rom spline, a B-Spline generates a smoother curve. Used in the controller, a B-spline provides continuous velocity and acceleration; where Catmull-Rom spline provides continuous velocity only, and acceleration may change by jumping at the control points.

Features of the B-Spline:

- The spline is $\mathbf{C^2}$-continuous, meaning the curve, its first and second derivatives are all continuous functions.
- The curve approximates the control points; and does not go through each control point.
- The spline yields changing value in the interval from $x_0 – (x_1 – x_0)$ to $x_{N-1}+(x_{N-1} – x_{N-2})$.
- The spline yields constant value equal to p_0 in the interval from $-\infty$ to $x_0 – (x_1 – x_0)$.
- The spline yields constant value equal to p_{N-1} in the interval from $x_{N-1}+ (x_{N-1} – x_{N-2})$ to $+\infty$.

Not-passing the control points is not always a drawback. If values p_0, p_1, p_2… p_{N-1} are obtained from some measuring process, the values include measuring error that has a stochastic component. B-Spline tends to filter out the stochastic error, thereby improving overall accuracy.

4.8.1.4 Two-Dimensional Splines

The controller supports two-dimensional Catmull-Rom and B-Splines.

A two-dimensional spline approximates a definition table that provides NxM control points p_{00}, p_{01}, …$p_{0,M-1}$, p_{10}, p_{11},… $p_{N-1,M-1}$ on the grid defined by knots x_0, x_1, x_2… x_{M-1} and y_0, y_1, y_2… y_{N-1}.

A two-dimensional spline is defined as tensor product of two one-dimensional splines. Two-dimensional splines share many features with the corresponding one-dimensional splines, for example:

Catmull-Rom Spline Surface	**B-Spline Surface**
$\mathbf{C^1}$-continuous	$\mathbf{C^2}$-continuous
Interpolates control points	Approximates control points

A section of a two-dimensional spline surface along any direction provides a curve, which is a corresponding one-dimensional file. For example, a two-dimensional Catmull-Rom spline is cut on the grid line that corresponds to knot y_2. The section is a one-dimensional Catmull-Rom spline built upon control points p_{20}, p_{21}, p_{22}, … $p_{2,M-1}$.

The behavior of a two-dimensional spline beyond the definition range is more complex than in the case of a one-dimensional spline. **Figure 12** illustrates the Catmull-Rom spline beyond the definition range:

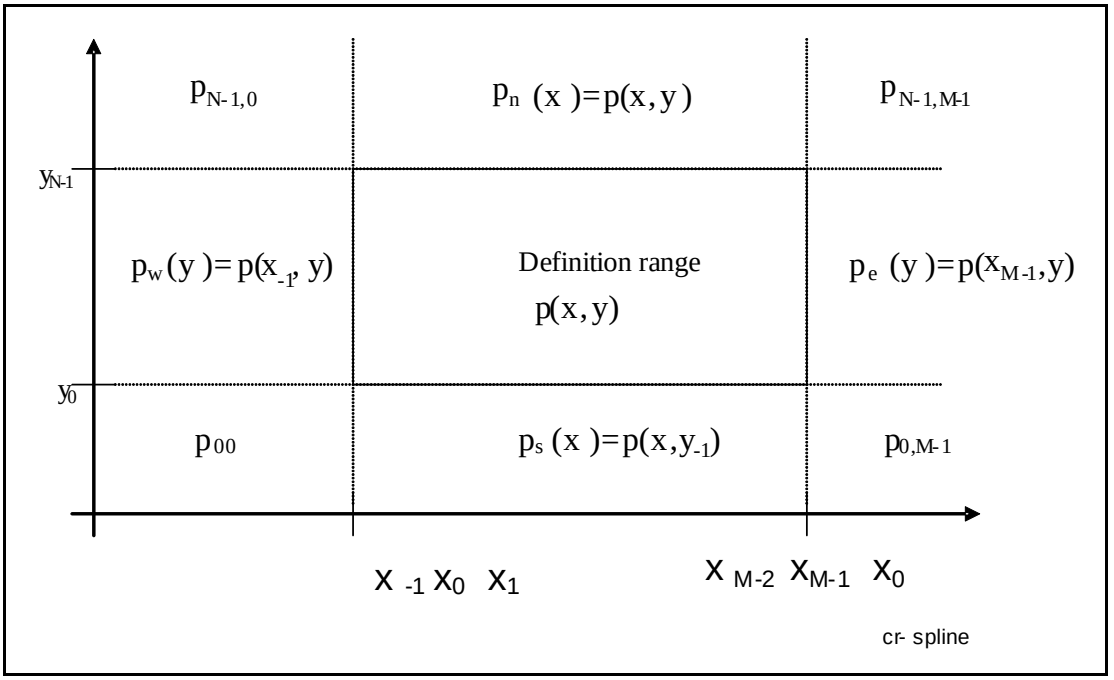


Figure 12 Catmull-Ron Spline Beyond the Definition Range

The behavior of Catmull-Rom depends on the area as follows:

- Within definition range: function of two arguments p(x, y).
- Southeast to definition range: constant value equal to control point p_{00}.
- Northeast to definition range: constant value equal to control point $p_{N-1,0}$.
- Northwest to definition range: constant value equal to control point $p_{N-1,M-1}$.
- Southwest to definition range: constant value equal to control point $p_{0,M-1}$.
- South to definition range: function of x, $p_s(x) = p(x, y_0)$
- North to definition range: function of x, $p_n(x) = p(x, y_{N-1})$
- West to definition range: function of y, $p_w(y) = p(x_0, y)$
- East to definition range: function of y, $p_e(y) = p(x_{M-1}, y)$

Similar to one-dimensional B-spline, two-dimensional B-spline has an extended range of result change. To specify the extended range, let us define four artificial knot points:

$x_{-1} = x_0 - (x_1 - x_0)$
$x_M = x_{M-1} + (x_{M-1} - x_{M-2})$
$y_{-1} = y_0 - (y_1 - y_0)$
$x_N = x_{N-1} + (x_{N-1} - x_{N-2})$

Then, the extended range spans from x_{-1} to x_M and from y_{-1} to y_N.

Area map of a B-spline is similar to the Catmull-Rom spline, but the extended range takes place of the definition range:

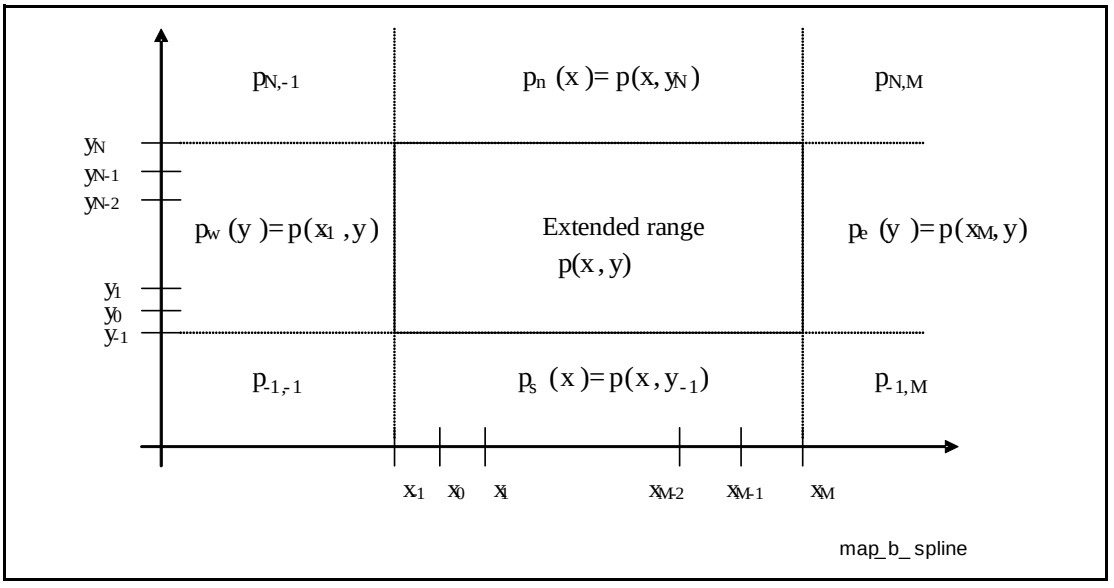


Figure 13 B-Spline Map

4.8.2 pvspline Command

The **pvspline** command is used to create spline motion.

Syntax:

pvspline[/switch] axis_designators [,time_interval]

point axis_designators, coordinate, [coordinate,] velocity [,time_interval]

mpoint axis_designators, point_matrix, number_of_points [,time_interval]

ends

Where **switch** can be one or a combination of:

r	The point coordinates are relative to the previous point.
w	Create the motion, but do not start until the **go** command.
c	Use the point sequence as a cyclic array, that is, after positioning to the last point return to the first point and repeat.
t	Non-uniform time interval; the time interval is specified for each point along with the point coordinates.

Points for PV and PVT motion are defined by **point** and **mpoint** commands (see **Section 4.8.2.1 - point Command** and **Section 4.8.2.2 - mpoint Command**).

Note	
	*The* **point** *and* **mpoint** *commands serve the same function for defining points along the path as the* **point** *and* **mpoint** *commands for multiple point-to-point motion (see* **Section 4.2.2 - mptp, point, mpoint, and ends Commands***); however, the controller employs a different algorithm when calculating the spline motion.*

The **ends** command terminates the point sequence. After the **ends** command, no **point** or **mpoint** commands for this motion are allowed.

The trajectory of the motion follows through the defined points. Time intervals between the points are uniform, or non-uniform as defined by the **t** switch.

Motion generated by the **pvspline** command does not follow the standard motion profile. Variables **VEL**, **ACC**, **DEC**, **JERK** have no effect on this motion. The motion profile is defined exclusively by the positions and velocities specified by the **point** and **mpoint** commands.

The **halt** command (see **Section 4.1.8 - halt Command**) is executed in a specific way with this type of motion. As with other motion types, the controller provides a smooth deceleration profile with the **DEC** value of the leading axis. However, unlike other motions types, the controller does not follow the motion trajectory during deceleration.

Spline motion created without the **t** switch implies uniform intervals between the points. If **pvspline** does not include the **t** switch, the **time_interval** argument has to be included. The argument defines time interval between the motion points in milliseconds.

If **pvspline** does not include the **t** switch, the **time_interval** argument must not be included. Instead, the time interval must be specified for each point as an additional argument for the **point** command or as additional array row in the **mpoint** command.

The **pvspline** command itself does not specify any point, so the created motion starts only after the first point is specified. The points of motion are specified by the **point** or **mpoint** commands that follow the **pvspline** command.

4.8.2.1 point Command

The **point** command for the spline interpolation specifies two values per axis that is involved: the first value defines the coordinate of the end point of the segment; the second value specifies the velocity at the end point. The coordinate is specified in the user units of the axis; the velocity is specified in user units per second.

The following fragment illustrates adding a point with PV spline motion.

`pvspline (0,1,3),10`	Create PV spline motion for axes 0, 1, and 3. Points are given at 10 millisecond intervals.
`point (0,1,3),200,100,300,1000,2000,1500`	Add a point with coordinates 0=200, 1=100, 3=300; velocities at the point are V_0=1000, V_1=2000, V_3=1500.

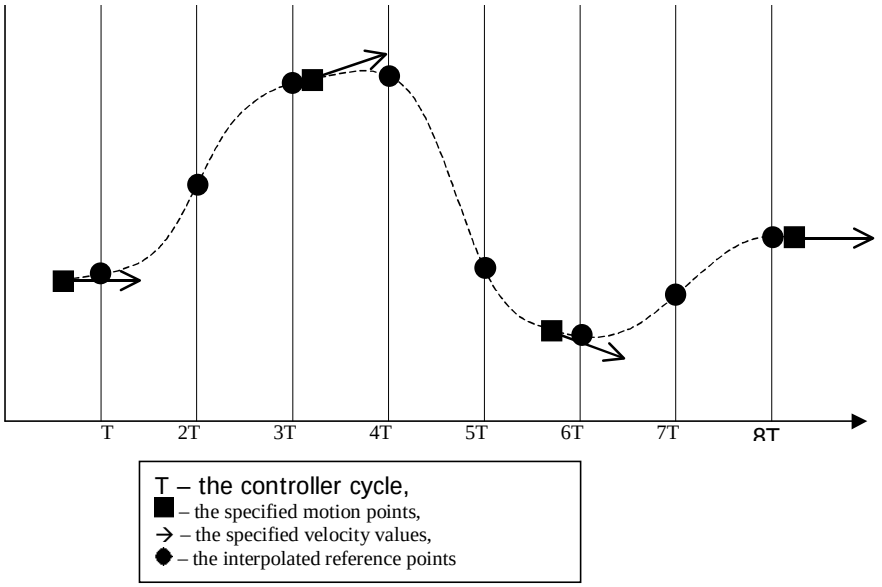
For each segment the controller constructs a third-order polynomial and calculates the reference coordinates using the polynomial.

The spline provides exact track through the specified points and exact specified velocity at the points. The spline also provides continuous velocity at all intermediate points.

In general the spline does not guarantee acceleration continuity at the connection points. However, the acceleration can be continuous if the proper velocity values are specified, and many host-based programs that prepare data for PV-interpolation actually calculate velocity values that will provide continuous acceleration.

The time interval between the points may be either uniform or non-uniform. In both cases the time interval is not required to be an integer or to be equal to an integer number of controller cycles. The controller uses the exact specified time interval to calculate the interpolated reference positions.

The following drawing illustrates the PV spline interpolation:



In the **point** command the **axis_designators** must specify the same axes in the same order as in the **axis_designators** of the corresponding **pvspline** command.

The other arguments contain different values depending on corresponding **pvspline** command.

If the related motion command is a **pvspline** without the **t** switch for M axes, there has to be 2*M arguments: 2 arguments per axis involved. The arguments specify the end point coordinates and the coordinate velocities at the end point in the following order:

- The end point coordinate for each axis involved (M values)
- The velocity at the end point for each axis involved (M values)

Each coordinate is specified in user units of the corresponding axis, each velocity is specified in user units per second.

If the related motion command is **pvspline/t** for M axes, there have to be 2*M+1 arguments: 2*M arguments specify the end point coordinates and velocities, and the last argument specifies the time interval between the previous and the current point. The time is specified in milliseconds.

4.8.2.2 mpoint Command

The **mpoint** command adds an array of points to PV or PVT spline motion.

The arguments of the **mpoint** command are:

- **axis_designators** - must specify the same axes in the same order as in the axes-specification of the corresponding mptp or path command.
- **point_matrix** - name of declared two dimensional array.
- **number_of_points** - specifies how many points are added to the motion by the command.

  Before the **mpoint** command can be executed, an array must be declared and filled with the point coordinates. Each row of the array contains coordinates of one point.

If the related motion command is **pvspline** without the **t** switch, for M axes the matrix must contain 2*M rows, 2 rows per axis involved. The values in each column specify:

- in row 1: the end point coordinate for each axis involved (M values)
- in row 2: the velocity at the end point for each axis involved (M values)

Each coordinate is specified in user units of the corresponding axis, each velocity is specified in user units per second.

If the related motion command is **pvspline/t**, for M axes the matrix must contain 2*M+1 rows:

- M rows for end point coordinates
- M rows for end point velocities
- plus additional row for the time interval between the points. The time is specified in milliseconds.

4.8.3 Spline Motion Variables

As mentioned, the spline motion does not use the values of the **VEL**, **ACC**, **DEC**, **JERK** variables for motion profile construction. The motion profile is constructed using the coordinates and velocities specified in the segment end points.

While the motion is in progress the controller updates the following read-only variables every controller cycle.

- **APOS** and **RPOS** are updated as for any other motion and read the motion result.
- Bits in **AST**, **MST** are updated as for any other motion and read the motion state.
- **GSEG** is updated for the leading axis with the number of the currently executed segment.
- **GSFREE** is updated for the leading axis with the number of free cells in the segment queue. If **GSFREE** is zero, the segment queue is full and the next coming point or mpoint command will be delayed until the required number of cells will be freed.
- **GVEC** is updated with the instant velocity for each axis involved. The **GVEC** values build up a vector of instant velocity and also can be used for retrieving a tangent vector.
- **GPATH**, **GVEL**, **GACC**, **GJERK**, **GPHASE**, **GRTIME** are not updated while the motion is in progress.

The following example illustrates how the **pvspline** command can be used for adding points on-the-fly. The example also shows a simple approach to synchronization between the host-based program that calculates the points and the controller that executes the motion.

The host-based program calculates the desired points and transmits the coordinates via Ethernet link. The motion involves 6 axes. Therefore, each point specification contains 12 real numbers (6 coordinates and 6 velocities).

Because transmitting each point separately would be a very inefficient use of the Ethernet, the host calculates the desired points in advance and transmits them in batches of 50 points. The controller then executes the motion. As soon as the controller is ready to accept the next batch of points and the host is ready to send that batch, the next transmission occurs, and so on.

The pace of the host and the controller do not have to be identical. However, the host is assumed to be fast enough to calculate the next 50 points before the controller has moved through the previous 50 points.

The controller executes the following program:

`real Points(12)(50)`	Declare an array of 50 points. The host will write the coordinates and velocities to the array.
`int N,HSync,NBuf`	Declare a synchronization variable (initiated to zero by default).
`pvspline XYZTAB, 10`	Create PV for axes XYZTAB. Points are given at 10-millisecond intervals.
`while Sync >= 0`	Continue until the host writes negative number to Sync.
`till Sync`	Wait until the points are received. Once the host has filled the Points array, it writes the Sync variable with a number of points written to the Points array.
`if Sync > 0`	Sync greater than 0 indicates that the host has finished the point generation.
`mpoint XYZTAB, Points, Sync`	Add points to the Points matrix.
`Sync = 0`	The controller informs the host that the next batch is expected. The motion through the points already received from the host has not completed, but the controller is ready to receive more points.
`end`	End if.
`end`	End while.
`ends XYZTAB`	End **pvspline**.
`stop`	

The program running on the host looks like the following pseudo code:

```
double HPoints(12)(50);
int N, HSync;
HANDLE Com;

open communication, start program in buffer NBuf of the controller;

while (Continue)
calculate N (<= 50) points in array Hpoints;

acsc_WriteReal(Com, NBuf, "Points" , 0, 11, 0, N-1, HPoints, 0) ;
acsc_WriteInteger(Com, NBuf, "Sync", -1, -1, -1, -1, &N, 0);

do
acsc_ReadInteger(Com, NBuf, "Sync", -1, -1, -1, -1, &HSync, 0);
while HSync;
reset Continue to zero if all points have been calculated;
end;

N = -1
acsc_WriteInteger(Com, NBuf, "Sync", -1, -1, -1, -1, &N, 0);
```

Synchronization between the host and the controller is provided by the **Sync** user variable. When the host has finished transmitting the next batch of points to the controller, it writes to **Sync** the number of points in the batch. The controller waits for non-zero **Sync** and then adds the points to the motion. When the controller has added the points, it writes zero to **Sync**, which signals to the host to transmit the next batch of points.

When the host comes to the end, it writes -1 to **Sync**, to indicate the end of the motion.

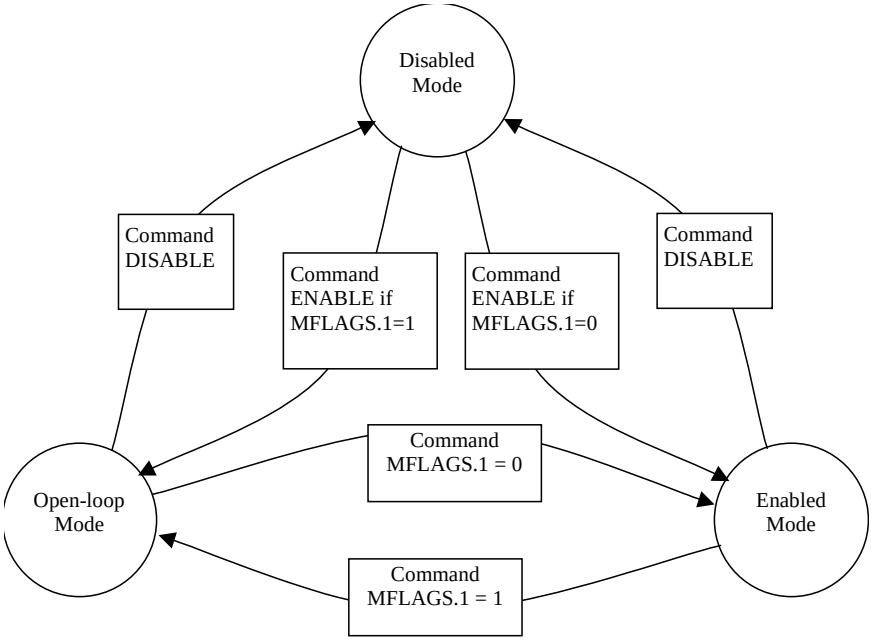
4.9 Open-Loop Operation (Torque Control)

The open-loop motor mode is often referred as a torque-control mode. The following table summarizes differences between three motor modes:

Table 8 Motor Modes

Mode	Disabled	Open-loop	Enabled
State of the Drive Enable output	Off	On	On
Calculation of control loop algorithm	No	No	Yes
Correspondence of Reference Position (**RPOS**) and Feedback Position (**FPOS**)	**RPOS** follows **FPOS**	**RPOS** follows **FPOS**	**RPOS** is calculated according to the commanded motion. **FPOS** follows the **RPOS** as provided by control loop algorithm.
Position Error (**PE**)	Zero	Zero	The control algorithm calculates **PE** = **RPOS** - **FPOS**
Voltage at the drive output	Zero	Proportional to **DCOM**	As calculated by the control algorithm

Bit 1 in the **MFLAGS** variable enables or disables open-loop mode. The following diagram explains transition between the three modes:



The circles in this diagram represent the motor modes and the arrows with rectangles show the controller commands that switch the controller from one mode to another.

As shown in the diagram, a motor can be switched from the enabled mode to the open-loop mode and back without changing to the disabled mode. The controller provides a smooth

transition between the modes. Even if the motor experiences uncontrolled movement while in the open-loop mode, switching back to the enabled mode does not cause any motor jump. However, be careful if you execute a motion while the controller is in open-loop mode. Once the command switches back to enabled mode, the controller continues the motion from the point where it finds the motor. No motor jump occurs, but the motion trajectory and the final point may be shifted by the value of uncontrolled offset in the open-loop mode.

While in open-loop mode, the controller calculates the drive voltage output based on the **DCOM** variable. The **DCOM** variable sets the drive output as a percentage of the maximum available drive output voltage. For example, the SPiiPlus PCI 4/8 provides differential drive output in the range from -10V to +10V. Therefore, assigning 100 to **DCOM** provides +10V on the drive output, assigning -100 provides -10V, and assigning 0 provides 0V.

The following program fragment shows an example of torque control implementation through the open-loop mode.

`enable X`	Enable the X motor.
`ptp/f X,400,100`	Move to the point where the contact search begins. Provide low (search) velocity of 100 count/sec in the final point.
`jog/v X,100`	Move to the contact point using the search velocity.
`till IN0.4; kill X; MFLAGS0.1=1; DCOM0=30`	Wait for the signal from the contact sensor. Kill the motion. Switch to open-loop mode. Apply 30% of the maximum torque.
`wait 50; DCOM0=10`	Wait 50 milliseconds and then change torque to 10% of the maximum torque.
`wait 100; MFLAGS0.1=0; ptp X,400`	Wait 100 milliseconds, then switch off the open-loop mode and move away from the contact point.

4.10 Step Velocity Profile (Non-Zero Minimal Velocity)

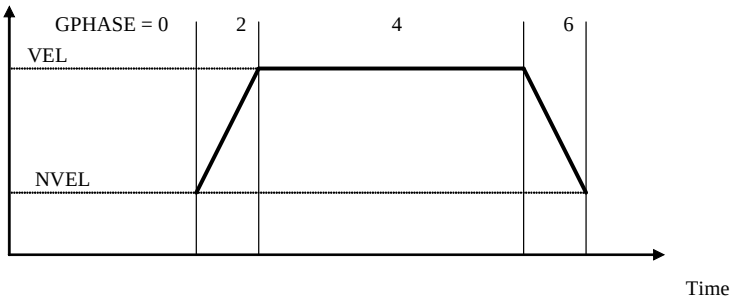
Many applications with step motors require the motion velocity to be restricted from below. In this case, the motion profile has non-zero start and finish velocity. To handle this case you can use the **NVEL** variable (see *SPiiPlus Command & Variable Reference Guide*) that specifies minimal velocity for each axis. If an application sets a non-zero **NVEL** for some axis, the controller uses the value as start and finish velocity in any motion related to the axis except for an axis that is governed by the **path** or **pvspline** command.

4.10.1 The NVEL Variable

NVEL is an array of 8 real values, one per axis. A non-zero value in any **NVEL** element determines that the motion of the corresponding axis will be executed with non-zero start and finish velocities.

If an **NVEL** element is zero, the normal motion profile starts from zero velocity and finishes at zero velocity. If an element is non-zero, in the beginning of motion the velocity immediately jumps to the value specified in the **NVEL** element and then continues normal motion profile. In the end, the motion approaches the final point at velocity specified in the **NVEL** element, then the velocity immediately drops to zero.

In a typical application, the step motor does not require acceleration build-up phases; the motion profile is trapezoidal:

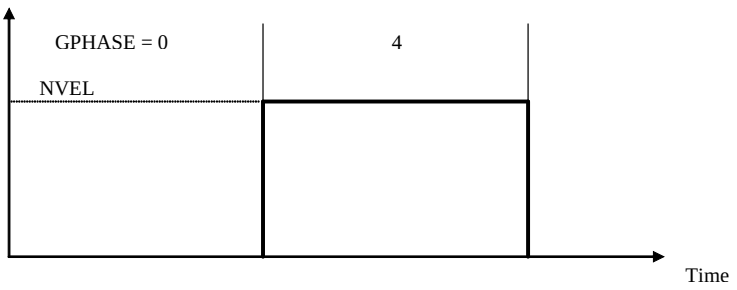


kill and **halt** commands are also affected, i.e., they slow down the velocity to the value specified in the **NVEL** element and then the velocity drops to zero.

4.10.2 Special NVEL Cases

4.10.2.1 Specified Velocity Less Than NVEL

If the absolute value of the **VEL** value is less than the **NVEL** value, the **VEL** value is ignored. In this case the motion profile is rectangular:



Also ignored are the velocity specified in a motion command or in an **imm** command if its absolute value is less than **NVEL**. In all cases, the actual motion velocity is not less than **NVEL**.

4.10.2.2 Multi-Axis Motion

Multi-axis motion is not typical for an axis with non-zero minimal velocity. If, however, the axis is involved in multi-axis motion, the controller uses **NVEL** as follows:

- If the axis is a leading axis in the motion, its **NVEL** is used as a minimal vector velocity in the motion profile
- If the axis is not leading, its **NVEL** is ignored.

In both cases, the axis velocity can be lower than the NVEL value. Therefore, be careful in defining multi-axis motion that involves both stepper motors and servo motors.

4.10.2.3 NVEL and Non-Default Connection

If a non-default connection is specified (see **Section 8.12 - Non-Default Connections**), the axis and the motor are different entities. In this case, **NVEL** refers to the axis, but not to the motor. For example, if **NVEL0** is set to 500, the velocity of any motion of the X axis will be limited from below to 500 units per second irrespective of which motor is affected by the motion.

5 Inputs and Outputs

The controller includes digital and analog inputs and outputs. This chapter discusses the following:

- General purpose digital inputs and outputs
- General purpose analog inputs and outputs

Safety inputs and digital encoder implementations are discussed in other sections of this guide (see **Chapter 6 - Fault Handling**).

The controller provides a set of general-purpose inputs and outputs that have no predefined function. You can assign a function to any input/output as required by your application.

The exact number of general purpose digital inputs and outputs depends on the controller configuration.

5.1 Digital Inputs and Outputs

Digital Inputs - A digital input is a binary signal in the form of low or high voltage that the controller accepts from an external source such as a switch or a relay.

Digital Outputs - A digital output is a binary signal that the controller provides to an external acceptor such as a LED or actuator.

5.1.1 Addressing Digital I/Os

Digital Inputs are presented by the ACSPL+ read-only integer array variable: **IN**. Digital Outputs are presented by the ACSPL+ integer array variable: **OUT**.

Each member of the array is a bitmask of input or output states, respectively. During the system configuration process, array members are bound to a certain unit, according to amount of I/O that the unit supports.

Note	
	*When you are operating a system whose configuration is currently unknown, you can use the Terminal command:* **#SI** *(see SPiiPlus ACSPL+ Command & Variable Reference Guide for details on Terminal commands) to find out the correlation between the* **IN**/**OUT** *array indexes and I/O.*

You address a digital I/O using the following format:

IN(port_#).bit_# or **INport_#.bit_#** – Input

OUT(port_#).bit_# or **OUTport_#.bit_#** – Output

Where:

IN **OUT**	Integer array **IN**, a read-only array Integer array **OUT**
port_#	The port number to which the bits belong. The controller's digital input/output ports are numbered from 0 to N-1, where N is the number of controller ports (see the controller's Hardware Guide for the number of input/output for your controller).
.bit_#	The specific bit within the port. Each port is divided into 32 bits that are numbered from 0 to 31. For example: **IN0.1** – input 1 of port 0 **IN3.19** – input 19 of port 3 **OUT0.5** – output 5 of port zero **OUT3.19** – output 19 of port 3

Rather than explicitly designating the port number, you can use an integer user-defined variable that equates to the number. In this case you have to include the parentheses, for example:

IN(u_var).1 - where **u_var** is an integer variable the value of which equates to the port number.

Note	
	*If the controller provides only 32 inputs or less, all inputs/outputs are located in port zero. In this case the port number can be omitted, and input is referred as:* **IN.0** (for input 0), **IN.22** (for input 22), **OUT.0** (for output 0), **OUT.2** (for output 2), *etc.*

5.1.2 Querying Digital I/Os

The **IN** and **OUT** arrays can be queried like any other variable in the Communication Terminal. Each element of the array is read as a 32-bit binary number.

Example:

```
: ?IN0                  What is the status of input 0
10111001,00011010,00000100,00000000
: ?IN0.1,IN0.2          What is the status of bits 1 & 2 of input 0
0
1
: ?OUT(0,3)             What is the status of outputs 0 and 3
10111001,00011010,00000100,00000000 10111001,00011010,00000100,00000000
10111001,00011010,00000100,00000000 10111001,00011010,00000100,00000000
```

5.1.3 Assigning Outputs

You can assign only **OUT** elements. The **IN** array is read-only.

Each digital input/output is treated as one binary bit. The low voltage level corresponds to zero (or "clear") and high voltage level corresponds to one or ("set").

Examples of assignment to the elements of an **OUT** are shown below:

`OUT0.1 = 0`	Set output OUT0.1 to zero
`OUT0.1 = 1`	Set output OUT0.1 to one
`OUT0.1 = V0`	If V0 = 0, set OUT0.1 to zero. Otherwise, set OUT0.1 to 1
`OUT0.15 = IN0.10`	Copy state of input IN0.10 to output OUT0.15
`OUT0.15 = ~ IN0.10`	Copy inverse state of input IN0.10 to output OUT0.15
`OUT6.1 = IN0.0 & IN0.1`	Set OUT6.1 to logical AND of inputs IN0.0 and IN0.1
`OUT0 = 0x0101`	Set signals OUT0.0 and OUT0.8 to one. Set all other bits of OUT0 to zero
`OUT0 = OUT0 | 0x0101`	Set signals OUT0.0 and OUT0.8 to one. Do not alter other bits of OUT0
`OUT0 = OUT0 & ~0x0101`	Set signals OUT0.0 and OUT0.8 to zero. Do not alter other bits of OUT0.
`OUT0 = (OUT0 & ~0x0101)|(V1 & 0x0101)`	Copy bits 0 and 8 from V1 to OUT0. Do not alter other bits of OUT0

5.1.4 Digital I/O in Conditional Commands

Commands such as **if**, **while** and **till** are always followed by a logical expression. Using I/O in the logical expression provides program branching options that are I/O state-dependent.

Examples:

`if ^IN0.1 goto L`	Go to label L only if IN0.1 is zero
`while IN0.1 & IN0.2`	Execute the subsequent commands up to command end while both IN0.1 and IN0.2 are one
`till IN0.10`	Wait until IN0.10 becomes one
`till IN0 & 0x0101`	Wait until at least one of IN0.0 and IN0.8 becomes one.

5.1.5 PLC Implementation

Programmable Logic Controller (PLC) is often used to manage digital inputs and outputs. SPiiPlus controllers provide implementation of PLC without separate PLC hardware. The techniques described in this section provide implementation of PLC functionality by the controller. This approach provides an easy integration of PLC program with motion control. For example, a motion can be started when a condition calculated by the PLC program is satisfied, and an output can be activated when a motion starts or terminates.

There are several options for implementing a PLC program:

- Implement the PLC program in a separate buffer. This is the most suitable approach if the PLC program must not interfere with motion, and has few connections to motion programs. The PLC program runs in parallel with motion programs in other buffers, and any desired connections are provided via global variables.
- Mix motion programs and PLC program in the same buffers. This approach provides a very close interaction between PLC and motion programs, resulting in faster reaction time, but in general has a more complex structure.
- Split the PLC program into two different parts running in two different buffers. This approach is most suitable when a time-critical part of the program has to operate faster than the rest of the program. PLC programs run at either a fast or slow scanning rate, and you must assign a greater priority one buffer using the **PRATE** variable.
- Implement a part of the PLC program as a set of autoroutines. This approach provides a very fast and interrupt-like response to critical conditions, because the autoroutine condition is checked each controller cycle.

The following is an example of a PLC program implemented in a separate buffer using autoroutines for fast response:

```
1    real T1
2    int Bits
3    Start:
4    OUT0.0 = MST0.#INPOS
5    if T1 <= TIME
6          if Bits.0  T1 = T1 + 30000  else  T1 = T1 + 15*60000  end
7    Bits.0 = ^Bits.0
8    end
9    OUT0.4 = IN0.4 & Bits.0
10   goto Start
11   on IN0.15; killall; ret
```

Line1 – Definition of local variable **T1** that is used to store the next switch time. **T1** may be defined as integer, but as a real, it can provide continuous running for an extended period without overflow. The program relies on the automatic initialization of all local variables to zero when they are declared.

Line2 – Definition of a local variable: **Bits**. In this program only one bit of **Bits** is used. One temporary integer variable can be used for storing 32 temporary bits.

Line3 – A label: **Start**. A typical case in PLC programming is a long program cycle that executes to the end and returns to the beginning. In the example shown above, the execution period is quite short even with default rate of 'one Line per each controller cycle'. In a long program, the execution cycle can reach hundreds of milliseconds.

This is a good reason to divide a typical PLC program into slow and fast sections.

Line 4 – **OUT0.0** reflects the 'in position' state of the motor. If the motor is not in position, the output is 0. If it is in position, the output is 1.

Lines 5-9 – **OUT0.4** controls a periodic activity that must be executed every 15 minutes for a 30-second period. It is executed only if **IN0.4** is active. In a typical application, the output might be connected to lubrication pump.

Line10 – Returns the motion to the **Start** point.

Line11 – An autoroutine that provides extra fast response to **IN0.15**, typically an emergency input. The whole autoroutine is implemented in one line providing an immediate kill of all motions within one controller cycle when input port 0 bit 15 is 1.

5.1.6 Digital I/O in Autoroutines

You may use digital I/O as conditions for autoroutines (see **Section 3.8.3 - Autoroutines**). The autoroutine can be very useful for PLC implementation and fault handling.

For example:

```
ON IN0.0      ! When input#0=0
OUT0.4=1      ! Set output#4 to 1
disp "Activates motor"
RET           ! Ends the autoroutine
```

5.1.7 Using HSSI I/O Extension

Use the High-Speed Synchronous Serial Interface (HSSI) channels available in the SPiiPlus controllers for connecting additional inputs and outputs. ACSPL+ supports access to HSSI through the standard arrays **EXTIN** and **EXTOUT**.

The arrays can be queried, indexed and used in expressions like other ACSPL+ variables. For detailed information about the HSSI interface see the *HSSI Modules Hardware Guide*.

5.2 Analog Inputs and Outputs

The controller provides a set of analog inputs and outputs. This section discusses the general purpose analog I/Os whose number is controller dependent.

Analog input - accepts analog signal from an external source, such as a sensor or a potentiometer.

Analog output - provides analog signal to an external receiver, such as an actuator or a measuring device.

Analog inputs/outputs have no predefined function in the controller. You can connect signals to inputs/outputs and process them as required by the application.

5.2.1 Addressing Analog I/Os

Analog Inputs are presented by the ACSPL+ read-only integer array variable: **AIN**. Analog Outputs are presented by the ACSPL+ integer array variable: **AOUT**.

Each member of the array is a value of input or output, respectively. During the system configuration process, array members are bound to a certain unit, according to amount of IO, that the unit supports.

Note	*When you are operating a system whose configuration is currently unknown, you can use the Terminal command:* **#SI** *(see SPiiPlus ACSPL+ Command & Variable Reference Guide for details on Terminal commands) to find out the correlation between the* **AIN**/**AOUT** *array indexes and I/O.*

You address an analog I/O using the following format:

AIN(port_#)[.bit_#] or **AINport_#[.bit_#]** – Input

AOUT(port_#)[.bit_#] or **AOUTport_#[.bit_#]** – Output

Where:

AIN **AOUT**	Integer array **AIN**, a read-only array Integer array **AOUT**
port_#	The port number to which the bits belong. The controller's input/output ports are numbered from 0 to N-1, where N is the number of controller ports (see the controller's Hardware Guide for the number of input/output for your controller).
.bit_#	The specific bit within the port. Each port is divided into 32 bits that are numbered from 0 to 31. For example: **AIN0.1** – input 1 of port 0 **AIN3.19** – input 19 of port 3 **AOUT0.5** – output 5 of port zero **AOUT3.19** – output 19 of port 3

The range of the **AIN** and **AOUT** arrays depends on the type of the input or output and the bit resolution of the Analog-to-Digital or Digital-to-Analog conversions.

- Example: For ±10V analog outputs with 16-bit Digital-to-Analog conversion resolution, the **AOUT** range is from -32768 (for –10V) to +32767 (for +10V).
- Example: For ±1.25V analog inputs with 14-bit Digital-to-Analog conversion resolution, the **AIN** range is from -8192 (for –1.25V) to +8192 (for +1.25V).

Note	*If an analog output is connected to a drive, it has a dedicated destination and cannot be used as a general purpose analog output.*

For model-dependent analog I/O information (for example, the number and range of inputs and outputs) see the controller's Hardware Guide.

5.2.2 Assiging Analog Outputs

You can assign any variable (ACSPL+ or user-defined) to an entity of **AOUT**. For example, ACSPL+ variables:

- **FPOS** – Feedback position
- **FVEL** – Feedback velocity
- **FACC** – Feedback acceleration

Note	**FVEL** *is calculated by digital differentiation of* **FPOS**. **FACC** *is calculated by digital differentiation of the* **FVEL** *variable.* *The* ACSPL+ **FVFIL** *variable defines a power of the smoothing filter used in the* **FVEL** *calculation (see SPiiPlus Command & Variable Reference Guide).*

Example:

The following ACSPL+ program assigns X-axis feedback position to output #3 and acceleration to output #1. In the example the user variables: **SF1** and **SF2** are scale factors.

```
real SF1, SF2;
SF1=0.01 ; SF2=0.001;        !Define scaling factors
While 1;
AOUT3 = FPOS0 * SF1;
AOUT1 = FACC0 * SF2;
END
```

If the acceleration signal is too noisy, increase **FVFIL**.

6 Fault Handling

Fault handling is not only a concern for isolating motion faults, it is also a vital concern to personal and equipment safety.

Safety features are necessary to protect both the equipment and you from potential injury. SPiiPlus controllers include numerous safety-related features, but the final responsibility for the safe use of the controller in a particular application lies with you. Before you create your application make sure that you thoroughly read and understand this chapter.

This chapter addresses:

- ❑ Safety Controls
- ❑ Working with Fault Procedures

6.1 Safety Control

Warning	
	***Some alarms, limits, and errors involve protection against potentially serious bodily harm and equipment damage. Be aware of the implications before changing or disabling any alarm, limit, or error.***

Safety control is one of the highest-priority tasks of the controller. The controller continually monitors safety conditions each controller cycle, in parallel to its other activities.

The controller sets one of the fault bits of the ACSPL+ **FAULT** variable when it detects a malfunction. The response to a fault may vary from sending a fault message to complete termination of all activities. For each fault type you can enable/disable the default controller response or define your own autoroutine response.

6.1.1 Types of Malfunctions

The controller monitors numerous safety conditions that may indicate different hardware or software malfunctions (faults). The most frequent causes of faults are given in **Table 9**:

Table 9 Types of Malfunctions (page 1 of 2)

Type of Fault	Examples
User error	Defining a required velocity that is invalid or beyond the limits.
Improper or broken wiring	A loose connection in the feedback encoder wiring.
Power amplifier malfunction	The power amplifier malfunctions and sends a fault signal to the controller.
Motor malfunction	A motor overheats.

Table 9 Types of Malfunctions (page 2 of 2)

Type of Fault	Examples
Controlled plant malfunction	The Emergency Stop input is activated.
Controller hardware malfunctions	The Main Processor Unit (MPU) and the Servo Processors (SPs) work together to detect malfunctions in the controller. Examples include the servo processor alarm and the servo interrupt.

6.1.2 How the Controller Detects Malfunctions

To detect malfunctions the controller monitors safety inputs, such as limit switches, and internal safety conditions, such as comparing the reference position with the software limits.

Internal safety conditions may consist of a static formula, such as checking the acceleration limit, or may include time dependencies, such as measuring the time interval between two interrupts to detect servo interrupt faults.

Some safety conditions are a mixture of both techniques. For example, position error control is based on both a static condition, whether a motor is positioned at a certain location, and a time dependency, how long the motor is positioned at the location.

6.1.3 Faults

When the controller detects a malfunction, it raises a specific fault bit. Fault bits are grouped into ACSPL+ **FAULT** and **S_FAULT** variables.

In certain cases, you may want to define which fault conditions are examined in a specific application. The ACSPL+ **FMASK** and **S_FMASK** variables specify which fault conditions must be examined in a particular application.

See *SPiiPlus Command & Variable Reference Guide* for complete details of these variables.

6.1.3.1 The FAULT Variable

The **FAULT** variable is an integer array containing eight elements (corresponding to the number of motors), where each element is made up of a set of bits. Each bit indicates a motor fault. Motor faults are related to a specific motor, power amplifier, or Servo Processor. Examples include tracking error and motor overheat.

6.1.3.2 The S_FAULT Variable

S_FAULT is a scalar variable, where each bit represents the aggregate status of a particular fault. The bits of **S_FAULT** are divided into two categories:

- Aggregated motor faults – Once the controller raises a bit in any element of **FAULT**, it immediately raises the corresponding bit of **S_FAULT**. Therefore, each bit of **S_FAULT** is an OR aggregate of the corresponding bits in all elements of **FAULT**.
- System faults such as Emergency Stop and Time Overuse that are not related to any specific motor.

6.1.4 Controller Response

The controller response to a fault can vary according to the requirements of your application:

- ❑ No response.
- ❑ Default response – One or more predetermined actions. You can disable the default response for any fault.
- ❑ Autoroutine response – User-defined actions implemented in an autoroutine. In the autoroutine you select a controller fault and controller responses to the fault. An autoroutine can replace the default response or supplement it with additional actions.

6.2 Safety Control Summaries

A fault is a critical error for which the controller provides a default response. You can control the response: deactivating it or changing it.

6.2.1 Summary of Faults and Default Responses

A fault can be either a motor or system fault. Motor faults refer to a specific motor, power amplifier, or Servo Processor and affect the state of the corresponding bit for that element of the **FAULT** variable. System faults do not refer to a specific motor. The corresponding bits are located in the **S_FAULT** variable.

For most controller-detected faults there is a default response, which is normally executed automatically when the fault occurs. Your ACSPL+ application can simply allow the default response for a fault or you can do either or both of the following:

- ❑ Disable the default response by using ACSPL+ variables **FDEF** and **S_FDEF**.
- ❑ Create an autoroutine (implementing your preferred response) activated by the occurrence of the fault. If desired, you can leave the default response enabled so that it will execute together with the autoroutine.

In addition to the above, each motor fault is either:

1. Latched in the **MERR** variable (see *SPiiPlus Command & Variable Reference Guide*) indicating that the motor is disabled or killed, in which case you can build a little routine that checks the value of **MERR** and resets it by running the **fclear** or **enable** (**enableall**) command, or
2. Set in the **MERR** variable only so long as the fault condition exists.

Table 10 provides a brief rundown of the faults reported by the ACSPL+ fault variables, as well as those latched in **MERR**.

Note	*In the table Right Limit restricts the motion in the positive direction and the Left Limit in the negative direction.*

Table 10 Faults and the Controller's Default Response (page 1 of 4)

Bit	Designator	Type	Fault Description	Default Response
0	#RL	Motor	RIGHT LIMIT. The controller raises the fault bit when the right limit switch is activated.	The controller kills the violating motor. As long as the fault is active, the controller kills any motion that tries to move the motor in the direction of the limit. Motion to return to the allowed range of motion is allowed.
1	#LL	Motor	LEFT LIMIT. The controller raises the fault bit when the left limit switch is activated.	The controller kills the violating motor. As long as the fault is active, the controller kills any motion that tries to move the motor in the direction of the limit. Motion to return to the allowed range of motion is allowed.
2	#NT	Network	NETWORK ERROR. The controller raises the fault bit when a loss of network is deteced.	The controller disables all axes until a valid network Sync signal is received.
3	#FAN	Motor	COOLING FAN FAULT. The controller raises this fault bit when it detects that the cooling fan is not on.	The controller kills the violating motor. As long as the fault is active, the controller kills any motion that tries to move the motor. This condition remains until the cooling fan is detected.
5	#SRL	Motor	SOFTWARE RIGHT LIMIT. The controller raises the fault bit when the motor reference position (**RPOS**) is greater than the software right limit margin (**SRLIMIT**).	The controller kills the violating motor. As long as the fault is active, the controller kills any motion that tries to move the motor in the direction of the limit. Motion in the direction away from the limit is allowed.
6	#SLL	Motor	SOFTWARE LEFT LIMIT. The controller raises the fault bit, when the motor reference position (**RPOS**) is less than the software left limit margin (**SLLIMIT**).	The controller kills the violating motor. As long as the fault is active, the controller kills any motion that tries to move the motor in the direction of the limit. Motion in the direction away from the limit is allowed.

Table 10 Faults and the Controller's Default Response (page 2 of 4)

Bit	Designator	Type	Fault Description	Default Response
7	#ENCNC	Motor	ENCODER NOT CONNECTED. The controller raises the fault bit when the primary encoder is not connected.	The controller disables the violating motor. The error code is latched in the **MERR** variable and remains active until you resolve the problems and enable the motor again or issue the **fclear** command.
8	#ENC2NC	Motor	ENCODER 2 NOT CONNECTED. The controller raises the fault bit when the secondary encoder is not connected.	No default response. The error code is latched in the **MERR** variable, and remains active until you resolve the problems and enable the motor again or issue the **fclear** command.
9	#DRIVE	Motor	DRIVE ALARM. The controller raises the fault bit when the signal from the drive reports a failure.	The controller disables the violating motor.
10	#ENC	Motor	ENCODER ERROR. The controller raises the fault bit when the primary encoder malfunctions.	The controller disables the violating motor. The error code is latched in the **MERR** variable and remains active until you resolve the problems and enable the motor again or issue the **fclear** command.
11	#ENC2	Motor	ENCODER 2 ERROR. The controller raises the fault bit when the secondary encoder malfunctions.	The controller disables the violating motor. The error code is latched in the **MERR** variable and remains active until you resolve the problems and enable the motor again or issue the **fclear** command.
12	#PE	Motor	NON-CRITICAL POSITION ERROR. The controller raises the fault bit when the position error (**PE**) limit is exceeded. The limit depends on the motor state and is defined by the following variables: ❑ **ERRI** if the motor is idle (not moving) ❑ **ERRV** if the motor is moving with constant velocity ❑ **ERRA** if the motor is accelerating or decelerating	None.

Table 10 Faults and the Controller's Default Response (page 3 of 4)

Bit	Designator	Type	Fault Description	Default Response
13	#CPE	Motor	CRITICAL POSITION ERROR. The controller raises the fault bit when the position error (**#PE**) exceeds the value of the critical limit. Whereas **#PE** errors occur during normal operation, **#CPE** is assumed to occur outside normal operation parameters and **#CPE** is greater than **#PE**. The critical limit depends on the motor state and is defined by the following variables: ❑ **CERRI** if the motor is idle (not moving) ❑ **CERRV** if the motor is moving with constant velocity ❑ **CERRA** if the motor is accelerating or decelerating	The controller disables the violating motor.
14	#VL	Motor	VELOCITY LIMIT. The controller raises the fault bit when the absolute value of the reference velocity (**RVEL**) exceeds the limit defined by the **XVEL** variable.	The controller kills the violating motor.
15	#AL	Motor	ACCELERATION LIMIT. The controller raises the fault bit when the absolute value of the reference acceleration (**RACC**) exceeds the limit defined by the **XACC** variable.	The controller kills the violating motor.
16	#CL	Motor	CURRENT LIMIT. The controller raises the fault bit, when the RMS current calculated in the Servo Processor exceeds the limit value defined by the **XRMS** variable.	The controller disables the violating motor.
17	#SP	Motor	SERVO PROCESSOR ALARM. The controller raises the fault bit when the axis Servo Processor loses its synchronization with the main processor. The fault indicates a fatal problem in the controller.	The controller disables the violating motor. The error code is latched in the **MERR** variable and remains active until you resolve the problems and enable the motor again or issue the **fclear** command.

Table 10 Faults and the Controller's Default Response (page 4 of 4)

Bit	Designator	Type	Fault Description	Default Response
20	#HSSINC	Motor	HSSI NOT CONNECTED. The controller raises the fault bit if the HSSI expansion module is not connected.	None.
25	#PROG	System	PROGRAM FAULT. The controller raises the fault bit when a run time error occurs in one of the executing ACSPL+ programs.	The controller kills all motors.
26	#MEM	System	MEMORY FAULT. The user application requires too much memory.	The controller kills all motors.
27	#TIME	System	TIME OVERUSE. The user application consumes too much time in the controller cycle.	No default response.
28	#ES	System	EMERGENCY STOP. The controller raises the fault bit when the **ES** signal is activated.	The controller disables all motors.
29	#INT	System	SERVO INTERRUPT. The servo interrupt that defines the controller cycle is not generated. The fault indicates a fatal controller problem.	The controller disables all motors.
30	#INTGR	System	INTEGRITY VIOLATION. The controller raises the fault bit when an integrity problem is detected.	No default response

6.2.2 Summary of Safety Inputs

Safety inputs and internal safety conditions are the building blocks for safety control. Safety inputs receive binary signals (low, represented by "0", or high voltage, represented by "1"), from external sources such as a switch or a relay. Unlike general-purpose inputs that have no predefined function, each safety input is dedicated to specific function.

There are six different motor safety inputs and one system safety input. The controller provides a complete set of safety inputs for each motor. For example there are eight left limit inputs: one per motor.

The state of the motor safety inputs is stored in the ACSPL+ **SAFIN** variable, while the current state of the Emergency Stop input is stored in the ACSPL+ **S_SAFIN** variable. A high level of a physical signal (voltage) raises the corresponding bit and a low level drops the corresponding bit.

The safety inputs occupy the same bit numbers in **SAFIN** and **S_SAFIN** as the corresponding faults in **FAULT** and **S_FAULT**. Therefore, the same constants are used for bit addressing.

The physical signal connected to a safety input may indicate a safety violation with either high or low level. For instance on one axis, the right limit switch may indicate a safety violation with high voltage, and the left limit switch with low voltage. Use the ACSPL+ **SAFINI** and **S_SAFINI** variables to define which level is active, thereby eliminating the need for hardware inverters.

Table 11 Safety Inputs

Bit	Fault	Fault Category	Fault Description
0	#RL	Motor	RIGHT LIMIT SWITCH
1	#LL	Motor	LEFT LIMIT SWITCH
9	#DRIVE	Motor	DRIVE ALARM - alarm signal from a drive.
28	#ES	System	EMERGENCY STOP - alarm signal from the controlled plant.

6.2.3 Summary of Safety-Related Variables

The **FAULT**, **S_FAULT**, **SAFIN**, **S_SAFIN** variables are read-only (**SAFIN**, **S_SAFIN** can be assigned values, but these apply only when the Simulator is used). The **SAFINI**, **S_SAFINI**, **FMASK**, **S_FMASK**, **FDEF**, **S_FDEF** variables are protected and can be assigned only in protected mode (see **Section 6.2.6 - Application Protection**).

Table 12 Safety-Related Variables (page 1 of 2)

Name	Size	Access	Remarks
FAULT	8 (one per axis)	Read-only	MOTOR FAULTS. Each motor fault occupies one bit. Not all bits are occupied by faults. Only those bits that correspond to motor faults are meaningful.
FDEF	8 (one per axis)	Read-write (protected mode)	FAULT DEFAULT MASK. The variable bits control availability of the default responses to motor faults. The default value for all the bits, 1, enables the default response. If a bit is 0, the default response is disabled. Only those bits that correspond to motor faults are meaningful.
FMASK	8 (one per axis)	Read-write Protected	MOTOR FAULT MASK. The variable bits control whether the controller checks for motor faults. The default value 1 causes the controller to check for the fault associated with that bit. Only those bits that correspond to motor faults are meaningful.
S_FAULT	Scalar	Read-only	SYSTEM FAULTS. Each system fault and each aggregated motor fault occupies one bit. Only those bits that correspond to the faults are meaningful.

Table 12 Safety-Related Variables (page 2 of 2)

Name	Size	Access	Remarks
S_FDEF	Scalar	Read-write (protected mode)	SYSTEM FAULT DEFAULT MASK. The variable bits control availability of the default responses to system faults. The default value for all the bits, 1, enables the default response. If a bit is 0, the default response is disabled.
S_FMASK	Scalar	Read-write (protected mode)	SYSTEM FAULT MASK. The variable bits control whether the controller checks for system faults. The default value 1 causes the controller to check for the fault associated with that bit. Only those bits that correspond to system faults are used.
S_SAFIN	Scalar	Read-only (read/write for Simulator)	SYSTEM SAFETY INPUTS. Bit #ES reads the current state of the Emergency Stop input. Other bits are meaningless.
S_SAFINI	Scalar	Read-write (protected mode)	SYSTEM SAFETY INPUTS INVERSION. Bit **#ES** defines which value of **S_SAFIN.#ES** bit causes a fault. Other bits are not used.
SAFIN	8 (one per axis)	Read-only (read/write for Simulator)	MOTOR SAFETY INPUTS. Each meaningful bit reads the current value of a motor safety input. Only those bits that correspond to the motor safety inputs are meaningful.
SAFINI	8 (one per axis)	Read-write (protected mode)	MOTOR SAFETY INPUTS INVERSION. A bit of the variable defines which value of the corresponding **SAFIN** bit causes a fault. Only those bits that correspond to the meaningful **SAFIN** bits are used.

6.2.4 Integrity Control

Integrity Control validates the firmware and the user application stored in the controller. The following groups of files are stored in the internal file system of the nonvolatile memory:

- Firmware: files SB4.EXE, SB4.BIN, SBAUTO.BT
- Default configuration values: files PAR.### and PARn.### , where n = 0,1…
- Default ACSPL+ programs: files ACSPLnn.###, where nn = 00,01,02…
- Default SP programs: files DSP.### and/or DSPn.###, where n = 0,1…
- Saved configuration values: files PAR.$$$ and PARn.$$$, where n = 0,1…
- Saved ACSPL+ programs: files ACSPLnn.$$$, where nn = 00,01,02…
- Saved SP programs: files DSP.$$$ and/or DSPn.$$$, where n = 0,1…

Firmware and the default files present in the controller from the beginning and can be replaced only by the Version Changer of the SPiiPlus MMI.

The saved files compose the user application. Saved files are created or replaced by the memory management commands (See *SPiiPlus Command & Variable Reference Guide*).

Integrity Control is active for the all files specified above. The controller stores the size and checksum of each file, existing or created. The controller then compares the stored size/checksum with size/checksum of the actual file to expose damaged files. Validation is performed automatically on power-up. After power-up you can use the **IR** command to validate files (see **Section 6.2.4.2 - Integrity Report Command**).

6.2.4.1 Integrity Violation Fault

The bit of the Integrity Violation fault resides in the **S_FAULT** variable, and can be addressed as: **S_FAULT.#INTGR** or **S_FAULT.30**.

The fault has no default response. The masks **S_FMASK** and **S_FDEF** do not affect processing of the bit.

The controller automatically validates integrity on power-up before loading the user application. Therefore, you are able to define an **AUTOEXEC** program that checks the Integrity Violation fault and reports the error as required.

6.2.4.2 Integrity Report Command

The #**IR** Communication Terminal command activates integrity validation and provides a report of current integrity state.

If any integrity problem is detected, the command raises fault bit **S_FAULT.#INTGR**.

The report displays a list of files. Each list entry displays a file name, expected file size and checksum of the file and actual file size and checksum.

The following is an example of an integrity report:

```
#IR
                    Size                    Checksum
                    Registered  Actual      Registered  Actual
C:\
  sb1218pc.frm      001DA050    001DA050    DF4F97F0    DF4F97F0
  model.inf         000014C3    000014C3    C5CC6B93    C5CC6B93
  array.txt         00000010    00000010    00D4FF89    00D4FF89
  1.prg             000002CB    000002CB    D19BF636    D19BF636
  ECAT.XML          00052B0A    00052B0A    4D35D447    4D35D447
C:\SB4\DSP\
  dsp.###           0004FF0C    0004FF0C    BCBB37F5    BCBB37F5
  ADJ0.$$$          0000017D    0000017D    8E1A3690    8E1A3690
  dsp.##1           0003DC8E    0003DC8E    0B678F5D    0B678F5D
c:\sb4\startup\
  Acspl_e.$$$       0000010B    0000010B    6C441150    6C441150
  Par.$$$           000001E9    000001E9    704245FA    704245FA
  Par0.$$$          00000D7C    00000D7C    21F7589A    21F7589A
  Par1.$$$          00000D7C    00000D7C    471078B2    471078B2
  Par2.$$$          00000D7D    00000D7D    12FA61B8    12FA61B8
  Par3.$$$          00000D7C    00000D7C    9142BBDC    9142BBDC
  Par4.$$$          00000D7C    00000D7C    B65BDCF2    B65BDCF2
  Par5.$$$          00000D7C    00000D7C    DB74FE08    DB74FE08
  Par6.$$$          00000D7C    00000D7C    008E1F1E    008E1F1E
  Par7.$$$          00000D7C    00000D7C    25A74034    25A74034
  Par8.$$$          00000D7C    00000D7C    4BC06150    4BC06150
  Par9.$$$          00000D7C    00000D7C    70D98266    70D98266
  Par10.$$$         00000DF1    00000DF1    8142E4DE    8142E4DE
  Par11.$$$         00000DF1    00000DF1    A561FBF9    A561FBF9
  Par12.$$$         00000DF1    00000DF1    C3801414    C3801414
  Par13.$$$         00000DF1    00000DF1    E79F2A2F    E79F2A2F
  Par14.$$$         00000DF1    00000DF1    0BBE414A    0BBE414A
  Par15.$$$         00000DF1    00000DF1    2FDD5865    2FDD5865
  Par16.$$$         00000DF1    00000DF1    53FC6F80    53FC6F80
  Par17.$$$         00000DF3    00000DF3    2D5F0297    2D5F0297
  Par18.$$$         00000DF1    00000DF1    9C3A9DB6    9C3A9DB6
  Par19.$$$         00000DF1    00000DF1    C059B4D1    C059B4D1
  Par20.$$$         00000DF1    00000DF1    966603F5    966603F5
  Par21.$$$         00000DF1    00000DF1    BA851B10    BA851B10
  Par22.$$$         00000DF1    00000DF1    DEA4322B    DEA4322B
  Par23.$$$         00000DF1    00000DF1    02C34946    02C34946
  Par24.$$$         00000DF1    00000DF1    26E26061    26E26061
  Par25.$$$         00000DF1    00000DF1    4B01777C    4B01777C
  Par26.$$$         00000DF1    00000DF1    6F208E97    6F208E97
  Par27.$$$         00000DF1    00000DF1    933FA5B2    933FA5B2
  Par28.$$$         00000DF1    00000DF1    B75EBCCD    B75EBCCD
  Par29.$$$         00000DF1    00000DF1    DB7DD3E8    DB7DD3E8
  Par30.$$$         00000DF1    00000DF1    B18A230C    B18A230C
  Par31.$$$         00000DF1    00000DF1    D5A93A27    D5A93A27
  Acspl01.$$$       00004E27    00004E27    6F167A54    6F167A54
  Acspl02.$$$       00004E27    00004E27    6F167A54    6F167A54
  Acspl03.$$$       00004E27    00004E27    6F167A54    6F167A54
  Acspl04.$$$       00004E27    00004E27    6F167A54    6F167A54
  Acspl05.$$$       00004E27    00004E27    6F167A54    6F167A54
  Acspl06.$$$       00004E27    00004E27    6F167A54    6F167A54
  Acspl07.$$$       00004E27    00004E27    6F167A54    6F167A54
  Acspl08.$$$       00004E27    00004E27    6F167A54    6F167A54
  Acspl00.$$$       00004E27    00004E27    6F167A54    6F167A54
  Acspl09.$$$       00004E27    00004E27    6F167A54    6F167A54
  Acspl10.$$$       00004E27    00004E27    6F167A54    6F167A54
  Acspl11.$$$       00004E27    00004E27    6F167A54    6F167A54
  Acspl12.$$$       00004E27    00004E27    6F167A54    6F167A54
  Acspl13.$$$       00004E27    00004E27    6F167A54    6F167A54
  Acspl14.$$$       00004E27    00004E27    6F167A54    6F167A54
  Acspl15.$$$       00004E27    00004E27    6F167A54    6F167A54
C:\sb4\user\
  oleg              00000024    00000024    4144536B    4144536B
  I                 00000330    00000330    011453BC    011453BC
  V                 00002EF0    00002EF0    414453A7    414453A7
System Integrity is OK
:
```

6.2.5 Report of Real-Time Usage Command

The **#U** Terminal command provides a report of Real Time Usage. The controller continuously measures the time taken by real-time tasks. For details on real-time tasks see **Section 2.1.3 - realtime and Background Tasks**.

When the **#U** command is received, the controller analyzes the measured times during the last 50 controller cycles and calculates minimal, maximal and average time. The results are reported in percentages.

Note	
	*You can also use the ACSPL+* **USAGE** *variable (see SPiiPlus Command & Variable Reference Guide) to monitor the usage. This variable is particularly useful in autoroutines for halting a program if the MPU usage is excessive.*

6.2.6 Application Protection

The following Terminal commands switch between protected mode and configuration mode:

- **#PROTECT**

  The **#PROTECT** command applies application protection (puts the controller in protected mode). The command can be followed by a password, where the password can be any sequence of ASCII characters enclosed in quotes (“”). The sequence can contain any printable or non-printable character, except the quotes, for example:

```
#PROTECT “123MyPassword”
```

- **#UNPROTECT**

  The **#UNPROTECT** command disables application protection (returns the controller to configuration mode).

Note	
	*If* **#PROTECT** *was applied with a password, the same password must accompany the corresponding* **#UNPROTECT** *command. If a password was not included in the* **#PROTECT** *command, the* **#UNPROTECT** *command does not need a password.*

If the controller is in protected mode, **#RESET** can be applied to delete a password-protected application if the password is unknown. In this case, the application cannot be viewed or saved, but the controller will be reset to the factory-defaults.

Once you have put the application in the Protected mode, you have the option of locking the value of any ACSPL+ standard read-write variable by applying the **SETPROTECTION** command. This command converts the specified variable to read-only and the variable's value that you set cannot be changed, for example:

```
SETPROTECTION VEL = 1!The VEL variable will retain the value of 1
       !throughout the life of the application.
```

For a description of the application protection commands see *SPiiPlus Command & Variable Reference Guide*.

6.2.7 Report Safety Configuration

The #**SC** command reports the current safety system configuration.

The controller response includes the following:

- ❑ active safety groups
- ❑ the configuration of each fault for each motor

For example:

```
#SC
Bit  Code     Fault                  0    1    2    3    4    5    6    7
0    #RL    Right Limit              K    K    K    K    K    K    K    K
1    #LL    Left Limit               K    K    K    K    K    K    K    K
2    #NT Network error               D    D    D    D    D    D    -    -
3    #FAN Cooling Fan Fault          -    -    -    -    -    -    -    -
4    #HOT Overheat                   -    -    -    -    -    -    -    -
5    #SRL Software Right Limit       K    K    K    K    K    K    K    K
6    #SLL Software Left Limit        K    K    K    K    K    K    K    K
7    #ENCNC Encoder Not Connected    D    D    D    D    D    D    D    D
8    #ENC2NC Encoder2 Not Connected  -    -    -    -    -    -    -    -
9    #DRIVE Drive Alarm              KD   KD   KD   KD   KD   KD   KD   KD
10   #ENC   Encoder Error            D    D    D    D    D    D    D    D
11   #ENC2  Encoder 2 Error          -    -    -    -    -    -    -    -
12   #PE    Position Error
13   #CPE   Critical Position Error  KD   KD   KD   KD   KD   KD   KD   KD
14   #VL    Velocity Limit           K    K    K    K    K    K    K    K
15   #AL    Acceleration Limit       K    K    K    K    K    K    K    K
16   #CL    Overcurrent              KD   KD   KD   KD   KD   KD   KD   KD
17   #SP    Servo Processor Alarm    D    D    D    D    D    D    D    D
25   #PROG  Program Error            K    K    K    K    K    K    K    K
26   #MEM   Memory Overuse
27   #TIME  Time Overuse
28   #ES    Emergency Stop           KD   KD   KD   KD   KD   KD   KD   KD
29   #INT   Servo Interrupt          D    D    D    D    D    D    D    D
30   #INTGR Integrity Violation      D    D    D    D    D    D    D    D
31   #FAILURE Component Failure      D    D    D    D    D    D    D    D
```

The following designations are used in the report:

- ❑ --- – fault detection is disabled (**FMASK=0**)
- ❑ (blank) – fault response is disabled (**FDEF=0**) or no default response is defined
- ❑ K – response is kill
- ❑ D – response is disable
- ❑ KD – response is kill-disable
- ❑ + – generalized fault

6.3 Working with Faults

6.3.1 Addressing the Fault Bits

Faults are represented as bits in the ACSPL+ variables **FAULT** and **S_FAULT**.

FAULT is an integer array containing eight elements (corresponding to the number of motors), where each element is made up of a set of bits. Each bit indicates one motor fault. Motor faults are related to a specific motor, power amplifier, or Servo Processor. Examples include Tracking Error, and Motor Overheat.

To address a specific motor fault bit, start with the specification of the **FAULT** element, followed by the bit selection operator (dot) and then the corresponding fault designator.

For example:

`FAULT(2).#LL`	Addresses the left limit fault bit of axis 2. The bit is raised if the 2 left limit switch is activated.
`FAULT(3).#DRIVE`	Addresses the drive fault bit of axis 3. The bit is raised if the 3 drive safety input is active.

S_FAULT is a scalar variable with two categories of bits:

- Aggregated motor faults. Once the controller raises a bit in any element of **FAULT**, it immediately raises the corresponding bit of **S_FAULT**. Therefore, each bit of **S_FAULT** is an OR aggregate of the corresponding bits in all elements of **FAULT**.
- System faults that are not related to any specific motor, such as Emergency Stop and Time Overuse.

The aggregated motor fault bits occupy the same bit positions as the corresponding motor fault bits in the **FAULT** variable. Use the designators of the motor faults to address the aggregated motor fault bits.

Examples:

`S_FAULT.#LL`	Addresses the aggregated Left Limit fault bit. The bit is raised if the Left Limit switch of any motor is activated.
`S_FAULT.#DRIVE`	Addresses aggregated Drive fault bit. The bit is raised if the Drive safety input of any motor is active.

Use the bit designators of the system faults to address the system fault bits.

Examples:

`S_FAULT.#ES`	Addresses the Emergency Stop fault bit. The bit is raised when the Emergency Stop safety signal is active.
`S_FAULT.#PROG`	Addresses the Program fault bit. The bit is raised when any program has failed due to a run-time error.

6.3.2 Querying Faults

The variables **FAULT** and **S_FAULT** are queried like any other variable. The controller reports the status of each meaningful bit.

Example:

```
?S_FAULT
  0 OFF Right Limit (#RL)
  1 ON Left Limit (#LL)
  2 OFF Network error (#NT)
  3 OFF Cooling Fan Fault (#FAN)
  4 OFF Overheat (#HOT)
  5 OFF Software Right Limit (#SRL)
  6 OFF Software Left Limit (#SLL)
  7 OFF Encoder Not Connected (#ENCNC)
  8 OFF Encoder 2 Not Connected (#ENC2NC)
  9 OFF Driver Alarm (#DRIVE)
 10 OFF Encoder Error (#ENC)
 11 OFF Encoder 2 Error (#ENC2)
 12 OFF Position Error (#PE)
 13 OFF Critical Position Error (#CPE)
 14 OFF Velocity Limit (#VL)
 15 OFF Acceleration Limit (#AL)
 16 OFF Overcurrent (#CL)
 17 OFF Servo Processor Alarm (#SP)
 20 OFF HSSI Not Connected (#HSSINC)
 25 OFF Program Error (#PROG)
 26 OFF Memory Overuse (#MEM)
 27 OFF Time Overuse (#TIME)
 28 OFF Emergency Stop (#ES)
 29 OFF Servo Interrupt (#INT)
 30 OFF Integrity Violation (#INTGR)
 31 OFF Component Failure (#FAILURE)
```

The number in the left column is the bit number, followed by an ON/OFF indicator and the fault description and the bit name in parentheses. In the above example all the faults are OFF except for the Left Limit fault of one or more axes.

Note that the **S_FAULT** variable indicates that there is a motor fault, but does not specify which motor has failed. To determine which motor has failed, query the **FAULT** variable, or use **?$** to query the state of all motors.

Fault bits can be queried individually:

```
?S_FAULT.#LL
1
?FAULT(0).#LL, FAULT1.#LL
0
1
```

The controller answers a query of an individual bit by showing the numerical value of the bit: either 0 or 1.

6.3.3 Using the Fault Bits in if, while, till Commands

Using the fault variables in the condition of commands **if**, **while**, or **till** provides a decision making mechanism that is based on the present state of the faults.

Examples:

`if FAULT(0).#HOT`
`  OUT(0).6 = 1`
`end`
	Activate an output (**OUT.6**) if X motor is overheated

`if FAULT(0).#LL | FAULT(0).#RL`
`  disp "X limit switch"`
`end`
	Display a warning if any limit switch of X motor is active

`till ^FAULT(0).#LL`	Wait until the X Left limit switch is released
`if S_FAULT disp "Failure";end`	Display a warning if any fault is active

The condition **if S_FAULT** is satisfied if **S_FAULT** is non-zero, i.e., if any bit of **S_FAULT** is raised. Any fault, either system or motor, raises a bit in **S_FAULT**. Therefore a non-zero **S_FAULT** indicates that one or more faults are active.

The variables **FAULT** and **S_FAULT** display the current state of the faults. A conditional command based on these variables uses the fault state at the instant when the command is executed. For example, if the X left limit was activated but then released, **FAULT0.#LL** is zero, and the command if **FAULT0.#LL** considers the condition unsatisfied.

6.3.4 Creating Fault-Processing Autoroutines

To create an autoroutine that processes a fault, specify the fault bit in the autoroutine condition.

Example:

`on FAULT(0).#LL`	Start the autoroutine when an X Left Limit fault occurs

A fault-processing autoroutine can reside in any program buffer. When the buffer is compiled, the controller checks the autoroutine condition each controller cycle. When the condition is satisfied, the controller interrupts the program that is currently executing in the buffer that the autoroutine resides in, and starts the autoroutine execution.

A fault-processing autoroutine can supplement or replace the default response to a fault. If the corresponding **FDEF** or **S_FDEF** bit enables the default response, the autoroutine starts and executes in parallel with the default response. If the corresponding **FDEF** or **S_FDEF** bit is zero, the default response is disabled, and the autoroutine is the only controller response.

Note	*Programming Note* *The controller examines all autoroutine conditions each controller cycle. However, if an autoroutine is executing in a buffer, and a condition of the second autoroutine in the same buffer is satisfied, the second autoroutine will start only after termination of the subroutine currently executing. Therefore, if an application includes a time-consuming autoroutine, avoid placing safety autoroutines that require short response times in the same buffer with the time-consuming autoroutine.*

Examples:

The following autoroutine displays a message when the Drive Alarm signal becomes active for the 0 axis motor:

```
on FAULT(0).#DRIVE
   disp "Axis 0 Drive Alarm"
ret
```

In the following autoroutines, the 0 and 2 axes motors must be disabled simultaneously. Therefore, if one of the drives fails, the second must be disabled as well. The default response disables the 0 axis motor if the 0 Drive Alarm occurs and disables the 2 axis motor if the 2 Drive Alarm occurs. The following pair of autoroutines supplements the default response by disabling a motor if the other motor fails:

```
on FAULT(0).#DRIVE disable 2; ret
on FAULT(2).#DRIVE disable 0; ret
```

When an X drive fault occurs, the following autoroutine terminates the controller activity for all motors:

`on FAULT(0).#DRIVE`	When 0 axis drive fault occurs
`disableall`	Disable all motors
`stopall`	Stop all other programs
`stop`	Stop the current program
`ret`	End of autoroutine

The **S_FAULT** variable contains the bits of the aggregated motor faults. These bits provide a convenient alternative to the motor faults if an application requires common processing of a motor fault irrespective of which motor caused the fault.

For example, the following autoroutine displays a message when the Left Limit switch of any motor is activated:

```
on S_FAULT.#LL
       disp "One of the Left Limit Switches is Activated"
ret
```

Autoroutine conditions can contain more than one fault bit, as is shown in the first line of the example below:

```
on S_FAULT.#LL | S_FAULT.#RL
        disp "Some Limit Switch Activated"
ret
```

The **S_FAULT** variable (used without a bit extension) indicates whether a fault has been detected by the controller. The following example shows an autoroutine that provides an alarm message if any fault occurs in the controller:

```
on S_FAULT
        disp "Something happened"
ret
```

The controller activates an autoroutine when the condition of the autoroutine changes from false to true. If the condition remains true, the autoroutine is not activated again until the condition becomes false, and then true again. Therefore the above autoroutine displays the alarm message only on the first fault. If one fault bit is already raised, and another fault occurs, the second fault does not generate an alarm message.

The following autoroutine displays a fault message each time a fault occurs:

```
int  LastFault
on  LastFault <> S_FAULT
        if  (LastFault ~ S_FAULT) & S_FAULT
               disp "Something happened"
        end
        LastFault = S_FAULT
ret
```

In the above example the local variable **LastFault** stores the current value of **S_FAULT**. The exclusive OR (~) of **LastFault** and **S_FAULT** detects the bits of **S_FAULT** that changed. The AND (&) with **S_FAULT** retains only the bits that changed from zero to one, and not from one to zero.

6.3.5 Disabling Fault Processing

Warning 	***Certain safety variables provide protection against potential serious bodily injury and damage to equipment. Be aware of the implications before disabling any alarm, limit or error.***

The ACSPL+ variables define which faults are examined and processed. If a bit of **FMASK** or **S_FMASK** is zero, the corresponding fault is disabled and the bit of **FAULT** or **S_FAULT** is not raised.

FMASK and **S_FMASK** are queried like any other variable, and the controller reports the status of each meaningful bit.

Example:

```
?FMASK(0)
  0 ON Right Limit (#RL)
  1 ON Left Limit (#LL)
  2 ON Network error (#NT)
  3 ON Cooling Fan Fault (#FAN)
  4 ON Overheat (#HOT)
  5 ON Software Right Limit (#SRL)
  6 ON Software Left Limit (#SLL)
  7 ON Encoder Not Connected (#ENCNC)
  8 ON Encoder 2 Not Connected (#ENC2NC)
  9 ON Driver Alarm (#DRIVE)
 10 OFF Encoder Error (#ENC)
 11 OFF Encoder 2 Error (#ENC2)
 12 ON  Position Error (#PE)
 13 ON  Critical Position Error (#CPE)
 14 ON Velocity Limit (#VL)
 15 ON Acceleration Limit (#AL)
 16 OFF Overcurrent (#CL)
 17 OFF Servo Processor Alarm (#SP)
 20 OFF HSSI Not Connected (#HSSINC)
```

```
?S_FMASK
 25 ON Program Error (#PROG)
 26 OFF Memory Overuse (#MEM)
 27 OFF Time Overuse (#TIME)
 28 ON Emergency Stop (#ES)
 29 OFF Servo Interrupt (#INT)
 30 OFF Integrity Violation (#INTGR)
 31 OFF Component Failure (#FAILURE)
```

Normally, you enable or disable fault detection through the Adjuster wizard of the SPiiPlus MMI Application Studio (see the *SPiiPlus MMI Application Studio User Guide*) when initially configuring the controller. The configured values of **FMASK** and **S_FMASK** are then stored in the flash memory and left unchanged during the application lifetime.

Changes to safety variables after initial controller configuration may affect your application. The following section is relevant only if you need to enable or disable faults after initial configuration.

Example:

`?FAULT(0).#DRIVE , SAFIN(0).#DRIVE`	Display the status of the 0 drive alarm fault, and the safety signal
1	Drive Alarm fault bit is set.
1	Drive Alarm safety signal is set.
`FMASK(0).#DRIVE = 0`	Disable 0 axis Drive Alarm fault

`?FAULT(0).#DRIVE , SAFIN(0).#DRIVE`	Display the status of the 0 axis Drive Alarm fault, and the safety signal
0	Drive Alarm fault bit is reset.
1	Drive Alarm safety signal is still set.

6.3.6 Defining the Active Level of Safety Input

Safety inputs receive physical signals from various sources, such as limit switches and relays. Each safety signal is sent in one of two forms (binary):

- High voltage level, no current in the controller input circuit.
- Low voltage level, outflowing current in the controller-input circuit.

By default, the high voltage level is defined as the active state of the signal, i.e., the state that triggers a fault. This is called the normal polarity.

By using the ACSPL+ variables **SAFINI** and **S_SAFINI**, which define what level is active for each safety input, you can change the default defining the low voltage level as active for a specific safety input. The low voltage level will now trigger a fault. This is called inverse polarity. If a bit of **SAFINI** or **S_SAFINI** is zero (default value), the corresponding input accepts the high level as active.

Note that the bits of **SAFIN** and **S_SAFIN** reflect the physical states of the signals, while the bits of **SAFINI** and **S_SAFINI** define the logical processing of the signals. **SAFINI** and **S_SAFINI** variables have no effect on the physical signal, and the bits of variables **SAFIN** and **S_SAFIN**, which display the raw values of the safety inputs are unaffected by the bits of **SAFINI**, **S_SAFINI**

The variables **SAFINI** and **S_SAFINI** are queried like any other variable. The controller reports the status of each meaningful bit that corresponds to a safety signal.

Example:

```
?SAFINI(0)
  0 ON Right Limit (#RL)
  1 ON Left Limit (#LL)
  4 OFF Overheat (#HOT)
  9 OFF Driver Alarm (#DRIVE)

?S_SAFINI
 28 OFF Emergency Stop (#ES)
 31 OFF Component Failure (#FAILURE)
```

In the above example, the fact that the response to the **SAFINI(0)** query shows that **RL** and **LL** are ON (bits 0 and 1) indicates that you have defined inverse polarity (low active level) for signals **#RL**, **#LL** of the 0 axis.

Normally, you define the signal polarity through the Adjuster wizard of the SPiiPlus MMI Application Studio (see the *SPiiPlus MMI Application Studio User Guide*) when initially configuring the controller. The configured values of **SAFINI** and **S_SAFINI** are then stored in the flash memory and are not changed during the application's lifetime.

Example:

`?FAULT(0).#LL , SAFIN(0).#LL`	Display the status of Left Limit and the Left Limit Safety signal for the 0 axis.
1 **1**	Left Limit fault bit is raised. Left Limit safety signal is high.
`SAFINI(0).#LL = 1`	Set inverse polarity
`?FAULT(0).#LL , SAFIN(0).#LL`	Display the status of Left Limit and the Left Limit Safety signal for the 0 axis.
0 **1**	Left Limit fault bit changes to zero. Left Limit safety signal remains high.

6.3.7 Fault Processing Modes

The controller defines two modes of behavior after a failure: regular and strict. Bit **#FCLEAR** of the **S_FLAGS** variable selects the fault processing mode:

- If **S_FLAGS.#FCLEAR = 0** (default) the controller is in regular mode
- If **S_FLAGS.#FCLEAR = 1** the controller is in strict mode.

The difference between the modes manifests when a fault occurs that kills a motor:

- In the regular mode the next motion command simply clears the reason for the previous kill for all involved motors and starts the new motion.
- In the strict mode the next motion command cannot activate the motion and fails. The motion cannot be activated as long as the reason for the previous kill is non-zero for any involved motor.

The reason for a kill operation is stored in the **MERR** variable. In the strict mode as long as a **MERR** element is non-zero, the corresponding motor cannot be put in motion. Commands **enable** and **fclear** clear the **MERR** elements for the specified motors and enable the next motion.

The same rules apply to the results of a **kill** command with non-zero second argument (the reason for the kill - see **Section 4.1.3 - kill and killall Commands**). The reason is stored in the **MERR** element and in the strict mode the next motion cannot be activated until the reason is cleared.

In the regular mode the behavior is simple and totally compatible with previous versions. However, you may prefer the strict mode, especially during application development. The following example gives a hint why the strict mode may be preferable:

```
Reciprocated:
ptp/r 0,10000
ptp/r 0,-10000
goto Reciprocated
```

Under normal conditions the motor continuously moves forward and backward by 10,000 units. Assume, however, that the first motion brings the motor to the right limit switch. The first motion terminates prematurely, because the motor is killed. However, the program continues running and executes the second motion command. In the regular mode the second motion starts successfully because it is directed out of the limit. Then the first motion command again

brings the motor to the limit. Therefore, in the regular mode the reciprocated motion continues and there is no clear indication of abnormal condition.

Assume further, for the same application, that a broken connection to the right limit switch causes the controller to mistakenly continuously detect that the right limit has been passed. The first motion fails immediately after start, but the second one executes. The result is that the motors move in a negative direction by steps of 10,000 units.

In the strict mode, the behavior is more predictable. After the first motion failed, the second one cannot start and the program itself terminates with error. You can check the information in **MERR** and **PERR** to disclose the reason for the failure.

If at any point of the application a fault is an expected condition and the program must continue, the program in the strict mode must analyze the **MERR** element and execute the **fclear** command before activating the next motion.

6.4 Network Faults

There are 3 types of possible faults:

- Initialization failure – the EtherCAT stack could not start properly
- Network failure during the normal work – TI software keeps running
- TI SW in a slave does not run properly or in reset state

6.4.1 Axis Network-Related Faults

Each axis has two network-related faults:

- **FAULTx.2** - Network Fault
- **FAULTx.17** - Servo Processor Alarm

The possible reasons for Network Fault are:

- Reset of EtherCAT slave chip
- Physical Ethernet line disconnection
- Power Down of a single slave

The Servo Processor Alarm is set whenever the handshake counter between the master and the slave is different from CTIME*20, i.e., the DSP worked exactly CTIME*20 times between two consequent MPU cycles.

The possible reasons for Servo Processor Alarm are:

- Communication loss with DSP for any reason
- DSP SW failure (Network might keep working OK)
- DSP SW over usage
- Loss of synchronization between MPU and DSP
- TI reset by Watch Dog or Power down

Both faults have a default response of disabling the axes that are affected. If there is a Network Fault on an axis, it will be always followed by Servo Processor Alarm. Malfunction of a single node will raise the Network Fault bit in all axes, in order to give the ability of immediate reaction.

6.4.2 Initialization Failure

The stack may fail to start up for several reasons. The ACSPL+ **ECST** variable shows different stages of stack initialization. As long as the stack has not reached full network initialization followed by successful load of DSP program, all axes of the corresponding DSP will be in constant Servo Processor Alarm.

In this case, **ECST** and **ECERR** will show the cause of the failure.

6.4.3 Network Failure During Operation

In the event of a loss of communication, for example, due to broken cable, power down, poor contact, etc., after successful initialization, the Firmware will analyse which nodes are out of order and will activate the Network Fault on related axes. If the DSP program is valid and running (for example in case of cable out), the DSP recognizes that the master is not controlling the bus and disables all its axes; in addition the handshake **SYNC** counter is frozen.

After successful reconnection, the DSP sees that the master has sent a recent **SYNC** value and returns to normal functioning. **FCLEAR** or **ENABLE** will reset the Network Fault and Servo Processor Alarm and allow normal axis operation.

6.4.4 DSP Software Failure

In case of node reset due to watchdog or power down or in case of an unpredictable DSP malfunction, the Servo Processor Alarm will always be activated on the related axes. In most cases it will also cause Network Fault. There is no way to clear this fault, because the DSP does not have a valid program and it is not properly synchronized with the MPU.

To overcome this, run **#HWRES** (Reboot controller) in the Communication Terminal, or perform a complete power down, and then power up to reset this node.

Note	*Other nodes may keep functioning, if their response to Network Fault is masked.*

6.5 Detailed Description of Faults

This section provides a detailed description of each fault, including a description of the bits involved, the default response, and examples of autoroutines.

6.5.1 Limit Switches: #LL, #RL

The exact usage of limit switches depends on the application. A specific axis may require only one pair of limit switches or no limit switches at all. The following diagram illustrates a typical use of two pairs of limit switches:

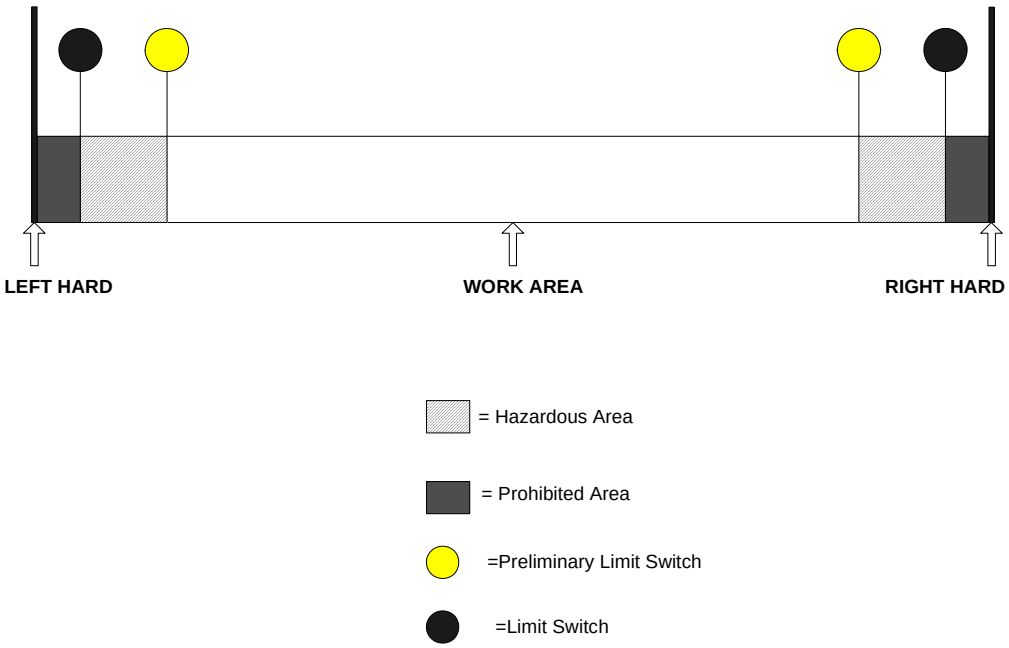


Figure 14 The Use of Limit Switches

Fault bits	**FAULT.#LL**, **FAULT.#RL** (in each element of **FAULT**)
Mask bits	**FMASK.#LL**, **FMASK.#RL** (in each element of **FMASK**)
Based upon safety signals	**SAFIN.#LL**, **SAFIN.#RL** (in each element of **SAFIN**)
Inversion bits	**SAFINI.#LL**, **SAFINI.#RL** (in each element of **SAFINI**)
Internal safety condition	None
Default response bits	**FDEF.#LL**, **FDEF.#RL** (in each element of **FDEF**)
Default response	The controller kills the violating motor. As long as the fault is active, the controller kills any motion that tries to move the motor in the direction of the limit. Motion to return to the allowed range of motion is allowed.

Autoroutine examples:

The first example supplements the default processing of X limit faults with alarm messages:

`on FAULT(0).#LL`	When a Left Limit fault occurs in the 0 axis motor.
`disp "0 Left Limit switch activated"`	Display the message: 0 Left Limit switch activated.
`ret`	
`on FAULT(0).#RL`	When a right limit fault occurs in 0 axis motor.
`disp "0 Right Limit switch activated"`	Display the message: 0 Right Limit switch activated.
`ret`	

The example below implements an autoroutine that disables the motor rather than the default response of killing the motion in case of a right limit or left limit fault. This response may be superior to the default response if the motor is equipped with a brake that is activated by the disable command because the brake may stop the motor faster than a kill command.

`on FAULT(2).#RL | FAULT(2).#LL.`	When there is a right limit or left limit fault in the 2 axis motor.
`disable 2`	Disable motor 2
`ret`	

6.5.2 Network Fault: #NT

Fault bits	**FAULT.#NT**
Mask bits	**FMASK.#NT**
Based upon safety signals	None
Inversion bits	
Internal safety condition	
Default response bits	None
Default response	Disables the axis

6.5.3 Cooling Fan Fault: #FAN

Fault bits	**FAULT.#FAN**
Mask bits	**FMASK.#FAN**
Based upon safety signals	None
Inversion bits	
Internal safety condition	
Default response bits	None
Default response	Disables the axis

6.5.4 Software Limit Switches: #SLL, #SRL

Fault bits	**FAULT.#SLL**, **FAULT.#SRL** (in each element of **FAULT**)
Mask bits	**FMASK.#SLL**, **FMASK.#SRL** (in each element of **FMASK**)
Based upon safety signals	None
Inversion bits	None
Internal safety condition	See explanation below.
Default response bits	**FDEF.#SLL**, **FDEF.#SRL** (in each element of **FDEF**)
Default response	The controller kills the violating motor. As long as the fault is active, the controller kills any motion that tries to move the motor in the direction of the limit. Motion in the direction out of the limit is allowed.

Software limit switches use the following ACSPL+ variables:

- **SLLIMIT** –Software Left Limit (Lower limit of working area)
- **SRLIMIT** –Software Right Limit (Upper limit of working area)

The condition for software limit switches is based on the motor reference **RPOS** variable, not the motor feedback **FPOS** variable. Therefore, the fault provides protection against errors in the ACSPL+ application, not against hardware malfunctions.

The controller monitors the reference position **RPOS** and reference velocity **RVEL** and implements the following verifications:

- If **RPOS** < **SLLIMIT**, the controller detects **#SLL** fault.
- If **RPOS** > **SRLIMIT**, the controller detects **#SRL** fault.
- If **RPOS** is within the range and **RVEL** is non-zero, the controller calculates the distance required to decelerate **RVEL** to zero using **KDEC** deceleration. If the final point of the calculated deceleration process is < **SLLIMIT**, the controller detects **#SLL** fault. If the final point of the calculated deceleration process is > **SRLIMIT**, the controller detects a **#SRL** fault.

This logic provides the moving edge of the software limit fault, depending on the instant velocity. As the controller kills the motor when the fault is detected, the termination point of the kill process will be very close to the corresponding software limit point.

The termination point is not exactly the software limit point because the controller checks the condition every controller cycle, i.e., at discrete time points. The termination point complies with the following conditions:

- The termination point lies beyond the corresponding software limit.
- Overrun is not more than **2*Vel*Cycle**,

  where **Vel** is an instant velocity and **Cycle** is the controller cycle.

For example, if the **FAULT(0).#SRL** fault is detected, the requested velocity of 0 axis is 10,000 count/sec and the controller cycle is 1 msec. The controller will overrun the software right limit for not more than 2*10000*0.001 = 20 counts.

Autoroutine examples:

The following autoroutines supplement the default processing of X software limit faults with an alarm messages:

```
on FAULT(0).#SLL
       disp "0 Software Left Limit violated"
ret



on FAULT(0).#SRL
       disp "0 Software Right Limit violated"
ret
```

6.5.5 Non-Critical Position Error: #PE

Fault bits	**FAULT.#PE** (in each element of **FAULT**)
Mask bits	**FMASK.#PE** (in each element of **FMASK**)
Based upon safety signals	None
Inversion bits	None
Internal safety condition	See explanation below.
Default response bits	**FDEF.#PE** (in each element of **FDEF**)
Default response	None

Use the **#PE** fault to detect non-critical violation of position accuracy, and the **#CPE** (see **Section 6.5.6 - Critical Position Error: #CPE**) fault to detect uncontrolled, excessive error that indicates loss of control.

The following ACSPL+ variables are associated with position errors:

- **ERRI** – Maximum position error while the motor is idle (not moving)
- **ERRV** – Maximum position error while the motor is moving with constant velocity
- **ERRA** – Maximum position error while the motor is accelerating or decelerating
- **DELI** – Delay on transition from ERRA to ERRI
- **DELV** – Delay on transition from ERRA to ERRV

The controller raises the **FAULT.#PE** bit if the position error exceeds the maximum specified value, which is equal to **ERRI**, **ERRV** or **ERRA** depending on the motion state.

The variables **DELI** and **DELV** are used in a similar manner with the **#CPE** fault.

The following diagram illustrates the use of these variables for a typical motion profile that includes acceleration, constant velocity and deceleration:

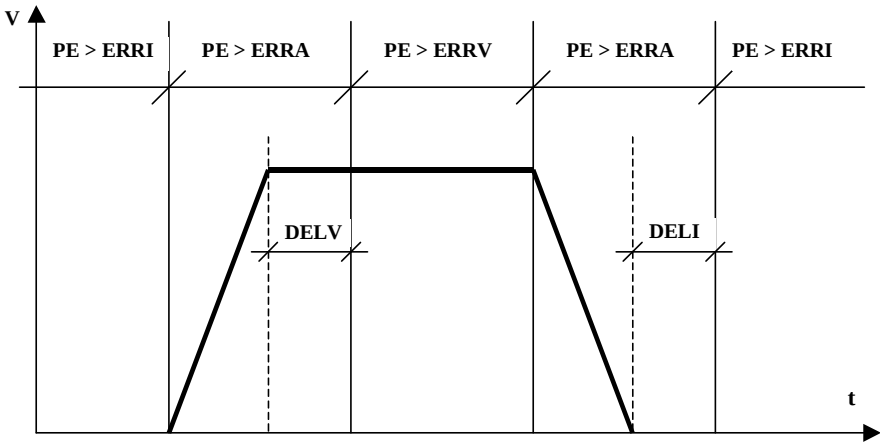


Figure 15 Use of Variables in a Typical Motion Profile

The allowed position error limit is:

- **ERRI** if the motor is idle
- **ERRV** if the motor is moving with constant velocity
- **ERRA** if the motor is accelerating or decelerating
- **DELV** defines delay on transition from **ERRA** to **ERRV**.
- **DELI** defines delay on transition from **ERRA** to **ERRI**.

Autoroutine examples:

The following autoroutine supplements the default response to a position error with an alarm message.

```
on FAULT(1).#PE
      disp "Accuracy violation - the motion was killed"
ret
```

The next example corrects the motion conditions by reducing the velocity (**VEL1**) until the error returns to within limits, instead of killing the motion.

`on FAULT(1).#PE`	When there is a position error fault in the 1 axis motor.
`while FAULT(1).#PE`	As long as there is a position error.
`imm VEL(1) = 0.9 * VEL(1)`	Reduce the velocity of the 1 axis motor by 10%.
`wait 10`	Delay.
`end`	
`ret`	

The controller automatically provides a smooth transition to the new velocity.

An application that incorporates the above autoroutine must satisfy the following conditions:

- All motions of the 1 axis are single-axis, or 1 is a leading axis in a group. If another axis is leading, all motions will use the velocity of that axis, and the command **VEL(1) = …** will have no effect.
- All motions of the 1 axis use the default velocity **VEL(1)** and do not specify individual velocity.
- The specific error monitored is the position error only while the motor is moving with constant velocity. To avoid the fault while the motor is idle or moves with acceleration, you have to initialize the variables **ERRI(1)** and **ERRA(1)** to sufficiently large values.

Note	*Programming Note:* *The above autoroutine executes an undefined number of loops with delays in each loop. Therefore, the execution time may be significant. As long as the autoroutine executes, no other autoroutine in the same buffer can be activated. Do not place this autoroutine in the same buffer that contains any time-critical autoroutine.*

6.5.6 Critical Position Error: #CPE

Fault bits	**FAULT.#CPE** (in each element of **FAULT**)
Mask bits	**FMASK.#CPE** (in each element of **FMASK**)
Based upon safety signals	None
Inversion bits	None
Internal safety condition	See explanation below.
Default response bits	**FDEF.#CPE** (in each element of **FDEF**)
Default response	The controller disables the violating motor.

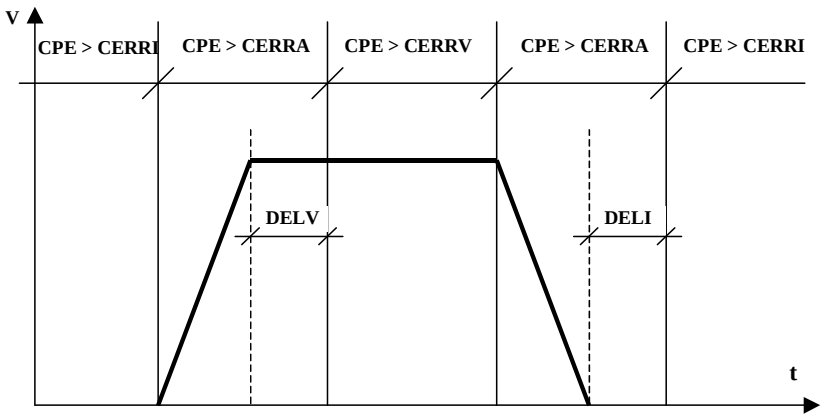
Use **#PE** fault (see **Section 6.5.5 - Non-Critical Position Error: #PE**) to detect non-critical violations of position accuracy, and the **#CPE** fault to detect uncontrolled, excessive error that indicates loss of control. **#CPE** should be greater than **#PE**.

The following ACSPL+ variables are associated with critical position error :

- **CERRI** - Critical position error if the motor is idle (not moving)
- **CERRV** - Critical position error if the motor is moving with constant velocity
- **CERRA** - Critical position error if the motor is accelerating or decelerating
- **DELI** - Delay on transition from **CERRA** to **CERRI**
- **DELV** - Delay on transition from **CERRA** to **CERRV**

The variables **DELI** and **DELV** are used also in the condition for the **#PE** fault.

The controller raises the fault bit if the position error exceeds the critical value. The critical value is equal to **CERRI**, **CERRV** or **CERRA** depending on the motion stage. The following diagram illustrates the use of these variables for a typical motion profile that includes acceleration, constant velocity and deceleration:



The Critical limit for position error is:

- **CERRI** if the motor is idle
- **CERRV** if the motor is moving with constant velocity
- **CERRA** if the motor is accelerating or decelerating.
- **DELV** defines delay on transition from CERRA to CERRV.
- **DELI** defines delay on transition from CERRA to CERRI.

A **#CPE** fault implies a serious problem in motor control. Do not disable the default response unless it is absolutely necessary in your application, i.e., keep **FDEF.#CPE = 1**.

Autoroutine examples

The following autoroutine supplements the default response with an alarm message:

```
on FAULT(3).#CPE
       disp "Axis 3 shows abnormal error. The motor was disabled."
ret
```

6.5.7 Encoder Error: #ENC, #ENC2

Fault bits	**FAULT.#ENC**, **FAULT.#ENC2** (in each element of **FAULT**)
Mask bits	**FMASK.#ENC**, **FMASK.#ENC2** (in each element of **FMASK**)
Based upon safety signals	None
Inversion bits	None
Internal safety condition	The controller latches fault **#ENC** if the phase shift between the signals of the primary encoder is lost, indicating a faulty encoder or noisy environment. The controller latches fault **#ENC2** if the phase shift between the signals of the secondary encoder is lost, indicating a faulty encoder or noisy environment.
Default response bits	**FDEF.#ENC**, **FDEF.#ENC2** (in each element of **FDEF**)
Default response	The controller disables the violating motor. The faults remain active until the user resolves the problems and enables the motor again or executes the **fclear** command.

Unlike most faults, **#ENC** and **#ENC2** faults are latched. The fault bits remain raised even after the cause of the fault has been eliminated. Only the next enable command resets the fault bits.

Occurrence of an **#ENC** fault indicates a serious problem in motor control. Do not disable the default response unless it is absolutely necessary in your application, i.e., keep **FDEF.#CPE = 1**.

Autoroutine examples

The following autoroutine supplements the default response with an alarm message:

```
on FAULT(2).#ENC
       disp "Encoder Error in 2 axis. The motor was disabled."
ret
```

6.5.8 Encoder Not Connected: #ENCNC, #ENC2NC

Fault bits	**FAULT.#ENCNC**, **FAULT.#ENC2NC** (in each element of **FAULT**)
Mask bits	**FMASK.#ENCNC**, **FMASK.#ENC2NC** (in each element of **FMASK**)
Based upon safety signals	None
Inversion bits	None
Internal safety condition	The controller raises fault bit **#ENCNC** if a primary encoder is not connected. The controller raises fault bit **#ENC2NC** if a secondary encoder is not connected.
Default response bits	**FDEF.#ENCNC**, **FDEF.#ENC2NC** (in each element of **FDEF**)
Default response	The controller disables the violating motor.

If the controller detects a pair of differential encoder inputs that are not in opposite states (high and low level), it raises the fault because this may indicate a problem such as a short circuit or unconnected wire.

An **#ENCNC** fault indicates a serious problem in motor control. Do not disable the default response unless it is absolutely necessary in your application, i.e., keep **FDEF.#CPE = 1**.

Autoroutine examples

The following autoroutine supplements the default response with an alarm message:

```
on FAULT(0).#ENCNC
        disp "Axis 0: Encoder Not Connected. The motor was disabled."
ret
```

6.5.9 Drive Alarm: #DRIVE

Fault bits	**FAULT.#DRIVE** (in each element of **FAULT**)
Mask bits	**FMASK.#DRIVE** (in each element of **FMASK**)
Based upon safety signals	**SAFIN.#DRIVE** (in each element of **SAFIN**)
Inversion bits	**SAFINI.#DRIVE** (in each element of **SAFINI**)
Internal safety condition	The controller never sets the fault bit while the motor is disabled. The controller starts monitoring the fault condition when the period of time defined by variable **ENTIME** elapses after the motor has been enabled.
Default response bits	**FDEF.#DRIVE** (in each element of **FDEF**)
Default response	The controller disables the violating motor.

The condition involves the following ACSPL+ variable:

ENTIME – Motor's enable time in milliseconds

Even if the **SAFIN.#DRIVE** bit is in an active state, the controller never raises the fault bit while the motor is disabled. When the **enable** command is issued, the controller waits for the period of time defined by the **ENTIME** variable, and only then starts monitoring the **SAFIN.#DRIVE** bit. If the Drive Alarm signal is still active at that time, the fault condition is satisfied.

The controller continues monitoring the fault condition until the motor is disabled by a **disable** command or a fault that disables the motor.

Occurrence of a **#DRIVE** fault indicates a serious problem in the motor control. Do not disable the default response unless it is absolutely necessary in your application, i.e., keep **FDEF.#CPE = 1**.

General autoroutine example:

The following autoroutine supplements the default response with an alarm message:

```
on FAULT(2).#DRIVE
       disp "Axis 2 Drive Alarm. The motor was disabled"
ret
```

6.5.10 Motor Overheat: #HOT

Fault bits	**FAULT.#HOT** (in each element of **FAULT**)
Mask bits	**FMASK.#HOT** (in each element of **FMASK**)
Based upon safety signals	**SAFIN.#HOT** (in each element of **SAFIN**)
Inversion bits	**SAFINI.#HOT** (in each element of **SAFINI**)
Internal safety condition	None
Default response bits	**FDEF.#HOT** (in each element of **FDEF**)
Default response	None

Autoroutine examples:

The first autoroutine activates the **OUT(0).1** output, which could be wired to switch on an additional motor ventilation fan. The second routine switches off the fan when the fault is no longer active:

```
on FAULT(1).#HOT
       OUT(0).1 = 1
ret



on ^FAULT(1).#HOT
       OUT(0).1 = 0
ret
```

6.5.11 Velocity Limit: #VL

Fault bits	**FAULT.#VL** (in each element of **FAULT**)
Mask bits	**FMASK.#VL** (in each element of **FMASK**)
Based upon safety signals	None
Inversion bits	None
Internal safety condition	If **abs(RVEL) > XVEL**, raise **FAULT.#VL**
Default response bits	**FDEF.#VL** (in each element of **FDEF**)
Default response	The controller kills the violating motor.

The condition involves the following ACSPL+ variable:

XVEL – Maximum allowed velocity for each motor

#VL uses the motor reference velocity **RVEL**, not the motor feedback velocity **FVEL**. Therefore, the fault bit is raised if an application command calls for excessive velocity, even if the motor has not reached this velocity. The fault can also be used for program testing without physical motion, while motors are disabled.

Autoroutine example:

The autoroutine informs you about the violation.

```
on FAULT(2).#VL
        disp "Axis 2 velocity limit was exceeded"
ret
```

6.5.12 Acceleration Limit: #AL

Fault bits	**FAULT.#AL** (in each element of **FAULT**)
Mask bits	**FMASK.#AL** (in each element of **FMASK**)
Based upon safety signals	None
Inversion bits	None
Internal safety condition	If **abs(RACC) > XACC**, raise **FAULT.#AL**
Default response bits	**FDEF.#AL** (in each element of **FDEF**)
Default response	The controller kills the violating motor.

Acceleration limit uses the following ACSPL+ variable:

XACC – Maximum allowed acceleration for each motor

#AL uses the motor reference acceleration **RACC**, not the motor feedback acceleration **FACC**. Therefore, the fault bit is raised if an application command calls for excessive acceleration, even if the motor has not reached this acceleration. The fault also can be used for a program testing without motion, while motors are disabled.

Autoroutine example:

The following autoroutine supplements the default response with an alarm message:

```
on FAULT(0).#AL
       disp "Axis 0 Acceleration limit exceeded. The motor was disabled."
ret
```

6.5.13 Current Limit: #CL

Fault bits	**FAULT.#CL** (in each element of **FAULT**)
Mask bits	**FMASK.#CL** (in each element of **FMASK**)
Based upon safety signals	None
Inversion bits	None
Internal safety condition	If **RMS** current > **XRMS**, raise **FAULT.#CL**
Default response bits	**FDEF.#CL** (in each element of **FDEF**)
Default response	The controller kills the violating motor.

The current limit fault is based on the Servo Processor algorithm that calculates the RMS value of the motor current. When the calculated RMS current exceeds the allowed value the Servo Processor reports an error that the MPU translates into a current limit fault.

Current limit processing uses the following ACSPL+ variables:

- **XRMS** – Maximum allowed RMS current for each motor
- **XCURI** – Maximum instantaneous current if the motor is idle (not moving)
- **XCURV** – Maximum instantaneous current if the motor is moving

Use the Configurator in SPiiPlus MMI to configure the specified variables.

Autoroutine example:

The following autoroutine kills the motion (and displays an alarm message) instead of the motor. (The default response can be disabled by adding **FDEF.#CL = 0** to the ACSPL+ program.)

```
on FAULT(1).#CL
       kill 1
       disp "Axis 1 RMS current limit exceeded. Motor halted."
ret
```

6.5.14 Servo Processor Alarm: #SP

Fault bits	**FAULT.#SP** (in each element of **FAULT**)
Mask bits	**FMASK.#SP** (in each element of **FMASK**)
Based upon safety signals	None
Inversion bits	None
Internal safety condition	If the Servo Processor lost synchronization with MPU, raise **FAULT.#SP**
Default response bits	**FDEF.#SP** (in each element of **FDEF**)
Default response	The controller kills the violating motor.

#SP indicates that communication between the MPU and one of the servo processors failed. The occurrence of the **#SP** fault indicates a serious hardware problem.

Do not disable the default response unless it is absolutely necessary in your application, i.e., keep **FDEF.#SP = 1**.

This fault may be caused by a problem in the SP program. If the SP program hangs, the fault remains permanent. If the SP program time exceeds the tick time (50 μsec), the fault is intermittent.

The disable reason reported by the controller is 5027 'Servo Processor Alarm'.

Autoroutine example:

The following autoroutine supplements the default response with an alarm message.

```
on FAULT(1).#SP
       disp "Axis 1 Servo Processor Alarm"
ret
```

6.5.15 HSSI Not Connected: #HSSINC

Fault bits	**FAULT.#HSSINC** (in each element of **FAULT**)
Mask bits	**FMASK.#HSSINC** (in each element of **FMASK**)
Based upon safety signals	None
Inversion bits	None
Internal safety condition	The controller raises fault bit **#HSSINC** if HSSI is not connected.
Default response bits	**FDEF.#HSSINC** (in each element of **FDEF**)
Default response	None

See also **Section 5.1.7 - Using HSSI I/O Extension**.

Autoroutine example:

The following autoroutine displays an alarm message.

```
on FAULT(0).#HSSINC
       disp "Axis 0: HSSI not connected."
ret
```

6.5.16 Emergency Stop: #ES

Fault bits	**S_FAULT.#ES**
Mask bits	**S_FMASK.#ES**
Based upon safety signals	**S_SAFIN.#ES**
Inversion bits	**S_SAFINI.#ES**
Internal safety condition	None
Default response bits	**S_FDEF.#ES**
Default response	The controller disables all motors.

Autoroutine example:

The following autoroutine kills all motions but does not disable the motors (this assumes that the default response has been disabled by **S_FDEF.#ES = 0**).

```
on S_FAULT.#ES
       killall
ret
```

6.5.17 Program Error: #PROG

Fault bits	**S_FAULT.#PROG**
Mask bits	**S_FMASK.#PROG**
Based upon safety signals	None
Inversion bits	None
Internal safety condition	The controller latches the fault when any ACSPL+ program encounters a run-time error.
Default response bits	**S_FDEF.#PROG**
Default response	The controller kills all executed motions.

Unlike most faults, the **#PROG** fault is latched. Once raised, the bit remains raised until the controller resets it on execution of any command that compiles or starts a program in any buffer.

Autoroutine examples:

The following autoroutine supplements the controller's default response, terminating all concurrent programs and displaying an alarm message.

```
on S_FAULT.#PROG
       stopall
       disp "Run-time error"
ret
```

Note that a run time error in a buffer stops all activity in the buffer. Therefore, the above autoroutine cannot intercept an error that occurred in the buffer where the autoroutine is located. However, it intercepts an error in any other buffer.

This autoroutine can supplement the default response (**S_FDEF.#PROG = 1**) or can replace it (**S_FDEF.#PROG = 0**).

The following autoroutine does the same (stops all programs) and also provides a diagnostic message.

```
on S_FAULT.#PROG
       stopall
       I0 = 0
       loop 10
              if PERR(I0) >= 3020
              disp "Program ", I0, " failed. Error ", PERR(I0)
              end
       IO = 1O + 1
       end
ret
```

The ACSPL+ **PERR** variable contains the termination codes of the ACSPL+ programs. Each element of **PERR** contains a termination code for a different buffer. At power-up all elements of **PERR** are reset to 0. When a program in any buffer finishes or terminates for any reason, the corresponding element of **PERR** is assigned with a code that specifies the termination reason.

The element resets to zero again once the corresponding buffer is compiled or its program execution starts.

Termination codes from 3000 to 3020 indicate normal termination. Codes greater than or equal to 3020 indicate run-time errors (see *SPiiPlus Command & Variable Reference Guide* for a complete breakdown of the termination codes).

6.5.18 Memory Overflow: #MEM

Fault bits	**S_FAULT.#MEM**
Mask bits	**S_FMASK.#MEM**
Based upon safety signals	None
Inversion bits	None
Internal safety condition	The controller latches the fault bit when the user application requires more memory than is available in the controller.
Default response bits	**S_FDEF.#MEM**
Default response	The controller kills all executed motions.

Unlike most faults, the **#MEM** fault is latched. Once raised, the bit remains raised until the controller resets it on execution of any command that compiles or starts a program in any buffer.

Because the controller uses dynamic memory handling, the amount of the memory available for a specific user application cannot be exactly determined. If an application raises this fault, you need to reduce the size of the application or add memory.

The following recommendations may be useful in eliminating the error:

- ❑ Reduce the length of ACSPL+ programs.
- ❑ Reduce the volume of user local and global variables. Pay special attention to arrays.
- ❑ Limit the length of commands that are sent to the controller. Do not use commands that exceed 2032 characters.
- ❑ Simplify the formulae used with the **connect** and **master** commands.

Autoroutine example:

The following autoroutine terminates all executing programs and displays an error message when the fault occurs.

```
on S_FAULT.#MEM
       stopall
       disp "Memory overflow"
ret
```

This routine can supplement the default response (**S_FDEF.#PROG = 1**) or can replace it (**S_FDEF.#PROG = 0**).

6.5.19 Time Overuse: #TIME

Fault bits	**S_FAULT.#TIME**
Mask bits	**S_FMASK.#TIME**
Based upon safety signals	None
Inversion bits	None
Internal safety condition	The user application demands too much processing time.
Default response bits	**S_FDEF.#TIME**
Default response	The controller kills all executed motions.

The controller raises the fault bit when the user application consumes too much processing time and **S_FMASK.#TIME** bit is raised.

The structure of the controller's real-time cycle is discussed in **Section 2.1.3 - realtime and Background Tasks** .

As the real-time processing time varies between cycles, the fault may occasionally occur and requires no special attention. However, frequent or permanent occurrence of the fault requires measures to correct the situation.

The controller has no default response to the fault. To monitor this fault, you must define your own autoroutine.

The following recommendations may be useful in reducing real time processing time, thereby eliminating the fault:

- ❑ Reduce the number of concurrently executed programs.
- ❑ Reduce the program execution rates (variables **PRATE**, **ONRATE**).
- ❑ Reduce the number of command specified in one program line.
- ❑ Reduce the number of autoroutines.
- ❑ Simplify the conditions in the autoroutines.
- ❑ Reduce the number of concurrently executed motions.
- ❑ Avoid short-time motions.
- ❑ Use segmented motion instead of a series of short PTP motions.
- ❑ Simplify the formula used in the **connect** and **master** commands.

Autoroutine example:

The following autoroutine accumulates statistics on the fault. The routine measures a time of 1000 fault occurrences and then displays the average time between faults. The routine relies on zero initialization of the local variables.

`Local int N`	Declare local variable for counting the number of faults.
`Local real ATIME`	Declare local variable that will show the time of 1000 faults.

```
on S_FAULT.#TIME                                    Activate routine when the TIME fault occurs.
 N = N + 1                                          Increment the number of faults
 If N >= 1000                                       If 1000 faults were accumulated…
 disp "#TIME occurs once per ", (TIME - ATIME)      Display average time between faults.
/ N, " ms"
 ATIME = TIME                                       Prepare for the next accumulation
 N = 0                                              Zero to prepare for the next accumulation
 end
ret
```

6.5.20 Servo Interrupt: #INT

Fault bits	**S_FAULT.#INT**
Mask bits	**S_FMASK.#INT**
Based upon safety signals	None
Inversion bits	None
Internal safety condition	The controller raises the fault bit when the servo interrupt is not generated or is irregular.
Default response bits	**S_FDEF.#INT**
Default response	The controller disables all motors.

Caution	
	***The Servo Interrupt fault indicates a serious failure. Do not disable the default response unless it is absolutely necessary in your application, i.e., keep* FDEF.#INT = 1.**

The MPU sets the fault if the 1ms interrupt is not received. The probable cause is a hardware problem. The controller response to the fault is to disable all motors. The disable reason reported by the controller is 5029 'Servo Interrupt'.

Autoroutine example:

The following autoroutine supplements the default response with termination of all ACSPL+ programs and an alarm message.

```
on S_FAULT.#INT
        stopall
        disp "Main interrupt is missing"
ret
```

6.5.21 Component Failure Faults: #FAILURE

Some components other than Drive, for example, power supply, I/O extension card or encoder card, may have fault outputs that are common for all components. The controller provides system faults that by default disable all axes if fault occurs.

The controller provides special functions that retrieve the card malfunctioned information (according to address on the I^2C bus) and the fault reason. This allows to user to write an ACSPL+ application that provides user-defined responses for different component faults. The components that have such outputs provide a special jumper that connects or disconnects the fault output to the controller.

6.5.21.1 Safety Variables

Model	*Currently only MC4U systems support these variables.*

Component Failure fault is treated as any other system fault. There is a dedicated bit 31 (**#FAILURE**) for this fault in **S_FAULT**, **S_FMASK**, **S_SAFIN**, **S_SAFINI** and **S_FDEF**.

S_FAULT.#FAILURE (or **S_FAULT.31**)	Indicates if there is fault or not. 1 = An MC4U hardware component other than the drive, such as the Power Supply, I/O card, or encoder card, has failed.
S_FMASK.#FAILURE (or **S_FMASK.31**)	Defines if the Component Failure Fault will be examined by the controller. 1 = Enables **S_FAULT.#FAILURE**.
S_FDEF.#FAILURE (or **S_FDEF.31**)	You use this varaiable for triggering a response that you have programmed. 1 = An MC4U hardware component other than the drive, such as the Power Supply, I/O card, or encoder card, has failed.
S_SAFIN.#FAILURE (or **S_SAFIN.31**)	Indicates the actual status of the Component Failure controller input - see **Section 6.6.3 - Examining System Fault Conditions**.
S_SAFINI.#FAILURE (or **S_SAFINI.31**)	Used for inverting the Component Failure input logic.

6.5.21.2 Component Failure Fault Handling in ACSPL+

The **GETCONF(247,<axis>)** function serves for retrieving the component malfunction information (according to address on the I^2C bus) and the fault reason. Using the 247 key triggers the function to retrieve a 32-bit integer number containing the following information about the failed component: Error Code, Card Address and Card Type. **Figure 16** shows the structure of the error data number. In the event that more than one component failed, Bit 15 is set to "1". The malfunctioning device is automatically reset.

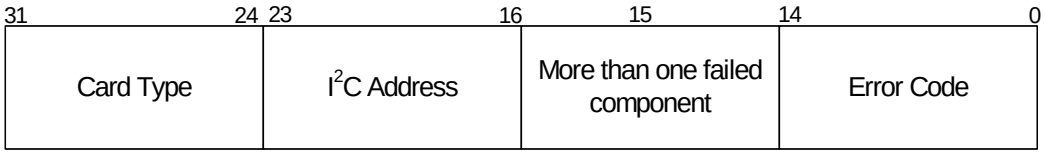


Figure 16 32-bit Error Data Number

The following ACSPL+ example shows how to treat the return value of the **getconf(247,<axis>)** function:

```
# Local variables
INT res, error, i2c_addr, dev, nexterr
ON S_FAULT.#FAILURE
REP:
       res = GETCONF(247, 0)
       error = res & 0x7FFF
       nexterr = res.15
       i2c_addr.0 = res.16
       i2c_addr.1 = res.17
       i2c_addr.2 = res.18
       dev.0 = res.24
       dev.1 = res.25
       dev.2 = res.26
       dev.3 = res.27
       dev.4 = res.28
       dev.5 = res.29
       dev.6 = res.30
       dev.7 = res.31
       DISP "ERROR =",error
       DISP "I2C ADDRESS =",i2c_addr
       DISP "DEVICE =",dev

       IF(nexterr)
        GOTO REP
RET
```

6.6 Detailed Description of Safety Controls

This section explains the details of safety control implementation in the controller software. Read this section if you want a deeper understanding of how the controller analyzes safety inputs, examines safety conditions and processes faults.

Note	
	*The* **SAFIN** *and* **S_SAFIN** *variables are normally read-only. However, they can be written to when working with the Simulator, to simulate safety inputs.*

6.6.1 Examining Fault Conditions - Flow Chart

Figure 17 illustrates how the controller examines fault conditions:

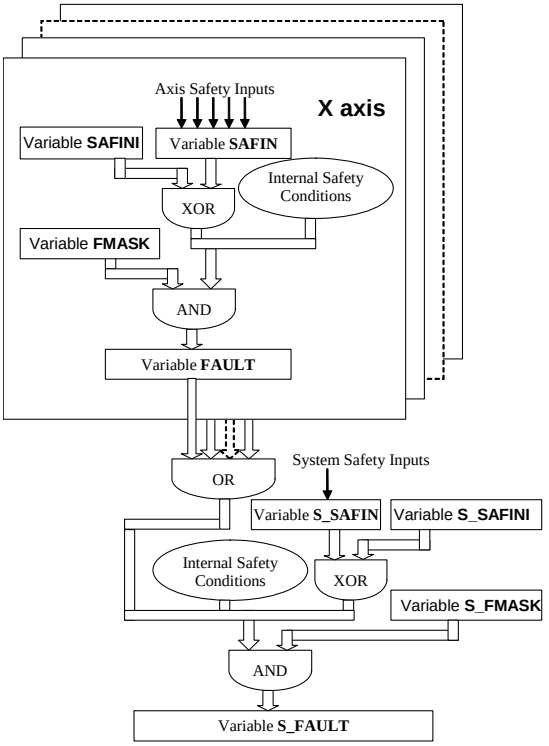


Figure 17 Fault Examination Flow Chart

The upper part of the diagram shows how motor faults are examined. The list of faults is identical for each motor and the controller repeats the process for each motor. The end product is the ACSPL+ **FAULT** variable.

The lower part of the diagram shows the elements that go into constructing the **S_FAULT** variable. Part of its bits are set as the OR-aggregate of the **FAULT** elements, and other bits are determined by examining the system faults.

6.6.2 Examining Motor Fault Conditions

The controller monitors motor fault conditions each controller cycle for each axis. There are two sources of motor faults:

- Motor safety inputs
- Internal safety conditions

The controller samples motor safety inputs each controller cycle and stores the values in the **SAFIN** variable. High voltage of a signal is stored as one in the corresponding bit of **SAFIN**. Low voltage is stored as zero. Only the following bits of **SAFIN** are meaningful:

- **#LL** - Left Limit
- **#RL** - Right Limit
- **#LL2** - Preliminary Left Limit
- **#RL2** - Preliminary Right Limit
- **#HOT** - Overheat
- **#DRIVE** - Drive Fault

For example the command:

```
if SAFIN(0).#DRIVE  V0 = 1 else V0 = -1 end
```

assigns a value of 1 to variable **V0** if the Drive Alarm signal of the 0 axis motor is high and -1 if low.

The **SAFINI** configuration variable defines which level of motor safety input causes a fault. In the above diagram XOR is a bit-wise operation. Therefore, if a bit of **SAFINI** is zero, high voltage of the corresponding signal causes fault. If a bit of **SAFINI** is 1, low voltage causes fault. Only those bits of **SAFINI** that correspond to the meaningful bits of SAFIN are used in fault processing. Other bits have no effect.

In addition to the safety inputs, the controller examines a number of internal safety conditions for each motor each controller cycle. The faults caused by the motor safety inputs and the faults detected by internal conditions provide a set of motor faults.

A detected motor fault is stored in a bit of variable **FAULT** only if the corresponding bit of variable **FMASK** is 1. If a bit of **FMASK** is zero, the controller does not raise the corresponding fault bit even if the fault condition or safety input is true. If a bit of **FMASK** is set to 1, the corresponding bit of **FAULT** is immediately set when the fault occurs. The bit rises to 1 or drops to zero in the same controller cycle as the corresponding safety input or internal safety condition shows change in the fault state.

Only those bits of **FAULT** that correspond to the motor faults are meaningful.

When a bit is raised, it activates the default response to the fault. An Autoroutine that processes the fault must use the bit of **FAULT** in as the condition.

6.6.3 Examining System Fault Conditions

System safety inputs and internal system safety conditions are monitored similarly to motor fault conditions.

There are three sources of **S_FAULT** bits:

- Aggregated motor faults
- System safety inputs
- Internal system safety conditions

Aggregated motor faults are implemented as a combination of motor faults for all axes. For example, if bit **FAULT.#LL** is one, bit **S_FAULT.#LL** also rises to one. Bit **S_FAULT.#LL** drops to zero only if the **#LL** bits in all **FAULT**s for all motors are zero. Each meaningful bit of **FAULT** has its counterpart in **S_FAULT**.

Emergency Stop is the only system safety input. The controller samples the input each controller cycle and stores the value in the **S_SAFIN** variable. Therefore, only one bit, **S_SAFIN.#ES**, is meaningful. High voltage of the signal is stored as one in the bit, low voltage is stored as zero. An application uses **S_SAFIN** if a raw immediate value of Emergency Stop input is required.

The configuration variable **S_SAFINI** defines which level of Emergency Stop input causes a fault. In **Figure 17** the XOR is a bit-wise operation. Therefore, if bit **S_SAFINI.#ES** is zero, high voltage of Emergency Stop causes a fault. If bit **S_SAFINI.#ES** is one, low voltage causes fault. Only one bit of **S_SAFINI** is used in fault processing. Other bits have no effect.

Each controller cycle the controller examines a number of internal system safety conditions. The aggregated motor faults, the Emergency Stop fault and the faults detected by internal conditions provide a set of system faults.

A system fault is stored in a bit of variable **S_FAULT** only if the corresponding bit of variable **S_FMASK** is one. If a bit of **S_FMASK** is zero, the controller does not raises the corresponding **S_FAULT** bit even if the fault is detected. If a bit of **S_FMASK** is one, the corresponding bit of **S_FAULT** follows the current state of the fault. The bit rises to one or drops to zero in the same controller cycle as the corresponding fault changes its state.

Only those bits of **S_FAULT** that correspond to the aggregated motor faults and to the system faults are meaningful.

When a bit of S_FAULT is raised, it activates the default response to the fault. An autoroutine that processes the fault must use the bit of **S_FAULT** as the condition.

6.7 Extended Fault Configuration

Warning	
	***The controller's default safety configuration and responses fit the needs of most applications.*** ***Only an experienced user should make modifications to the safety configuration. Improper configuration may result in unsafe operation, may damage the equipment, and may constitute a hazard to personnel.***

As mentioned earlier in this chapter, the controller response to a fault can be modified with the ACSPL+ **FDEF**/**S_FDEF** variables and with autoroutines.

The **#SC** terminal command reports the current safety system configuration (see **Section 6.2.7 - Report Safety Configuration**).

There are several other options for safety system configuration:

- **Safety groups**

  Two or more axes can be combined in a safety group. All axes belonging to the safety group respond to any fault synchronously. If a fault affects one axis in the group, it immediately affects all the other axes in the group (refer to the **safetygroup** command in the *SPiiPlus Command & Variable Reference Guide*).

- **Kill-disable response**

  In response to a fault, the controller executes **kill**, decelerates the motor to zero velocity, and then disables the motor.

- **Changing response without an autoroutine**

  An axis can respond to a fault in one of the following basic ways:

  - No response
  - Kill response
  - Disable response
  - Kill-disable response

  For each type of fault, the controller defines a default response, which can be overridden with an autoroutine. The **safetyconf** command (refer to the **safetyconf** command in the *SPiiPlus Command & Variable Reference Guide*) switches an axis between the four basic fault responses. An autoroutine is only required if the desired response is not one of the these.

- **Fault generalization**

  The fault response for a specific fault and a specific axis can be generalized to affect all the axes. For example, by default the 0 Drive Alarm fault disables the 0 axis motor, but has no affect on the 1 axis motor or any other motor. However, if you generalize the Drive Alarm fault for axis 1, the 1 axis motor will be affected by a Drive Alarm fault on any axis.

- **Specific motor configuration**

  The default configuration for all axes is identical. For example, the default response to a Limit Switch fault is to kill motion. However, the response can be modified individually for each motor. For example, if a Limit Switch fault occurs, the 0 axist motor can be killed while the 1 axist motor is disabled.

7 Connection to the Plant

This chapter details the connection between the motion controller and the controlled plant.

7.1 General Diagram

All connections between the controller and the controlled plant are provided through the SP processors as shown in the following diagram:

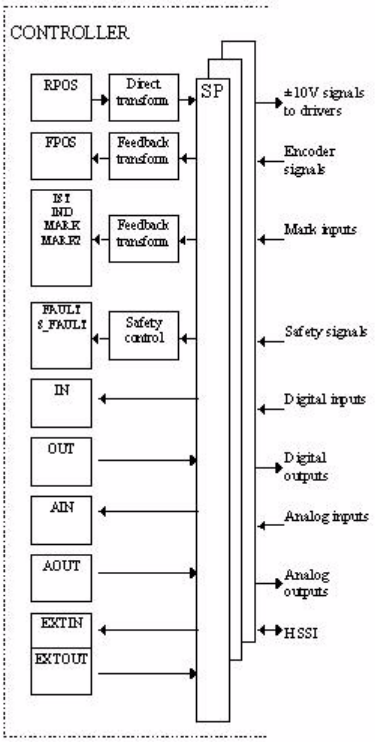


Figure 18 SPiiPlus-Plant Connections and Related Parameters

The ACSPL+ variables that provide access from your application to the control object are shown to the left. The following variables are available:

- **RPOS** – Reference Position, array of eight elements, contains the desired motor position calculated by the controller.
- **FPOS** – Feedback Position, array of eight elements, reads the current motor position.
- **IND** – Index Position, array of eight elements, reads the position of the last encountered encoder index.
- **MARK** – Mark Position, array of eight elements, reads the position of the last encountered Mark signal.
- **MARK2** – Secondary Mark Position, array of eight elements, reads the position of the last encountered Mark2 signal.

- **FAULT** – Faults, array of eight elements, the bits of the variable report the axis-related faults detected by the safety control
- **S_FAULT** – System Faults, scalar, the bits of the variable report the system faults detected by the safety control.
- **IN** – General Purpose Inputs, array of eight elements, each bit reports the state of one general-purpose input.
- **OUT** – General Purpose Outputs, array of eight elements, each bit defines the state of one general-purpose output.
- **AIN** – Analog Inputs, array of 16 elements, each element reads the numerical value of the voltage supplied to one analog input. The number of analog inputs varies depending on the controller model.
- **AOUT** – Analog Outputs, array of 16 elements, each element defines the voltage generated by one analog output. The number of analog inputs varies depending on the controller model.
- **EXTIN** – Extended Inputs, array of eight elements, each bit represents a bit in the input or HSSI register.
- **EXTOUT** – Extended Outputs, array of eight elements, each bit represents a bit in the HSSI register

7.2 User-Defined Units

The controller allows you to define the units used for motion programming and monitoring. This is done by setting the value of the **EFAC** variable.

During the configuration stage, you can set the value of the **EFAC** variable to specify the ratio between the desired unit and an encoder count for each axis. For example, if the encoder resolution is 1000 counts per millimeter, and you desire to program in millimeters, the corresponding **EFAC** element must be set to 0.001.

The user-defined units apply to all motion commands, so that all position-related arguments must be specified in user-defined units. The user-defined units also affect all position-related standard variables. For example, if the user-defined units are millimeters, then the unit of **RPOS**, **FPOS**, **APOS**, **MPOS**, **IND**, **MARK**, etc., will be in millimeters.

The user unit also affects velocity, acceleration and jerk variables. For example, if the **EFAC** value defines the unit as millimeter, then the units of **VEL**, **RVEL**, **FVEL**, **XVEL**, etc., will all be millimeters per second. The unit of the variables **ACC**, **DEC**, **KDEC**, **RACC**, **FACC**, etc., will be millimeters per second as well as the unit of the variables **JERK**, **GJERK**.

You can define the same or different units for each axis, irrespective of the encoder resolution.

Example (rotary motor):

```
EFAC(0) = 360 / (2000x4)
```

Where:

Feedback resolution	2000 lines/rotation
Internal Multiplier	x4
User units	360 degrees per rotation
EFAC value	0.045 (i.e., 360/(2000*4))

Example (linear motor):

```
EFAC(0) = 1 / (500*4)
```

Where:

Feedback resolution	2 micron (500 lines/mm)
Internal Multiplier	x4
User units	mm
EFAC value	0.005 (i.e., 1/(500*4))

Example (linear motor, sin-cos encoder)

```
EFAC(0) = 1 /(500*64)
```

Where:

Feedback resolution	500 lines/mm
Internal Multiplier	x64
User units	mm
EFAC value	0.00003125 (i.e., 1/(500*64))

Example (rotary motor connected to ball screw)

```
EFAC(0) = 1 / (2000*4*(1/0.5))
```

Where:

Feedback resolution	2000 lines/ rotation
Gear ratio	1 motor rotation=0.5mm motion of load
Internal Multiplier	x4
User units	mm
EFAC value	0.000625 (i.e., 1/(2000*4*(1/0.5)))

7.3 Direct and Feedback Transform

The SPiiPlus controller accepts the encoder signals and calculates the feedback position in encoder counts. All servo algorithms in the SPiiPlus controller are based on encoder counts, not user units. Therefore, when reading the feedback position from the SP, the controller executes feedback transforms according to the formula:

FPOS = FP*EFAC + EOFFS

where **FPOS** is the controller feedback position in user units, **FP** is an SP-calculated feedback position in encoder counts, **EFAC** is a user-defined value of the corresponding **EFAC** factor, and **EOFFS** represents an offset.

EOFFS is a read-only standard variable that provides the offset between the SP-calculated position and the controller feedback position. The value of **EOFFS** changes when the set command defines a new origin for an axis.

When writing the reference position to the SP, the controller executes a direct transform given by the formula:

RP = (RPOS – EOFFS) / EFAC

where **RPOS** is the controller reference position in user units, and **RP** is the SP reference position in encoder units.

7.4 Index and Mark Values

Index and Mark processing is based on the SP hardware latch of the encoder reading when the index or mark input signal occurs. The latched position is accurate to within one encoder count at speeds of up to 5 million counts per second.

The following input signals are used for position latching:

- Encoder Index – One index signal is available per axis.
- **MARK** and **MARK2** – These signals are available only for axes 0, 1, 2, and 3.

Each of the three signals has an independent circuit and a separate latch register. The circuit latches the encoder reading in the register on the positive edge of the corresponding signal.

The controller further processes the latched signal. In this example we will use the index to explain the process. Processing the mark values is similar.

When the index signal latches the encoder reading, the controller raises a flag (the bit **IST.#IND**), converts the value into user units (scale and offset), and stores the value in the **IND** variable. Afterwards, as long as the **IST.#IND** bit is set, the controller ignores any new index occurrences. Therefore, the **IND** variable does not change even if a new index was encountered. To reactivate the index latching, you must explicitly reset the **IST.#IND** bit.

The following program fragment reports the index value each time that the index is encountered:

```
IST0.#IND = 0          Reset any index encountered previously
INDREP                 Start of the loop
 till IST0.#IND        Wait until the index is latched
```

`disp IND0`	Report the index position
`IST0.#IND = 0`	Reactivate index latching
`goto INDREP`	Go to the start of the loop

7.5 Safety Inputs

The following safety inputs are available for each axis:

- **Left Limit** – Signal from Left Limit switch.
- **Right Limit** – Signal from Right Limit switch.
- **Drive Alarm** – Alarm signal from the drive; normally is on when the drive is disabled, and is off when the drive is enabled.
- **Network** – Loss of network connectivity has been detected.
- **Cooling Fan** – .Loss of cooling fan has been detected.
- **Overheat** – Signal from a temperature sensor (not supported in all controller models).

The following safety input is not related to a particular axis and indicates a general failure of the control object:

- **Emergency Stop** – General failure of the control object.

The controller processes each safety input and raises the corresponding fault bit in the **FAULT** or **S_FAULT** variable accordingly. The following bitmapped variables are involved in the processing of the safety inputs:

- **SAFIN** and **S_SAFIN**, read-only variables (except when used with Simulator) – Indicate the raw state of safety inputs before any processing.
- **SAFINI** and **S_SAFINI**, configuration variables – Define the active state of each signal, specifying inversion of a signal if required.
- **FMASK** and **S_FMASK**, configuration variables – Enable or disable using each signal in the safety control.
- **FAULT** and **S_FAULT**, read-only variables – Indicate the faults detected by the safety control.
- **FDEF** and **S_FDEF**, configuration variables – Enable or disable the controller default response when the fault occurs.

7.6 Digital Inputs/Outputs Repetitive

A digital input accepts a binary signal from an external source, such as a switch or a relay. A digital output provides a binary signal to an external acceptor such as an LED or actuator. Unlike the safety inputs, a digital input or digital output has no predefined function in the controller. You can connect signals to inputs or outputs and process them as required by the application.

The inputs are represented by the integer array: **IN**. The outputs are represented by the integer array: **OUT**. Each digital input or output is treated as one binary bit. The low voltage level corresponds to zero and high voltage level corresponds to one. Each element of the **IN** and **OUT** arrays is a 32-bit integer number that can represent up to 32 inputs or outputs. In all current SPiiPlus models, the number of inputs and outputs is less than 32; therefore all actual inputs and outputs are represented in **IN0** and **OUT0**. All other elements of **IN** and **OUT** are reserved for future extension. For the exact number of inputs/outputs see the specifications of the controller model.

The following example shows an autoroutine that changes the state of output 5 once the state of input 3 changes from 0 to 1.

```
on IN0.3              Activate autoroutine on positive edge of input 3.
 OUT0.5 = ^OUT0.5     Invert the state of output 5.
ret                   End of the autoroutine
```

7.7 Analog Inputs/Outputs

An analog input accepts analog signal from an external source, such as a sensor or a potentiometer. An analog output provides analog signal to an external receiver, such as an actuator or a measuring device. Analog inputs and outputs have no predefined function in the controller. You can connect signals to inputs and outputs and process them as required by the application.

The analog inputs are represented by the integer array: **AIN** and the analog outputs are represented by the integer array: **AOUT**. Each analog input/output is represented by one array element. The range of the **AIN** and **AOUT** arrays depends on the type of the input or output and the bit resolution of the **ADC** or **DAC**.

Example, for ±10V analog outputs with 16-bit **DAC** resolution, the **AOUT** range is from -32768 (for -10V) to +32767 (for +10V).

Example: For ±1.25V analog inputs with 14-bit **DAC** resolution, the **AIN** range is from -8192 (for -1.25V) to +8192 (for +1.25V).

If an analog output is connected to a drive, it has a dedicated destination and cannot be used as a general-purpose analog output.

For model-dependent analog I/O information (for example, the number and range of inputs and outputs) see the controller's Hardware Guide.

The following example represents an autoroutine in SPiiPlus PCI-4/8 that displays a message when the voltage in the 3rd analog input rises above +.75V:

```
on AIN3 > 4096          Activate autoroutine once the value of AIN3 exceeds 4096 (+.75V)
 disp "AIN3 > .75V"     Display a message
ret                     End of the autoroutine
```

7.8 High-Speed Synchronous Serial Interface

The High-Speed Synchronous Serial Interface (HSSI) provides a simple interface for various external devices.

The HSSI provides up to 256 input bits and up to 256 output bits. Not every controller model supports all 512 bits. For the exact number of inputs and outputs supported, see the controller model's Hardware Guide.

The HSSI bits are represented by an integer array: **EXTIN** or **EXTOUT**. Each HSSI bit corresponds to one bit in one element of the **EXTIN** and **EXTOUT** array.

The HSSI bits have no predefined function in the controller. You are free to use HSSI as required in the application. In the simplest case, the HSSI bits can be used for an extension of the digital inputs/outputs. In a more sophisticated application, any sensor or actuator including encoders and drives can be connected through the HSSI.

The HSSI interface is described in detail in the *SPii/SPiiPlus HSSI Modules Guide*.

8 Advanced Features

This chapter describes various advanced ACSPL+ programming features that are available to you. Topics covered are:

- **Data Collection**
- **Position Event Generation (PEG)**
- **Sin-Cos Encoder Multiplier Configuration**
- **Interrupts**
- **Dynamic Braking**
- **Constant Current Mode**
- **Hall Sensor Commutation**
- **Communicating with the SPiiPlus C Library**
- **Communicating with 3rd Party Devices**
- **trigger Command**
- **Dynamic TCP/IP Addressing**
- **Non-Default Connections**
- **Input Shaping**
- **DRA Algorithm**
- **BI-Quad Filter**

8.1 Data Collection

Data collection is useful in the following applications:

- Troubleshooting
- Adjusting servo control loops
- Applications that require detailed information about internal controller processes
- Teach-and-Go applications

8.1.1 dc Command

Description

The **dc** command is used for executing data collection.

Syntax

dc*[/switch] [axis], array_name, #_of_samples, period, list_of_variables*

stopdc[/switch] [integer]

Where **switch** can be one or a combination of:

s	Start data collection synchronously to a motion.
w	Create the synchronous data collection, but do not start until a **go** command is issued. **w** can only be used with the **s** switch.
t	Temporal data collection: sampling period is calculated automatically according to collection time.
c	Start cyclic data collection (can be used with switches **s** and **w**).

- Data collection started by the **dc** command without the **s** switch is called system data collection.
- Data collection started by the **dc/s** command is called axis data collection.
- Data collection started by the **dc/c** command is called cyclic data collection.

  Unlike the standard data collection that finishes when the collection array is full, cyclic data collection does not self-terminate. Cyclic data collection uses the collection array as a cyclic buffer and can continue to collect data indefinitely. When the array is full, each new sample overwrites the oldest sample in the array. Cyclic data collection can only be terminated by the stopdc command.

The **dc** command is used with the following parameters:

axis	Axis to which the data collection must be synchronized. The parameter is required only for axis data collection (**s** switch).
array_name	Array that stores collected data. The array must be previously defined as global, integer or real. The array size must be compatible with the number of samples and number of stored variables (see **Section 8.1.4 - Understanding System Data Collection**).
#_of_samples	The number of data samples to collect.
period	Sampling period in milliseconds. Actual sampling period may differ from this value, because the controller rounds it to an integer number of controller periods. For example if you set the period to 3.3 milliseconds, the controller will round it to 3 milliseconds. If switch **t** is included in the command, the parameter defines a minimal period (see explanation of temporal data collection below).
list of variables	Up to eight variable names, whose values are to be collected. Each variable can be a scalar variable, or an element of an array. Both local and global variables can be specified. Irrespective of the storage array type, any combination of integer and real variables is allowed - the controller automatically executes type conversion if required.

8.1.2 spdc - High-Speed Data Collection

Description

The **spdc** command starts data collection from an SP variable.

Syntax

spdc *array, number_of_samples, period, sp_variable, shift*

Aguments

array	A global one-dimensional user-defined integer array that accumulates the data.
number_of_samples	The number of samples that must be collected (must be less than or equal to the array size.
period	The sampling period.
sp_variable	Specification of the SP variable, its format is: sp-number:variable-name. For example, **SP1:MPU_X_CONTROL**.
shift	Specifies the scaling for the samples as a number of right shift.

The minimum sampling period is 0.05 millisecond, which defines a sampling frequency of 20kHz. If **CTIME** is 2 millisecond, minimal sampling period is 0.1 millisecond.

The maximum sampling period is limited by **CTIME** value. By default, **CTIME** is 1 millisecond, which restricts sampling period to 1 millisecond maximum.

The controller rounds the specified period to an integer number of minimal periods and restricts it to the permitted range.

Due to hardware limitations, the controller collects 12 bit values, i.e., the collected samples must be in the range from -2048 to +2047. If the collected variable goes out of the range, you have to define proper shifts (through the **shift** argument) to bring the values into the range.

8.1.3 ACSPL+ Variables Involved in Data Collection

Data collection uses the following ACSPL+ variables:

AST	Axis State – Bit **#DC** in this bit-mapped variable is ON while an axis (started with /**s**) data collection for the corresponding axis is in progress. The bit is OFF if no axis collection for the axis is executed.
DCN	DC Number of Samples (axis variable) – While an axis data collection is in progress the variable displays the index of the array element that stores the next sample. When an axis data collection for the corresponding axis terminates, the variable stores the number of actually collected samples. If the data collection terminates automatically, the variable is always equal to the requested number of samples specified in the **dc** command. If the data collection terminates due to the **stopdc** command, the variable may be less than the requested number of samples.

DCP	DC Period (axis variable) – When an axis data collection for the corresponding axis terminates, the variable stores the sampling period. Unless a temporal data collection was executed, the variable is always equal to the requested period specified in the **dc** command. For temporal data collection (**/t**) the variable may be greater than the requested minimal period.
S_DCN	General DC Number of Samples – While a system data collection is in progress the variable displays the index of the array element that stores the next sample. When a system data collection terminates, the variable stores the number of actually collected samples. If the data collection terminates automatically, the variable is always equal to the requested number of samples specified in the dc command. If the data collection terminates due to the **stopdc** command, the variable may be less than the requested number of samples. For cyclic data collection **S_DCN** displays the current number of collected samples and changes as follows: 1. At the start of data collection **S_DCN** is assigned with zero 2. Each sampling tick **S_DCN** is incremented until it reaches the size of the sample array 3. **S_DCN** then remains unchanged (the new sample overwrites the oldest and the total number of samples remains the same) As long as the cyclic data collection is in progress, the application cannot use the sample array. After the cyclic data collection finishes, the controller repacks the sample array so that the first element represents the oldest sample and the last element represents the most recent sample.
S_DCP	System DC Period – When a system data collection terminates, the variable stores the sampling period. Unless a temporal data collection was executed, the variable is always equal (see explanation of the **period** parameter above) to the requested period specified in the dc command, unless it was rounded to the controller cycle . For temporal data collection (**/t**) the variable may be greater than the requested minimal period.
S_ST	System State – Bit: **#DC** in this bit-mapped variable is ON while a general (started without **/s**) data collection, either standard or cyclical, is in progress. The bit is OFF if no general collection is executed.

8.1.4 Understanding System Data Collection

Note	
	*Data collection is disabled when the SPiiPlus MMI Application Studio **Scope** is operating.*

When the controller executes the **dc** command, it starts a separate real-time data collection process that progresses in parallel with ACSPL+ programs and motion execution. Each sampling period the process latches the values of all specified variables and stores them in the specified array. The process continues until the specified number of samples is stored, or the command stopdc is executed. After process termination the array contains a time series of the specified variables that may then be used for data analysis.

This is shown in the following example:

```
global real DCA(2)(1000)
dc DCA, 990, 3, FPOS(0), FPOS(1)
```

- In the first line, a matrix consisting of two columns and 1,000 lines is set up for data collection
- The second line starts the data collection of the Feedback Position values for axes X and Y. 990 samples are to be collected, with a period of three milliseconds. The first step of the data collection stores the current value of **FPOS0** in **DCA(0)(0)** and **FPOS1** in **DCA(1)(0)**. The second step stores **FPOS0** in **DCA(0)(1)** and **FPOS0** in **DCA(1)(1)**.

Each variable is stored in one line of the array. Therefore the first dimension of the array (the number of lines) must be equal or greater than the number of variables. If the number of lines is greater than the number of variables, the extra array lines remain unaffected by the data collection. If only one variable is specified for data collection, a one-dimensional array is allowed.

Each sample of data collection fills up one column of the array. Therefore the second dimension of the array (number of columns) must be equal or greater than the requested number of samples. If the number of columns is greater than the number of samples, the extra array columns remain unaffected by the data collection. The following examples show incorrect usages of the **dc** command.

The following **dc** command is not allowed because the number of variables exceeds the number of array lines:

```
global int IA, IC(1000)
dc IC,1000, 1, IA, FPOS(0)
```

The following **dc** command is not allowed, because the number of required samples exceeds the number of array columns:

```
global int IA, IC(1000)
dc IC,1001, 1, IA
```

If **/s** is not specified, the system data collection process starts immediately after executing the dc command. Normally, data collection stops automatically after <number of samples> x <period> milliseconds. If the stopdc command executes while the data collection is in progress, the data collection stops prematurely, and the remaining array columns remain unaffected. To terminate system data collection, the **stopdc** command must contain no parameters.

Note	*The variable* **S_DCN** *provides the number of samples stored during data collection*

The following are examples of **dc** commands:

`global int IA, IC(1000)`	Set up a one-dimensional 1000-line array for collecting data about a user-defined integer variable.
`global real RA(2)(2000)`	Set up a two dimensional array with two columns and 2000 lines for collecting data about a real user-defined variable.
`dc IC, 1000, 1, AOUT(0)`	Collect 1000 samples of AOUT on the 0 axis at a rate of one per millisecond.
`dc RA, 2000, 2.5, IA, FPOS(7)`	Collect 2000 samples of FPOS on axis D, at a rate of 3 per millisecond (it should be noted that the command calls for 2.5 milliseconds, but the controller rounds it up to the nearest whole number).
`dc RA, 500, 1, TIME`	Collect 500 samples of the ACSPL+ variable **TIME** at a rate of one per millisecond

8.1.5 Axis Data Collection

When switch **s** is included in the **dc** command, data collection starts synchronously with a motion. The first parameter must specify an axis with which the data collection is synchronized.

If the axis is idle at the moment when the **dc/s** command executes, no synchronization is provided and the data collection starts immediately just as if the **dc** command was executed without the switch. If the axis or axis group is in motion when the **dc** command executes, the data collection lines up in the motion queue and waits for all motions that were planned before it, to finish. When the motion queue comes to the data collection, the data collection starts, and immediately the next motion in the motion queue starts. Data collection therefore introduces no delay in the motion queue and does not affect motion execution.

Having started, data collection continues in parallel to the executed motion. Data collection finishes when the specified number of samples is stored, or the stopdc command executes.

Data collection initiated by the **dc** command with no associated parameters is stopped by the stopdc command with no associated parameters. To stop synchronous data collection initiated by the **dc/s** command, the **stopdc** command must also include the **s** switch and the axis specification.

The **dc/s** command synchronizes the data collection start to the motion of an axis, but is not limited to collecting data only of that axis. Any parameter for any other axis may be specified for data collection. For example, the command

```
global real Array(2)(5)
dc/s 0,Array,500,1,FPOS(0),FPOS(1)
```

synchronizes data collection to the start of motion of axis 0, but collects data on both axes 0 and 1.

Only one data collection process, started by the **dc** command, can execute at a time. The next **dc** command can execute only after the data collection started by the previous **dc** command finishes. However, data collection, initiated by the **dc/s** command, may progress in parallel with the data collection initiated by the **dc** command. Moreover, several data collection processes initiated by the **dc/s** command may progress in parallel, providing they refer to different axes or axis groups. For example these two commands are executed serially:

```
dc/s 0, Array,500,1,FPOS(0)
dc/s 0, Array,500,1,FPOS(1)
```

While these commands are executed in parallel (unless the 0 and 1 axes belong to the same axis group):

```
dc/s 0,Array,500,1,FPOS(0)
dc/s 1,Array,500,1,FPOS(1)
```

The following is a full example of using the **dc/s** command:

```
GLOBAL REAL DC_DataX(2)(15000)

GLOBAL REAL DC_DataY(2)(15000)

GLOBAL REAL DC_DataA(2)(15000)

GLOBAL REAL DC_DataAsync(3)(10000)

ENABLE 0; ENABLE 1; ENABLE 2

!MASTER MPOS(1) = RPOS(0)+200

!SLAVE/p 1

DISP "DCN = %D, %D, %D", DCN(0), DCN(1), DCN(2)

!DC/s X, DC_DataX, 15000, 1, APOS(0), RPOS(0)  ! commented

DC/s Y, DC_DataY, 15000, 1, APOS(1), RPOS(1)

DC/s A, DC_DataA, 15000, 1, APOS(4), RPOS(4)

!DC DC_DataAsync,

DISP "DCN = %D, %D, %D", DCN(0), DCN(1), DCN(2)

wait 1000000

PTP/re X, 10

DC/s X, DC_DataX, 15000, 1, APOS(0), RPOS(0)   ! added

STOPDC
```

8.1.6 stopdc Command

Description

The **stopdc** command is also applicable to cyclic data collection.

An additional integer argument can be specified in the **stopdc** command; **stopdc** without arguments terminates the data collection immediately. The **stopdc** with an argument creates delayed termination of the data collection. For example:

`stopdc 50`	Collect additional 50 samples and then finish.

It should be noted that the syntax of **stopdc** command that terminates synchronous data collection has been changed. If your application uses synchronous data collection, slight changes may be required. To terminate a synchronous data collection that was commanded by a **dc/s** command, the **stopdc** command must also have switch **s**. For example:

`dc/s 0,ARR,50,10,VAR1`	Start synchronous data collection by axis 0 to a global real array ARR of size 50, sampling period 50, collect global variable VAR1.
`stopdc/s 0`	Terminate synchronous data collection by axis 0.

8.2 Position Event Generation (PEG)

The purpose of the Position Event Generator (PEG) mechanism is to generate accurate high speed position-based events.

Note	
	*For details of the structure of PEG engines in NT systems, see SPiiPlus NT PEG and MARK Operations Application Notes.*

The SPiiPlus NT/DC motion controllers provide the user with 6 identical PEG engine units. Each PEG engine can operate in one of the following two modes:

- **Incremental PEG mode** - provides the ability to generate a fix width pulse whenever a fixed position interval has passed, starting at a predefined start point and ending at a predefined end point
- **Random PEG mode** - provides the ability to control a PEG pulse and a four-bit vector at pre-defined positions, which are stored as a 256 member user-defined array

Each PEG engine can generate 1 PEG pulse: **PEGx_PULSE** (both in Incremental and Random modes) signal and 4 state signals: **PEGx_OUTy** - a 4-bit output vector, on each random position PEG event. State signals are set to a defined logical level or set to generate a pulse on transition, as defined by a 256 member **PEGx_OUT_ARRAY** integer array.

The PEG engines can be configured to be triggered by a position of any of the controller Encoders, with certain restrictions that result from the board's architecture.

The PEG engine outputs can be assigned to 10 physical interface outputs and the PEG pulse width and polarity are programmable.

8.2.1 Running Incremental PEG

Incremental PEG is defined by the first point P0, last point P1 and interval I. Incremental PEG generates the first pulse in position P0 and then repeats the pulse in positions P0+I, P0+2*I, P0+3*I, etc. The interval can be either positive or negative. The software terminates the PEG when the motor crosses position P1.

If the interval is small and the time between events is less than the controller cycle, additional pulses may be generated at points beyond position P1.

The Incremental PEG is set and activated by issuing the following commands:

1. Configuring specific PEG engines to specific encoders through the **ASSIGNPEG** command.
2. Assigning the physical output pins through the **ASSIGNPOUTS** command.
3. Setting Incremental PEG for the specific axis using **PEG_I** *axis, width, first_point, interval, last_point*
4. Waiting for **PEGREADY** to be '1' before activating PEG.
5. Starting PEG.

   PEG firing is initiated if the specific PEG engine is configured properly by **ASSIGNPEG** and **PEG_I** is issued. PEG continues to be fired periodically until *last_point* is reached.

   Incremental PEG is activated periodically starting from the *first_point* position event, and stops at the *last_point* position event. All pulses, including the last pulse, are of equal duration (*width*).

   PEG is generated only after *first_point* is reached. If *first_point* is not reached during the motion, PEG will not be generated.

   It is recommended that *first_point* position be the maximum current position for movement in the positive direction and the minimum current position for movement in the negative direction, before PEG engine activation.
6. Stopping the PEG motion with **STOPPEG** (optional)

   STOPPEG is a synchronous delayed command which stops the PEGs from firing.

   Since the Incremental PEG mechanism does not keep track of the direction of the movement, you may have to issue **STOPPEG** after *last_point* has been reached. This is needed in order to avoid unintentional PEG firing due to a rapid reversal of a movement's direction once *last_point* has been passed.

   If **STOPPEG** has been executed before *last_point* has been reached, you can use **STARTPEG** to continue PEG from the current position.

8.2.2 Running Random PEG

The Random PEG is set and activated by issuing the following commands:

1. Setting the event position array: **PEGx_POS_ARRAY**. The array has a maximum of 256 members.
2. Setting the output states array: **PEGx_OUT_ARRAY**. This array contains the state vectors which are to be issued per position event of the **PEGx_POS_ARRAY**.
3. Assigning the specific PEG engine to a specific encoder through the **ASSIGNPEG** command.
4. Assigning the physical output pins through the **ASSIGNPOUTS** command.
5. Setting Random PEG for the specific axis using **PEG_R** *axis, width, mode, first_index, last_index, POS_ARRAY, OUT_ARRAY.*
6. Waiting for **PEGREADY** to be '1' before activating PEG.
7. Starting PEG.

   PEG firing is initiated if the specific PEG engine is configured properly by **ASSIGNPEG** and **ASSIGNPOUTS** and **PEG_R** is issued. PEG continues to be fired upon position match until *last_point* is reached.
8. Stopping PEG with **STOPPEG** (optional)

   STOPPEG stops the PEGs from firing. If **STOPPEG** has been issued and the motion has not reached *last_point*, you can use **STARTPEG** to continue PEG from the current position.

It takes 800 **CTIME** cycles per axis to load the two arrays of position (**PEGx_POS_ARRAY**) and states (**PEGx_OUT_ARRAY**). As an example, in a liner application, a back and forth movement consists of the following stages:

1. Load arrays (if all 256 points are used it takes approximately 800 **CTIME** cycles, fewer points takes less time).
2. Activate PEG.
3. Start movement.

   The movement will cause the PEG signals to be fired each time a position match event occurs. Following the *last_point* event, the sequence above is repeated with the movement activated in the opposite direction.

8.2.3 ASSIGNPEG

The **ASSIGNPEG** function is used for engine-to-encoder assignment as well as for the additional digital outputs assignment for use as PEG state and PEG pulse outputs. The parameters are different from the original SPiiPlus definitions.

Syntax

ASSIGNPEG *axis, engines_to_encoders_code, gp_out_assign_code*

Arguments

axis	The axis index, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
engines_to_encoders_code	Bit code for engines-to-encoders mapping according to **Table 13** and **Table 14**
gp_out_assign_code	General Purpose outputs assignment to use as PEG state and PEG pulse outputs according to **Table 15** and **Table 16**.

Table 13 Mapping PEG Engines to Encoders (Servo Processor 0)

Bit Code	Servo Processor 0			
	Encoder 0(X)	Encoder 1(Y)	Encoder 2(A)	Encoder 3(B)
000 (default)	PEG0	PEG1	PEG2	no
001	PEG0	PEG1	no	PEG2
010	PEG0 PEG2	PEG1	no	no
011	PEG0	PEG1 PEG2	no	no
100	PEG0 PEG1 PEG2	no	no	no
101	no	PEG0 PEG1 PEG2	no	no

Table 14 Mapping PEG Engines to Encoders (Servo Processor 1) (page 1 of 2)

Bit Code	Servo Processor 1			
	Encoder 4(Z)	Encoder 5(T)	Encoder 6(C)	Encoder 7(D)
000 (default)	PEG4	PEG5	PEG6	no
001	PEG4	PEG5	no	PEG6
010	PEG4 PEG6	PEG5	no	no

Table 14 Mapping PEG Engines to Encoders (Servo Processor 1) (page 2 of 2)

Bit Code	Servo Processor 1			
	Encoder 4(Z)	Encoder 5(T)	Encoder 6(C)	Encoder 7(D)
011	PEG4	PEG5 PEG6	no	no
100	PEG4 PEG5 PEG6	no	no	no
101	no	PEG4 PEG5 PEG6	no	no

Note that in **Table 13** and **Table 14** the **Bit Code** affects all of the connectors in the row. For example, the **Bit Code**: 001 for an axis associated with Servo Processor 0 performs the following assignments:

PEG0 → Encoder 0

PEG1 → Encoder 1

PEG2 → Encoder 3

For an axis associated with Servo Processor 1 it performs the following assignments:

PEG4 → Encoder 4

PEG5 → Encoder 5

PEG6 → Encoder 7

Table 15 General Outputs Assignment for Use as PEG State and PEG Pulse Outputs (Servo Processor 0) (page 1 of 2)

Bit Code	Servo Processor 0			
	GP Out 0	GP Out 1	GP Out 2	GP Out 3
0000 (default)	GP Out 0	GP Out 1	GP Out 2	GP Out 3
0001	PEG_Pulse(0)	GP Out 1	GP Out 2	GP Out 3
0010	GP Out 0	PEG_Pulse(2)	GP Out 2	GP Out 3
0011	GP Out 0	GP Out 1	PEG_Pulse(1)	GP Out 3
0100	GP Out 0	GP Out 1	GP Out 2	Reserved
0101	GP Out 0	PEG_Pulse(2)	GP Out 2	Reserved
0110	PEG_Pulse(0)	GP Out 1	PEG_Pulse(1)	GP Out 3
0111	PEG_Pulse(0)	PEG_Pulse(2)	PEG_Pulse(1)	Reserved
1000	Reserved	Reserved	Reserved	Reserved
1001	Reserved	Reserved	Reserved	Reserved
1010	Reserved	Reserved	Reserved	Reserved

Table 15 General Outputs Assignment for Use as PEG State and PEG Pulse Outputs (Servo Processor 0) (page 2 of 2)

Bit Code	Servo Processor 0			
	GP Out 0	GP Out 1	GP Out 2	GP Out 3
1011	Reserved	Reserved	Reserved	Reserved
1100	Reserved	Reserved	Reserved	Reserved
1101	Reserved	Reserved	Reserved	Reserved
1110	Reserved	Reserved	Reserved	Reserved
1111	Reserved	Reserved	Reserved	Reserved

Table 16 General Outputs Assignment for Use as PEG State and PEG Pulse Outputs (Servo Processor 1)

Bit Code	Servo Processor 1			
	GP Out 4	GP Out 5	GP Out 6	GP Out 7
0000 (default)	GP Out 4	GP Out 5	GP Out 6	GP Out 7
0001	PEG_Pulse(4)	GP Out 5	GP Out 6	GP Out 7
0010	GP Out 4	PEG_Pulse(6)	GP Out 6	GP Out 7
0011	GP Out 4	GP Out 5	PEG_Pulse(5)	GP Out 7
0100	GP Out 4	GP Out 5	GP Out 6	Reserved
0101	GP Out 4	PEG_Pulse(6)	GP Out 6	Reserved
0110	PEG_Pulse(4)	GP Out 5	PEG_Pulse(5)	GP Out 7
0111	PEG_Pulse(4)	PEG_Pulse(6)	PEG_Pulse(5)	Reserved
1000	Reserved	Reserved	Reserved	Reserved
1001	Reserved	Reserved	Reserved	Reserved
1010	Reserved	Reserved	Reserved	Reserved
1011	Reserved	Reserved	Reserved	Reserved
1100	Reserved	Reserved	Reserved	Reserved
1101	Reserved	Reserved	Reserved	Reserved
1110	Reserved	Reserved	Reserved	Reserved
1111	Reserved	Reserved	Reserved	Reserved

Note that in **Table 15** and **Table 16** the **Bit Code** affects the entire row. For example, for an axis associated with Servo Processor 0, **0110** switches **PEG_Pulse(0)** to **GP Out 0** and **PEG_Pulse(2)** to **GP Out 1**.

The same **Bit Code** applied to an axis associated with Servo Processor 1 switches **PEG_Pulse(4)** to **GP Out 4** and **PEG_Pulse(5)** to **GP Out 6**.

All other GP Out are unchanged.

Comments

ASSIGNPEG is a blocking command in the sense that the ACSPL+ progam moves to the next line or command only after this command has been fully executed or an error is generated.

The ***axis*** parameter can be any of axes controlled by the same servo processor, the result will be the same.

8.2.4 ASSIGNPOUTS

Description

The **ASSIGNPOUTS** function is used for setting input pins assignment and mapping between **FGP_OUT** signals to the bits of the ACSPL+ **OUT(x)** variable, where **x** is the index that has been assigned to the controller in the network during System Configuration.

OUT is an integer array that can be used for reading or writing the current state of the General Purpose outputs - see *SPiiPlus ACSPL+ Command & Variable Reference Guide*.

Each PEG engine has 1 PEG pulse output and 4 state outputs for a total of 5 outputs per PEG engine and a total of 30 outputs for the whole PEG generator. The controller supports 10 physical output pins that can be assigned to the PEG generator. The user defines which 10 outputs (of the 30) of the PEG generator are assigned to the 10 available physical output pins. Some of the output pins are shared between the PEG and the HSSI.

The tables below define how each of the 30 outputs of the 6 PEG engines can be routed to the 10 physical output pins - 4 PEG out signals, 3 PEG state signals, and 3 HSSI signals. It needs to be noted that some of the signals cannot be routed to the physical pins.

Syntax

ASSIGNPOUTS *axis, output_index, bit_code*

Arguments

axis	The axis index, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
output_index	0 for **OUT_0**, 1 for **OUT_1**, ..., 9 for **OUT_9**
bit_code	Bit code for engine outputs to physical outputs mapping according to **Table 17** and **Table 18**.

Table 17 provides the mapping of the PEG engine outputs (30 signals) to the physical outputs of the first group of 5 out of the 10 output pins (Servo Processor 0), and **Table 18** provides to mapping to the second group of 5 out of 10 output pins (Servo Processor 1).

Table 17 Mapping of Engine Outputs to Physical Outputs (Servo Processor 0)

Bit Code	OUT_4 (HSSI1_DO)	OUT_3 (T_PEG)	OUT_2 (Z_PEG)	OUT_1 (Y_PEG)	OUT_0 (X_PEG)
000 (default)	HSSI1_DO	PEG5_pulse	PEG4_Pulse	PEG1_pulse	PEG0_Pulse
001	PEG0_OUT1	PEG0_OUT0	PEG2_Pulse	PEG1_OUT0	PEG4_OUT0
010	PEG2_OUT1	PEG2_OUT0	PEG1_OUT1	Reserved	Reserved
011	Reserved	Reserved	Reserved	Reserved	Reserved
100	Reserved	Reserved	Reserved	Reserved	Reserved
111	Reserved	FGP_OUT3	FGP_OUT2	FGP_OUT1	FGP_OUT0

Table 18 Mapping of Engine Outputs to Physical Outputs (Servo Processor 1)

Bit Code	OUT_9 HSSI0_CON	OUT_8 HSSI0_DO	OUT_7 X_STATE2	OUT_6 X_STATE1	OUT_5 X_STATE0
000 (default)	HSSI0_CON	HSSI0_DO	PEG0_OUT2	PEG0_OUT1	PEG0_OUT0
001	PEG0_OUT3	PEG0_OUT2	PEG2_OUT0	PEG1_OUT1	PEG1_OUT0
010	PEG2_OUT1	PEG6_OUT0	PEG6_Pulse	PEG5_Pulse	PEG4_Pulse
011	PEG6_OUT1	Reserved	PEG5_OUT1	PEG5_OUT0	Reserved
100	Reserved	Reserved	Reserved	Reserved	Reserved
111	Reserved	Reserved	FGP_OUT6	FGP_OUT5	FGP_OUT4

The **Bit Code: 111**, both for Servo Processor 0 and Servo Processor 1, is used for switching the physical output pins to Fast General Purpose Outputs: **FGP_OUT0** to **FGP_OUT6**. The state of the Fast General Purpose Outputs can be read or changed using the ACSPL+ **OUT(x)** variable. The Fast General Purpose Outputs are mapped as follows:

FGP_OUT0 is mapped to bit 16 of the ACSPL+ **OUT(x)** variable

FGP_OUT1 is mapped to bit 17 of the ACSPL+ **OUT(x)** variable

FGP_OUT2 is mapped to bit 18 of the ACSPL+ **OUT(x)** variable

FGP_OUT3 is mapped to bit 19 of the ACSPL+ **OUT(x)** variable

FGP_OUT4 is mapped to bit 20 of the ACSPL+ **OUT(x)** variable

FGP_OUT5 is mapped to bit 21 of the ACSPL+ **OUT(x)** variable

FGP_OUT6 is mapped to bit 22 of the ACSPL+ **OUT(x)** variable

Comments

ASSIGNPOUTS is a blocking command in the sense that the ACSPL+ progam moves to the next line or command only after this command has been fully executed or an error is generated.

Examples

The following examples illustrate using the **ASSIGNPOUTS** in order to use PEG outputs as GP outputs

Example 1:

ASSIGNPOUTS 0, 2, 0b111

This defines the **Z_PEG** output as **FGP_OUT2** and maps it to the bit 18 of the ACSPL+ **OUT** variable (see **Table 17**).

If you run, for example,

OUT(x).18 = 1

Where **x** is the index assigned to the controller during System Configuration, **FGP_OUT2** output will be activated.

Then if you run:

OUT(x).18 = 0

FGP_OUT2 will be deactivated.

Example 2:

ASSIGNPOUTS 4, 7, 0b111

This defines the **X_STATE2** output as **FGP_OUT6** and maps it to the bit 22 of the ACSPL+ **OUT** variable (see **Table 18**).

Note	*A separate command should be set for every GP output.*

8.2.5 STARTPEG

Description

The **STARTPEG** command initiates the PEG process on the specified axis. The command is used in both the Incremental and Random PEG modes.

Syntax

STARTPEG *axis*

Arguments

axis	The axis index, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.

Comments

STARTPEG is a blocking command in the sense that the ACSPL+ progam moves to the next line or command only after this command has been fully executed or an error is generated.

8.2.6 STOPPEG

Description

The **STOPPEG** command terminates the PEG process immediately on the specified axis. The command is used in both the Incremental and Random PEG modes.

Syntax

STOPPEG *axis*

Arguments

axis	The axis index, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.

Comments

STOPPEG is a blocking command in the sense that the ACSPL+ progam moves to the next line or command only after this command has been fully executed or an error is generated.

8.2.7 PEG_I

Description

The **PEG_I** command is used for setting the parameters for the Incremental PEG mode.

Syntax

PEG_I *axis, width, first_point, interval, last_point*

Arguments

axis	The axis index, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
width	Width of the Pulse.
first_point	First point for the PEG generation.
interval	The distance between PEG events.
last_point	Last point for PEG generation.

Comments

The parameters that can be set by the command are identical to those that can be set for non-NT SPiiPlus controllers except for *time-based-pulses* and *time-based-period* the setting of which is not supported for SPiiPlus NT/DC controllers.

Example

In this example PEG pulses are fired on axis 6 based on axis encoder 7

```
GLOBAL AXIS6
GLOBAL AXIS7
AXIS6=6 ! Axis assignment
AXIS7=7
```

```
assignpeg AXIS6, 0b001, 0b0
! Refer to Table 14: last column indicates that Encoder 7 is assigned to PEG6.
! 0b0 indicates that the digital outputs are at their default value according
! to Table 16, thus not used as PEG signals.

assignpouts AXIS6, 7, 0b010
! Axis6 being assigned. Output 7 is used. 0b010 indicates PEG6_Pulse is being
! fired (from Table 18).

ST:
peg_i AXIS6,0.5,-100,-200,-10000
TILL AST(AXIS6).#PEGREADY
!Wait till command executes and configuration is set, in order to ensure proper
!PEG engine's execution  prior to start of movement
ptp/e AXIS6,-12000
stoppeg AXIS6 ! Prevent PEG pulses' firing on the 'way back'
ptp/e AXIS6,0
goto ST

stop
```

8.2.8 PEG_R

Description

The **PEG_R** command is used for setting the parameters for the Random PEG mode.

Syntax

PEG_R *axis, width, mode, first_index, last_index, POS_ARRAY, OUT_ARRAY*

Arguments

axis	The axis index, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
width	Width of the Pulse.
mode	Output signal configuration according to **Table 19**.
first_index	Index of first entry in the array for PEG generation
last_index	Index of last entry in the array for PEG generation
POS_ARRAY	The Random Event Positions array, maximum of 256 members
OUT_ARRAY	The Outputs States array defining the four PEG output states, maximum of 256 members

Table 19 PEG Output Signal Configuration

Bit	Signal	Description	Default Value
0	Pulse polarity	0- Output-0 positive pulse 1- Output-0 negative pulse	'0'
1	State polarity	0- Output-0 positive state 1- Output-0 negative state	'0'
2-3	Output type	00- Output-0 three state 01- Output-0 State 10- Output-0 Pulse 11- Output-0 Pulse&State	'00'
4	Pulse polarity	0- Output-1 positive pulse 1- Output-1 negative pulse	'0'
5	State polarity	0- Output-1 positive state 1- Output-1 negative state	'0'
7-6	Output type	00- Output-1 three state 01- Output-1 State 10- Output-1 Pulse 11- Output-1 Pulse&State	'00'
8	Pulse polarity	0- Output-2 positive pulse 1- Output-2 negative pulse	'0'
9	State polarity	0- Output-2 positive state 1- Output-2 negative state	'0'
11-10	Output type	00- Output-2 three state 01- Output-2 State 10- Output-2 Pulse 11- Output-2 Pulse&State	'00'
12	Pulse polarity	0- Output-3 positive pulse 1- Output-3 negative pulse	'0'
13	State polarity	0- Output-3 positive state 1- Output-3 negative state	'0'
15-14	Output type	00- Output-3 three state 01- Output-3 State 10- Output-3 Pulse 11- Output-3 Pulse&State	"00"

Comments

The parameters that can be set by the command differ from those that can be set for non-NT SPiiPlus controllers with the addition of the new *first_index* and *last_index* parameters. The setting of parameters: *time-based-pulses* and *time-based-period*, however, which can be set for non-NT SPiiPlus controllers is not supported.

Example

In this example PEG pulses are fired on axes 0,1,2 according to encoder of axis 0

```
GLOBAL AXIS0
GLOBAL AXIS1
GLOBAL AXIS2

GLOBAL REAL ARR(16) !Definition of a 16 point position array
ARR(0)=100;ARR(1)=150;ARR(2)= 200;ARR(3)=250;
ARR(4)=300;ARR(5)=350;ARR(6)=400;ARR(7)= 450;
ARR(8)=500;ARR(9)=550;ARR(10)= 600;ARR(11)=650;
ARR(12)=700;ARR(13)=750;ARR(14)=800;ARR(15)=1000;

GLOBAL INT STAT(16) !Definition of 16 point state array. The values define
PEGx_OUTy
STAT(0)=0b0001;STAT(1)=0b0100;STAT(2)=0b1111;STAT(3)=0b1110;
STAT(4)=0b0101;STAT(5)=0b0001;STAT(6)=0b0100;STAT(7)=0b1111;
STAT(8)=0b1110;STAT(9)=0b0101;STAT(10)=0b0001;STAT(11)=0b0100;
STAT(12)=0b1111;STAT(13)=0b0001;STAT(14)=0b0100;STAT(15)=0b1111;

AXIS0=0
AXIS1=1
AXIS2=2

assignpeg AXIS0, 0b100, 0b0

assignpouts AXIS0, 0, 0b000 !Assign bit code 000 from Table 17 to AXIS0, at Pin0
assignpouts AXIS1, 1, 0b000
assignpouts AXIS2, 2, 0b001

ST:
peg_r AXIS0,0.5,0x4444,0,15,ARR,STAT !Activate random PEG for axis 0
       peg_r AXIS1,0.5,0x4444,0,15,ARR,STAT
       peg_r AXIS2,0.5,0x4444,0,15,ARR,STAT

TILL AST(AXIS0).#PEGREADY
!Wait till command executes and configuration is set, in order to ensure proper
!PEG engine's execution  prior to start of movement
       TILL AST(AXIS1).#PEGREADY
       TILL AST(AXIS2).#PEGREADY

       ptp/e AXIS0,5000
       stoppeg AXIS0 ! Prevent PEG pulses' firing on the 'way back'
       ptp/e AXIS0,0
goto ST

stop
```

8.3 Sin-Cos Encoder Multiplier Configuration

The sin-cos encoders are encoders with analog outputs.

For convenient modification of encoder related variables, use the SPiiPlus MMI Application Studio: **Toolbox → Setup →System and Faults Configurator**, see *SPiiPlus MMI Application Studio User Guide* for details..

8.3.1 Sin-Cos Encoder Multiplier

The sin-cos encoder multiplier provides a combination of high speed and high resolution that cannot be achieved with an external encoder multiplier producing a digital quadrature signal.

Sin-cos encoder multiplier is an optional feature. You need to specify the number of sin-cos encoder multipliers when you order the controller.

8.3.1.1 Technical Data

- Maximum multiplication factor: 65536 counts per encoder sine signal period.
- Maximum velocity: $42*10^9$ counts/sec or $641*10^3$ sine periods/sec.
- Maximum acceleration: $65536*10^8$ counts/sec^2 or 10^8 sine periods/sec^2.
- The encoder-related controller features: **index**, **Mark**, and **PEG**, do not support the full resolution of the multiplier. Resolution for these features is limited to 4 counts per encoder sine signal period.

8.3.1.2 Configuring the Sin-Cos Multiplier

You specify the sin-cos encoder multiplier as a feedback source using the ACSPL+ **E_TYPE** variable (see *SPiiPlus Command & Variable Reference Guide*). The variable is an array sized according to the number of axes. If an element in **E_TYPE** is set to 4, the corresponding axis will use the sin-cos encoder multiplier as a feedback source. You can select any axis to use the sin-cos encoder multiplier, but the total number of multipliers is limited to the number ordered with the controller (the allowed number is hardwired in the PAL). If you try to select more multipliers than allowed, the controller issues error 1148.

The ACSPL+ variable **E_SCMUL** (see *SPiiPlus Command & Variable Reference Guide*) specifies the desired value of multiplication as a power of 2. The maximum value of 16 corresponds to a multiplication of 65536 = 216. The minimum value of 2 corresponds to a multiplication of 4 = 22.

An axis that uses the sin-cos encoder multiplier is not different from any other axis in the controller. Any motion can involve the multiplier axis. The multiplier and non-multiplier axes can be used simultaneously in one multi-axis motion.

The following example shows how the encoder multiplier affects the **EFAC** (see **Section 7.2 - User-Defined Units**) calculation:

Assume the encoder has 250 lines per mm and you assign **E_SCMUL** the value 10, which provides multiplication x1024 (i.e., 210). The desired programming unit is millimeter. In this case you have to specify **EFAC** as:

1/(250*1024) = 0.00000390625

Encoder-related features: index, Mark, PEG use the same user unit. However, actual resolution of these functions is lower than the resolution of encoder feedback. As mentioned above, the resolution of index, mark and PEG corresponds to 4 counts per encoder sine period. In the former example the resolution will be 1/(250*4) = 0.001mm.

The controller continuously checks the integrity of the encoder multiplier feedback. If any error occurs, the controller activates the Encoder Error fault.

8.3.2 Sin-Cos Filter

The sin-cos filter reduces the electrical noise level of the encoder signal, thereby reducing jitter at standstill position.

Warning 	***Change this parameter only with the motor disabled.***

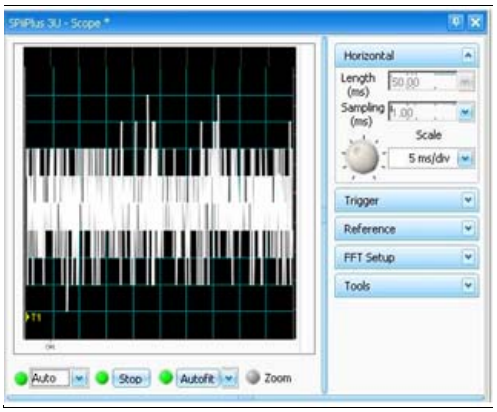
The ACSPL+ **SLEMOS** variable (see *SPiiPlus Command & Variable Reference Guide*) sets the order of the filter. The range is 0 to 4 (default is 0). When **SLEMOS = 0**, the filter is not active.

8.3.2.1 Filter Limitations

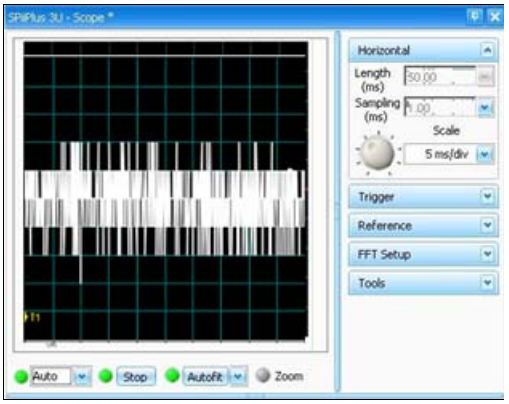
1. The filter is active only when the motor is not moving (**GPHASE=0**).
2. The filter may be used only when the encoder counter is in the range of ±231counts. (If the motor moves in one direction for a long time, causing the counter to overflow, the position reading will be incorrect.)

8.3.2.2 Test Results

The following figure shows the position error at standstill of a rotary motor with sin-cos encoder. Encoder resolution is 2000 lines per revolution.



(A) E_SCMUL=13, SLEMOS=0



(B) E_SCMUL=13, SLEMOS=4

(A) shows the position error with a multiplier of 8192 and the filter inactive (**SLEMOS=0**).

(B) shows the position error with a multiplier of 8192 and with the filter active (**SLEMOS=4**).

It can be seen that jitter has been reduced by almost one half and a jitter of ±2 counts is achieved.

8.4 Interrupts

Typically, you work with the SPiiPlus interrupts using the SPiiPlus C Library and do not need the low-level details specified in this section. Refer to the *SPiiPlus C Library Reference* document for explanation of the interrupt managing functions.

Reading any interrupt status register also resets the bits in the register. For hardware interrupts this enables the interrupt to be generated again. For software interrupts this is not enough; the host drive must write zero also to an interrupt tag in order to enable the corresponding software interrupt to be generated again.

8.4.1 Hardware Interrupts

The PCI card generates interrupts through an interrupt line.

To find out the exact source of the interrupt the host drive must read two interrupt status registers. Because one interrupt may be caused by two or more sources, the host drive must always read both registers.

The hardware interrupt status registers reside at address Offset+0x1000 and contain the following bits:

Bit	Description
3	PEG 0
4	PEG 1
5	PEG 3
6	PEG 4

Bit	Description
7	MARK 1 0
8	M2ARK 0
9	MARK 1 2
10	M2ARK 2
11	MARK 1 3
12	M2ARK 3
13	MARK 1 4
14	M2ARK 4
15	Emergency Stop

8.4.2 Software Interrupts

The software interrupt status register resides at address Offset+0x1002 and contains the following bits:

Bit	Description
0	Physical motion end
1	Logical motion end
2	Motion failure (Motion interruption due to a fault)
3	Motor failure (Motor disable due to a fault)
4	Program termination
5	Command execution
6	ACSPL+ interrupt (by **INTERRUPT** command)
7	Digital input
14	Controller cycle (the controller raises this interrupt in the beginning of each controller cycle)
15	Communication interrupt (the controller raises this interrupt after sending a complete message to the FIFO)

Reading any interrupt status register also resets the bits in the register. For hardware interrupts this enables the interrupt to be generated again. For software interrupts this is not enough: the host drive must write zero also to an interrupt tag in order to enable the corresponding software interrupt to be generated again.

Software interrupt tags occupy the first 8 words of **DPR**, addresses from Offset+0x2000 to Offset+0x2007. When a software interrupt occurs, the corresponding tag contains detailed information about the interrupt source. For example, the tag of the Physical Motion End interrupt specifies the axes that caused the interrupt. When a specific software interrupt has occurred, the next interrupt of the same type can be generated only after the host drive reads both interrupt status register and writes zero to the corresponding tag.

8.4.3 Software Interrupt Tags

The following tags are available:

Bit	Description
0	Details of Physical motion end interrupt. Bit 0 specifies that axis 0 has finished, bit 1 - axis 1, and so on.
1	Details of Logical motion end interrupt. Bit 0 specifies that axis 0 has finished, bit 1 - axis 1, and so on.
2	Details of Motion failure interrupt. Bit 0 specifies that axis 0 has failed, bit 1 - axis 1, and so on.
3	Details of Motor failure interrupt. Bit 0 specifies that axis 0 has failed, bit 1 - axis 1, and so on.
4	Details of Program termination interrupt. Bit 0 specifies that buffer 0 has finished, bit 1 - buffer 1, and so on.
5	Details of Command execution interrupt (dynamic buffer only). Bit 0 specifies event in buffer 0, bit 1 - buffer 1, and so on.
6	Details of ACSPL+ interrupt (by INTERRUPT command). Bit 0 specifies interrupts from buffer 0, bit 1 - buffer 1, and so on.
7	Details of Digital input interrupt. Bit 0 specifies interrupts from input 0, bit 1 - input 1, and so on.

8.4.4 Interrupt Configuration Variables

The following ACSPL+ variables enable/disable interrupts from a specific source:

- **IENA** - Scalar variable that enables/disables all interrupts that belong to a specific interrupt status bit.
- **ISENA** - Array that enables/disables interrupts within a specific interrupt status bit. Each elements corresponds to one interrupt status bit and specifies which axes or buffers or inputs are enabled to cause interrupt.

8.4.4.1 IENA Variable

The **IENA** variable contains the following bits:

Bit	Description
3	Enable PEG axis 0 interrupt
4	Enable PEG axis 0 interrupt
5	Enable PEG axis 2 interrupt
6	Enable PEG axis 2 interrupt
7	Enable MARK 1axis 0 interrupt
8	Enable M2ARKaxis 0 interrupt
9	Enable MARK 1 axis 1 interrupt
10	Enable M2ARK axis 1 interrupt
11	Enable MARK 1 axis 2 interrupt

Bit	Description
12	Enable M2ARK axis 2 interrupt
13	Enable MARK 1 axis 3 interrupt
14	Enable M2ARK axis 3 interrupt
15	Enable Emergency Stop interrupt
16	Enable Physical motion end interrupt
17	Enable Logical motion end interrupt
18	Enable Motion failure (Motion interruption due to a fault) interrupt
19	Enable Motor failure (Motor disable due to a fault) interrupt
20	Enable Program termination interrupt
21	Enable Dynamic buffer interrupt
22	Enable ACSPL+ interrupt (by **INTERRUPT** command)
23	Enable Digital input interrupt

8.4.4.2 ISENA Variable

The **ISENA** variable is an array containing the following elements:

Bit	Description
0	Controls Physical motion end interrupt. Bit 0 enables interrupts from axis 0, bit 1 - axis 1, and so on.
1	Controls Logical motion end interrupt. Bit 0 enables interrupts from axis 0, bit 1 - axis 1, and so on.
2	Controls Motion failure interrupt. Bit 0 enables interrupts from axis 0, bit 1 - axis 1, and so on.
3	Controls Motor failure interrupt. Bit 0 enables interrupts from axis 0, bit 1 - axis 1, and so on.
4	Controls Program termination interrupt. Bit 0 enables interrupts from buffer 0, bit 1 - buffer 1, and so on.
5	Controls Command execution interrupt (dynamic buffer only). Bit 0 enables interrupts from buffer 0, bit 1 - buffer 1, and so on.
6	Controls ACSPL+ interrupt (by **INTERRUPT** command). Bit 0 enables interrupts from Buffer 0, bit 1 - buffer 1, and so on.
7	Controls Digital input interrupt. Bit 0 enables interrupts from input 0, bit 1 - input 1, and so on.

8.5 Dynamic Braking

The dynamic brake reduces the motor runoff if the motor becomes disabled during motion. In dynamic braking the controller short-circuits the motor armature.

If dynamic braking is enabled for an axis, the controller applies the braking when the feedback velocity falls below the value specified by the **VELBRK** parameter (default - 0).

The **MFLAGS.#BRAKE** bit enables dynamic braking (default - off).

8.6 Constant Current Mode

The Constant Current mode provides extra safety. When the mode is activated, and the emergency stop signal is on, the motor is kept at a standstill by the drive.

- The function, **setconf (133, 1, 1)** enables constant current mode for axis 1. The function, **setconf (133, 1, 0)** disables constant current mode for axis 1.
- To retrieve the Constant Current mode status for axis 1, **getconf (133, 1)** is used. It retrieves a non-zero value if Constant Current mode is on and zero if Constant Current mode is off.

The drive activates constant current mode only if all of the following conditions are true:

- Emergency Stop (**ES**) signal is on.
- All axes are disabled.
- Dynamic brake mode is off (**MFLAGS(Axis).#BRAKE = 0**).

Only once these conditions are true can the constant current mode be enabled by the function **setconf (133, 1, 1)**. If any of the conditions is changed, it will deactivate the constant current mode.

To set/get the current level used for constant current mode, the following **setconf**/**getconf** parameters are used:

- **setconf (130, {1|5}, current level)**, where current level defines that the constant current level as a percentage of the maximum current for the specified axis (1 or 5 only). The default maximum value that can be set is 16% for axis 1 and 30% for axis 5 (this value can be changed, see description of key 131 below).
- **getconf (130, {1|5})** retrieves the present value of current level for the specified axis.

The default value of current level is 0, so it must be defined before constant current mode can be used.

To set/get the maximum value that can be set with **setconf (130, . . .)**, the following **setconf**/**getconf** parameters are used:

- **setconf (131, {1|5}, max_current_level)**, where **max_current_level** defines the maximum allowable value of the current level as a percentage of the maximum current for the specified axis (1 or 5 only).
- **getconf (131, {1|5})** retrieves the present maximum allowable current level for the specified axis.

The following ACSPL+ program illustrates how to implement constant current mode:

```
! Constant current mode implementation
on S_FAULT.#ES
    disableall
    wait 100
    MFLAGS(1).#BRAKE = 0; MFLAGS(5).#BRAKE = 0 !Disable Brake Mode for
                                               !axes 1, 5
    setconf(130, 1, 10) ! set constant current level (10%) for axis 1
    setconf(130, 5, 20) ! set constant current level (20%) for axis 5
    setconf(133, 1, 1)  ! enable constant current mode
ret
on ^S_FAULT.#ES | S_FAULT.#DRIVE
    setconf(133, 1, 0)  ! disable constant current mode
ret
```

8.7 Hall Sensor Commutation

- Hall sensor commutation requires a first time adjustment from the commutation dialog in the SPiiPlus MMI Application Studio Adjuster wizard (see *SPiiPlus MMI Application Studio User Guide* for details).
- In subsequent power-ups, the motor will start moving according to the Hall sensors until it encounters the first change in the Hall sensors. At this point, commutation will switch to full sinusoidal commutation.
- The quality of commutation relies on the physical alignment of the Hall sensors relative to the magnetic field of the motor, and in most cases is done very accurately by the motor manufacturers. Using this method, factors like high friction, vertical load, etc. have no effect on the commutation quality.
- The connection sequence of the three Hall sensors is not important. Simply verify that the three sensors are connected, and that the Hall counter counts 0,1,2,3,4,5. It does not matter if the Hall counters count is opposite of the encoder. This situation is identified and dealt with during the initial adjustment.

8.7.1 Hall Support Parameters and Functions

Table 20 Parameters and Functions for Hall Support (page 1 of 2)

Parameter	Description
SLCHALL	Holds the Hall shift. The Adjuster commutation program calculates this parameter and saves it. Do not change this parameter manually.
MFLAGS.27	If this bit = 1, commutation will be based on the Hall sensors.
MFLAGS.28	Hall direction inversion. The Adjuster commutation program calculates this parameter and saves it. Do not change this parameter manually. 1 = Controller inverts Hall direction.

Table 20 Parameters and Functions for Hall Support (page 2 of 2)

Parameter	Description
GETCONF(110, Index)	Returns the Hall counter of the axis specified by Index. The Hall direction bit is not taken into account. This function is used by the Adjuster commutation program.
GETCONF(111, Index)	This function is used by the Adjuster program during commutation.

Note	
	1. *After hardware reset, if the Hall commutation was successful, the firmware automatically sets bit **MFLAGS.9** = 1, if **MFLAGS.27** = 1.* 2. *For proper Hall commutation, the encoder resolution and number of poles should be defined correctly. The current loop should be adjusted.*

8.7.2 SLSTHALL Variable

SLSTHALL	Servo Loop, State of Hall sensor
Size	Array of 8 elements
Type	Integer
Accesibility	Read
Comments	Range 0..5

8.7.3 Hall Verification Procedure

The connection sequence of the three Hall sensors is not important. You only have to verify that the three Hall sensors are connected and the Hall counter counts 0,1,2,3,4,5 or vice versa. It does not matter if the Hall counters count in an order that is the reverse of the encoder; this will be identified and taken care of during the initial adjustment. The procedure for Hall verification is as follows:

1. Open the SPiiPlus MMI Application Studio **Scope**.
2. For one of MMI Scope channels select the Feedback Position.
3. For another channel select the ACSPL+ variable **SLSTHALL(axis)**.
4. Rotate the motor by hand. **Figure 19** illustrates the required result. Rotate the motor by hand in opposite direction. The expected result is illustrated in **Figure 20**.

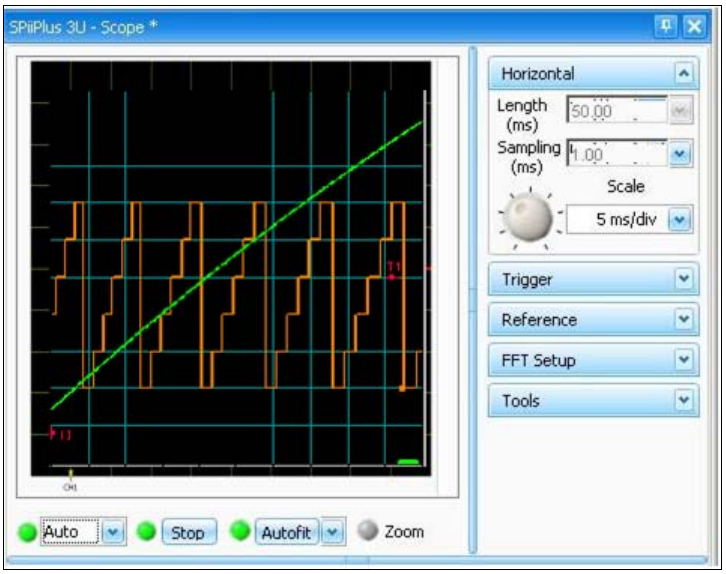


Figure 19 Resultant Track of Motor Motion

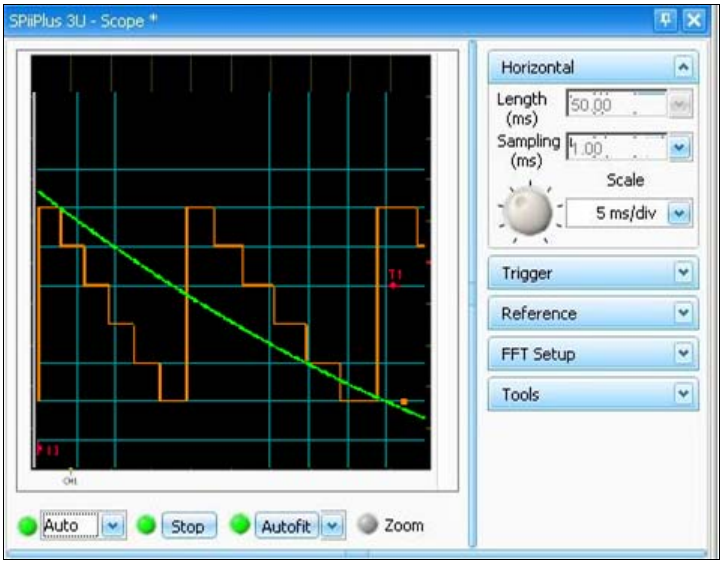


Figure 20 Resultant Track of Motor Motion in Opposite Direction

8.8 Communicating with the SPiiPlus C Library

This section provides an overview of the SPiiPlus C Library communication channels.

8.8.1 Remote Connection

The C Library installed on a specific computer supports not only applications running on the same computer, but also applications on remote computers. The only requirement is for a TCP/IP connection between the computers.

Figure 21 illustrates the simultaneous connection of two local and one remote applications.

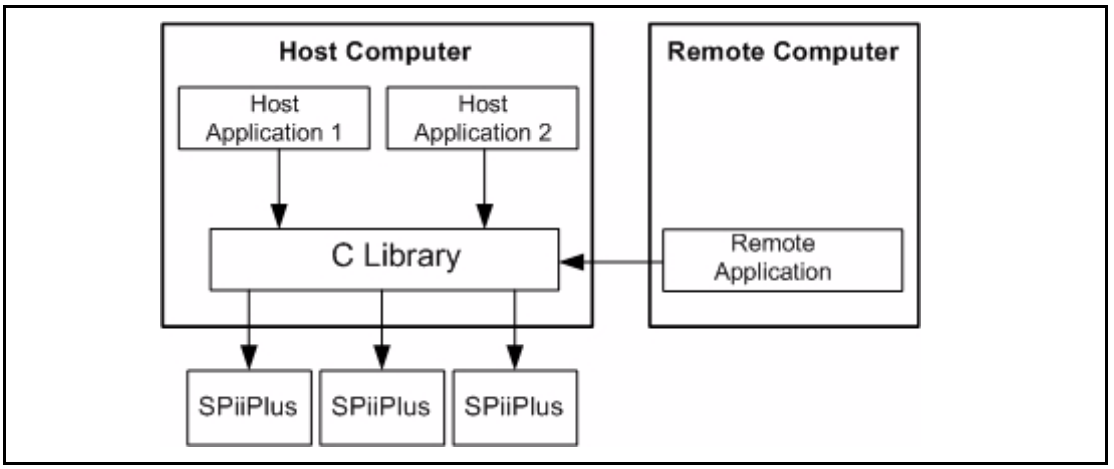


Figure 21 Simultaneous Connection for Remote Support

8.8.2 Callbacks in all Communication Channels

The Callback mechanism provides the fastest response to the controller events. The implementation is transparent for user applications. In all communication channels, the callback API consists of the functions, **SetCallback** and **SetCallbackExt**.

8.8.2.1 Timing

Callback operations include sending/receiving a message that requires much more time. Specific rates depend on the communication channel rate.

From the viewpoint of callback mechanism, all communication channels are functionally equivalent, but differ in timing.

8.8.2.2 Software Interrupts

The following interrupts are generated by the controller firmware and therefore are called software interrupts:

- Physical motion end
- Logical motion end
- Motion failure (Motion interruption due to a fault)
- Motor failure (Motor disable due to a fault)
- Program termination
- Command execution
- ACSPL+ interrupt (by **interrupt** command)
- Digital input
- Logical motion start
- Motion phase change
- Trigger

8.8.2.3 Hardware Interrupts

The following interrupts are generated by the controller hardware and therefore are called hardware interrupts:

- Emergency stop
- Mark 1 (axes 0, 1, 2, 3)
- Mark 2 (axes 0, 1, 2, 3)
- PEG (axes 0, 1, 2, 3)

Table 21 describes hardware interrupt callback conditions in different communication channels.

Table 21 Hardware Interrupt Callback Conditions

Callback	Condition of Alert Message
Emergency stop	The message is sent when bit **S_FAULT.#ES** changes from zero to one. The message is disabled if **S_FMASK.#ES** is zero.
Mark 1 and Mark 2	The message is sent when corresponding **IST.#MARK** or **IST.#MARK2** bit changes from zero to one.
PEG	The message is sent when the corresponding AST.#PEG bit changes from one to zero.

8.8.3 TCP/IP Port Assignment for Remote Connection

The C Library installed on a specific computer supports not only applications running on the same computer, but also applications on remote computers. The only requirement is for a TCP/IP connection between the computers.

8.8.3.1 TCP/IP Port Assignment

Note	
	*For a description of all the UMD functions see SPiiPlus Utilities User Guide.*

To establish a remote connection using TCP/IP, select **Enable Access from Remote Application** on the User Mode Driver (UMD) **Remote Connection** tab.

If the default port is not busy, no communication error messages are encountered, and no problem is anticipated. In this case, use the function **acsc_SetServerExt** with the **ACSC_DEFAULT_REMOTE_PORT** parameter to set the remote port address from the user application.

If the default port (9999) is busy, the UMD will return the following error message:

> Requested port 9999 is in use by another application. Select another port in the Remote Connection tab.

In this case, proceed as follows:

1. From the **Remote Connection** tab, select **Change** from the **Remote Port Connection** list.
2. Enter the remote port address in the **Enter valid port number** dialog.
3. From the **Remote Connection** tab, select **Enable Access from Remote Application**. As soon as the check box is selected, communication with the remote port is attempted. If communication does not succeed, the following error message appears:

   Requested port [**port number**] is in use by another application. Select another port in the Remote Connection tab.

4. Repeat Steps 1-3 until communications are established. When communications are successfully established, the UMD stores the settings.
5. Click **OK** and then **Close**.
6. In the user application, use the function **acsc_SetServerExt** and specify the same port number that was entered in the UMD GUI.

Note	
	*Every time the User Mode Driver (UMD) initializes, the availability of the specified port is checked. If the system configuration or port number have changed, the UMD generates an error message and the* **Enable Access from Remote Application** *on the UMD* ***Remote Connection*** *tab check box will be cleared.*

8.8.3.2 Disabling Remote UMD Connections

After installation, the remote connection is disabled.

To enable the remote connection, select **Enable Access from Remote Application** on the UMD **Remote Connection** tab. The remote application can now connect to the UMD, until it is disabled.

Disable the remote connection as follows:

1. Clear **Enable Access from Remote Application** on the UMD **Remote Connection** tab.
2. Restart the UMD for the changes to take effect.

8.8.3.3 UMD Log Types

The UMD logs constantly at run-time. The data is stored in binary format in an internal cyclic buffer and is translated to text just before it is written to file.

You may choose one of two mutually exclusive log types:

- **Dump on Request** – all the binary data that is stored in the internal binary buffer and is flushed by explicit request to the file, see **acsc_FlushLogFile**.
- **Continuous** – this is a background thread that takes care of periodic file updates. It reads the binary buffer and performs text formatting to the file.

Perform the **Dump on Request** log type as follows:

1. From the UMB **Log Settings** tab, select **Dump on request** as the **Log File Mode**.
2. Select the Log file path

Refer to **Figure 22** and note that **Dump** appears on the **Execution** command and that **Maximum logging** time is disabled.

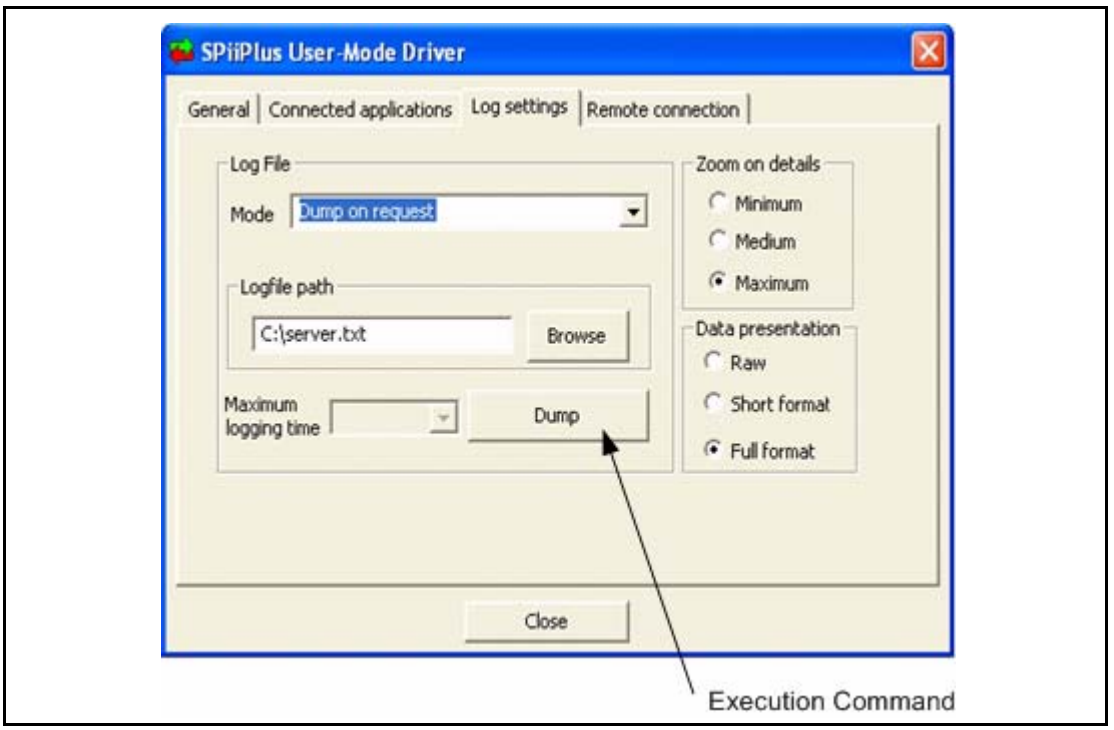


Figure 22 UMD Log Settings - Dump on Request

Perform a **Continuous Log File** type as follows:

1. From the UMD **Log Settings** tab, select **Continuous** as the **Log File Mode**.
2. Select the Log file path

   Refer to **Figure 23** and note that **Start Log** appears on the **Execution** command and that **Maximum logging time** is enabled.
3. Set **Maximum logging time**. Possibilities range from one hour to One week. The **Infinite** setting is not recommended because of a possible disk overflow situation.

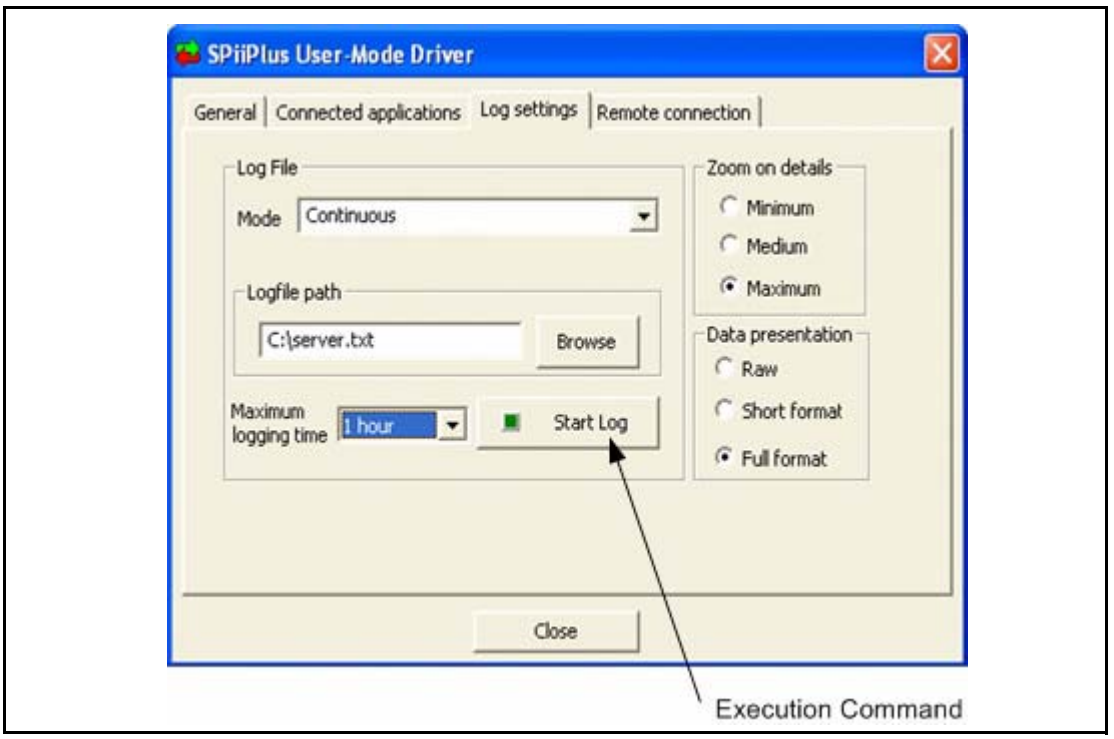


Figure 23 UMD Log Settings - Continuous

After clicking **Start Log,** and the log is active, the **Execution** command appears as **Stop Log**.

8.8.3.4 Unloading the UMD from Memory

Unload the UMD from memory as follows:

1. Right click the UMD icon in the status area
2. Click **Unload**.
3. Click **OK** on the confirmation message. The UMD is then unloaded from the memory.

8.9 Communicating with 3rd Party Devices

Another use of the **setconf** and **getconf** functions (see *SPiiPlus Command & Variable Reference Guide*) relates to external communication channels. The functions are used to facilitate the following tasks:

- Connection to user panel (teach pendant, console) that make use of special communication protocols (MODBUS, etc.)
- Connection to devices with special interfaces (laser interferometer, intelligent drives, etc.)

Note	
	*Currently only serial channels can be connected to devices such as these. All functions described below are implemented for RS232 serial channels only.*

8.9.1 Channel Configuration Report

The **#CC** terminal command reports the current configuration of communication channels.

The controller responds with a table that specifies the configuration for each supported communication channel.

Note	
	*For any specific hardware configuration, not all listed channels may be physically available.*

The following example illustrates the report:

```
#CC
Channel Type      Command/Special
 1      Serial    Command Rate:115200
 2      Serial    Special Rate:9600 Parity:Even
10      TCP/IP    Command Connection:point-to-point Peer:10.0.0.96
```

The report describes the following channels:

- 1 - COM1, in command interpretation mode, rate 115200 kbps, no special options, i.e. no parity bit, normal stop bit, no break.
- 2 - COM2, in special mode, rate 9600 kbps, even parity bit, normal stop bit, no break.
- 10 – Ethernet (TCP/IP protocol), in command interpretation mode, point-to-point connection, currently connected to peer address 10.0.0.96.

8.9.2 Assigning COM Channel for Special Input

The **setconf** function with key 302 assigns a communication channel for special input as follows:

setconf(302, channel, {0|1})

Where:

channel	Indicates the assigned COM channel, it can be one of the following values: ❑ 1 – COM1 ❑ 2 – COM2

The values that can be used in connection with the channel are:

- 1 – assigns the channel for special input
- 0 – returns the channel to regular command processing.

If a channel is assigned for special input, the controller does not process commands from this channel. Any input from this channel can be processed only by the **inp** function (see **Section 8.9.4 - inp Function**). Output to the channel is provided by regular **disp** and **send** commands (see *SPiiPlus Command & Variable Reference Guide*).

Note	
	*While a channel can be set to special communication mode by the* **setconf** *command through the same communication channel, the channel cannot be returned to normal mode through the same channel. The reason is that in special communication mode the channel does not process commands and therefore cannot execute* **setconf**. *You have to use another channel to return the channel to the regular command processing mode*

The **getconf** function with key 302 retrieves the state of a communication channel as follows:

getconf(302, channel)

The function returns:

- 0 – if the channel is in normal command-processing mode
- 1 – if the channel is in special-communication mode

8.9.3 Setting Communication Parameters

Currently, the **setconf** function with key 303 is supported only for serial channels (channel number 1 or 2). The function configures the communication rate for the specified channel as follows:

setconf(303, channel, baud)

Where:

channel	Indicates the assigned COM channel, it can be one of the following values: ❑ 1 – COM1 ❑ 2 – COM2
baud	Specifies the communication rate in the channel.

The most popular communication rates are the following (kbps):

- ❑ 115200
- ❑ 57600
- ❑ 19200
- ❑ 9600
- ❑ 4800
- ❑ 2400
- ❑ 1200
- ❑ 600
- ❑ 300

The function **getconf** with key 303 retrieves the communication rate of the specified channel as follows:

getconf(303, channel)

8.9.4 inp Function

Description

The **inp** function reads data characters from the specified channel and stores them into integer array.

Syntax

int inp*(channel [,variable] [,start_index] [,number] [,timeout])*

Aguments

channel	Indicates the assigned COM channel, it can be one of the following values: ❑ 1 – COM1 ❑ 2 – COM2
variable	Name of user-defined integer array.
start_index	Index of the array.
number	Specifies number of characters to read.
timeout	Maximum waiting time (in milliseconds) for response from the channel.

The function returns the number of actually assigned characters.

If the **variable** argument is omitted, the function dumps all characters received before in the channel. If the **variable** argument is specified, the function accepts one or more characters from the specified channel and assigns them to the sequential elements of the **variable** array.

Each ASCII character is represented by its numerical value and is stored in a separate element of the array.

If **start_index** is specified, the first received character is assigned to the array element with the specified index. If **start_index** is omitted, the assignment starts from the first element of the array.

If **number** is specified, the function does not return until the exact number of characters is received. Any received character, including carriage return and other non-printable characters, is stored in the array. In this case the function return value is strictly equal to **number**.

If **number** is omitted, the function continues receiving characters until the last element of array is assigned or the carriage return character is received. The received carriage return is not stored in the array. The function return indicates the number of assigned array elements.

If **timeout** is specified, the function waits for input not more than the specified number of milliseconds. If **timeout** is omitted, the waiting time is not limited.

8.9.5 String Handling Commands

Text handling commands are used for sending text to the host for purposes of displaying on the monitor or for recording in a log.

8.9.5.1 disp Command

Description

The **disp** command builds an ASCII output string and sends it to a communication channel. Upon receipt, the host displays the message on the monitor.

Syntax

disp *expression | "string" [,expression | "string". . .]*

Arguments

expression	ACSPL+ expression (can be a single variable)
string	A string, which must be enclosed with double quotation marks.

A string argument has the format of:

"[text] [escape-sequence] [format-specifier] . . ."

Where:

text	Any ASCII text characters
escape-sequence	The escape-sequence can be: ❑ \r - Carriage return 0x0d ❑ \n - New line 0x0a ❑ \t - Horizontal tabulation 0x09 ❑ \xHH - Any character. The two hexadecimal digits, HH, represent the character's ASCII code.
format-specifier	The format specification syntax adheres to a restricted version of the C language syntax: **% [width] [.precision] type** ❑ width - Optional number that specifies the minimum number of characters in the output. ❑ .precision - Optional number that specifies the maximum number of characters printed for all or part of the output field, or the minimum number of digits printed for integer values ❑ type - Required character that determines whether the associated argument is interpreted as a character, a string, or a number (see **Table 22**).

Table 22 String Format Type (page 1 of 2)

Character	Output Format
d	Signed decimal integer.
I	Signed decimal integer.
o	Unsigned octal integer.
u	Unsigned decimal integer.
x	Unsigned hexadecimal integer, using "abcdef."
X	Unsigned hexadecimal integer, using "ABCDEF."
e	Signed value having the format: **[-]d.dddd e [sign]ddd** Where: ❑ d is a single decimal digit ❑ dddd is one or more decimal digits ❑ ddd is exactly three decimal digits ❑ e indicates exponent ❑ sign is + or -.
E	Identical to the **e** format except that **E** rather than **e** indicates the exponent.

Table 22 String Format Type (page 2 of 2)

Character	Output Format
f	Signed value having the format: **[-]dddd.dddd** Where dddd is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision.
g	Signed value printed in **f** or **e** format, whichever is more compact for the given value and precision. The e format is used only when the exponent of the value is less than -4 or greater than or equal to the precision argument. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.
G	Identical to the **g** format, except that **E**, rather than **e**, introduces the exponent (where appropriate).

If an input string argument contains n format specifiers, the specifiers apply to the n subsequent expression arguments.

The **disp** command processes the arguments from left to right. The processing is as follows:

- Expressions:

  The expression is evaluated and the ASCII representation of the result is placed in the output string. The format of the result is determined by a formatting specification (if any) in the input string.

- Input strings:

  Text is sent as-is to the output string. Escape sequences are replaced by the ASCII codes that they represent. Formatting specifications are applied to the results of any expressions that follow the string.

Examples:

`disp "%15.10f",FPOS(0)`	Display value of **FPOS(0)** in 15 positions with 10 digits after the decimal point
`disp "%1i",IN0.2`	Display current state of **IN0.2** as one digit 0 or 1.
`disp "FVEL(0)=%15.10f",FVEL(0)`	Display value **FVEL(0)** with 10 decimal points, e.g., displayed output is: FVEL(0)= 997.2936183303
`disp "IN0 as hex: %04X",IN0`	Display value of **IN0** in hex, e.g., displayed output is: IN0 as hex: 0A1D
`disp "IN0.0-3 as binary: %1u%1u%1u%1u", IN0.0,IN0.1,IN0.2,IN0.3`	Display values of bits 0-3 of **IN0** as binary, e.g., displayed output is: IN0.0-3 as binary: 0110

The output string is sent to a communication channel. The channel is specified by the current value of standard **DISPCH** (default channel) variable. The following values are available:

- 1 – Serial communication channel COM1.
- 2 – Serial communication channel COM2.
- 6 – Ethernet network (TCP).
- 7 – Ethernet network (TCP).

- 8 – Ethernet network (TCP).
- 9 – Ethernet network (TCP).
- 10 – Ethernet Point-to-Point network. (UDP)
- -1 – No default channel is specified, the **disp** command uses the last channel activated by the host.
- -2 – All channels.

8.9.5.2 send Command

Description

The **send** command is the same as the **disp** command but also specifies the communication channel for the output string. The communication channel is the first argument.

Syntax

send *channel-number, disp-arguments*

Where:

channel-number is an integer identifying the communication channel to which the message will be sent (see **Table 23**).

Table 23 Channel Number Argument

Channel #	Description
1	Serial communication channel COM1
2	Serial communication channel COM2
6	Ethernet network (TCP)
7	Ethernet network (TCP)
8	Ethernet network (TCP)
9	Ethernet network (TCP)
10	Ethernet Point-to-Point network. (UDP)
-1	No default channel is specified, uses the last channel activated by the host
-2	All channels

disp-arguments are the same as those detailed for the **disp** command (see **Section 8.9.5.1 - disp Command**).

8.9.5.3 Differences between Query Commands and the disp/send Commands

- Query commands are executed immediately and cannot be stored in a program buffer. The **disp** and **send** commands can be executed either immediately or can be stored in a buffer and executed as a part of program.
- Query commands can address any variable: ACSPL+ standard, global or local. The **disp** and **send** commands can address any ACSPL+ or global variable. Among the local

variables, only the local variables defined in the same buffer where the command is located are accessible to the disp command.

- Query commands can address whole arrays or sub-arrays. The **disp** and **send** commands must specify a calculable expression, i.e., only single elements of array may be involved.
- Query commands cannot contain expressions. The **disp** and **send** commands can contain expressions.
- The controller sends the reply to a query command to the same channel from which the command was received. Results of the **disp** command are sent to the communication channel defined by the **DISPCH** variable (see *SPiiPlus Command & Variable Reference Guide*). Results of the **send** command are sent to the communication channel defined by the communication channel argument (channel-number).

8.9.5.4 str Function

Description

The **str** function converts an integer array to a string.

Syntax

string str*(variable [,start-index] [,number])*

Arguments

variable	Name of user-defined integer array.
start_index	Index of the array.
number	Specifies number of characters.

Return value

The function returns a string composed of the array elements interpreted as characters.

Comments

Each element of the **variable** array is interpreted as an ASCII character. If an element value is in the range from 0 to 255, it is directly converted to the corresponding ASCII character. Otherwise, the value's modulo 256 is converted.

If neither **start_index** nor **number** is specified, the conversion takes all elements of the array. If only **start_index** is specified, the conversion takes all characters from the specified index to the end of array. Specifying **number** limits the number of characters in the resulting string.

The function can be used within the **send** or the **disp** command.

Example

The following example provides a mirror for channel 2, so that any received character is sent back:

```
int Char(1)
inp(2,Char,,1)
send 2,str(Char)
```

8.9.5.5 dstr Function

Description

The **dstr** function converts a string to an integer array.

Syntax

int dstr(string, variable [,start-index] [,number])

Arguments

string	String of characters enclosed in double quotation marks
variable	Name of user-defined integer array.
start_index	Index of the array.
number	Specifies number of characters.

Return value

The function returns the number of actually assigned characters.

Comments

The function decomposes **string** into individual characters and assigns the characters to the sequential elements of the **variable** array.

Each ASCII character is represented as its numerical value and stored in a separate element of the array.

If **start_index** is specified, the first character is assigned to the array element with the specified index. If **start_index** is omitted, the assignment starts from the first element of the array.

If **number** is omitted, the function assigns all characters of the string. If **number** is specified, the function assigns the specified number of characters. In both cases the assignment stops when the last array element is reached.

8.10 trigger Command

Description

The **trigger** command is used for specifying conditions for general purpose triggering.

Syntax

trigger *channel [,expression] [,timeout]*

Arguments

channel	Specifies an integer number from 0 to 7. The number defines the triggering channel, selects the **AST** element where the triggering bit will be set and defines the bit in the interrupt tag.
expression	Specifies the triggering condition. After the command is executed, the controller calculates the expression each controller cycle. Triggering occurs when the expression changes its value from zero to non-zero. If the argument is omitted, the command disables triggering in the specified channel.
timeout	Specifies triggering timeout in milliseconds. A positive number specifies how much time the controller waits for triggering. Once the timeout has elapsed, and triggering has not occurred, the controller raises the trigger bit unconditionally. After any triggering, the controller starts timeout counting from zero. If the argument is omitted, the triggering works without timeout.

Comments

One **trigger** command can cause triggering many times. The controller continues calculating the expression until another **trigger** command is executed in the same channel. Each time when the expression changes its value from zero to non-zero, the controller raises the trigger bit and produces the interrupt.

The following table specifies triggering bit and interrupt tag for each triggering channel:

Table 24 Trigger Bit and Interrupt for each Channel

Channel	Triggering Bit	Interrupt tag (Software interrupt 10), hexadecimal
0	AST0.11	0x00000001
1	AST1.11	0x00000002
2	AST2.11	0x00000004
3	AST3.11	0x00000008
4	AST4.11	0x00000010
5	AST5.11	0x00000020
6	AST6.11	0x00000040
7	AST7.11	0x00000080

8.11 Dynamic TCP/IP Addressing

8.11.1 TCP/IP Variable

The firmware supports both static and dynamic assignments of TCP/IP addresses.

The TCP/IP address is defined by the **TCPIP** variable. If TCPIP has a non-zero value, the controller uses the value as its TCP/IP address. In this case, other configuration parameters receive the following default values:

- Subnet mask - 255.255.255.0
- Gateway address - no gateway, i.e. no routing is supported

If **TCPIP** is zero, the controller uses the DHCP protocol to receive the network configuration from the DHCP server. The network configuration received from the DHCP server includes the following parameters:

- Controller's TCP/IP address
- Subnet mask
- Gateway address

To retrieve the assigned address in an ACSPL+ program, use the **getconf** function (see *SPiiPlus Command & Variable Reference Guide*) with key 310.

To find all SPiiPlus controllers in the network segment, use the C Library function: **acsc_GetEthernetCards**.

8.11.2 Using getconf/setconf to Access TCP/IP Address

The **getconf** and **setconf** functions with key 310 provide access to the controller TCP/IP address.

getconf(310, 0) returns an integer value that contains the TCP/IP address currently assigned to the controller. The index argument (second argument) of the function should be zero. If a TCP/IP protocol is not configured, or not supported, the return value is zero.

setconf(310, 0, address) configures the TCP/IP address for the controller. The index argument (second argument) of the function should be zero. If the address argument is zero, **setconf** activates a new execution of the DHCP protocol and obtains a new TCP/IP address from the host (the host may configure the same address as before). **setconf** does not change the **TCPIP** parameter. After power-up, the controller is initialized with the TCP/IP address set in the TCP/IP parameter.

The address value is a 32-bit integer that contains four bytes. Each byte represents one part of the TCP/IP dot address. The bytes follow in computer order. For example:

```
setconf(310, 0, 0x6400000a)  Assigns address 10.0.0.100
?x/getconf(310,0)            Executed from the terminal returns the hexidecimal
                             value: 6400000a
```

There are several limitations when using **setconf**(310):

- If the **TCPIP** variable stored in the flash is zero, setconf(310) must be used only with zero address argument. In other words, if the controller is configured for dynamic addressing, assigning static address is not allowed.
- If the **TCPIP** variable stored in the flash is not zero, **setconf**(310) must be used only with non-zero address arguments. In other words, if the controller is configured for static addressing, switching to a dynamic address is not allowed.
- **setconf**(310) has a long execution time. During this time, communication with the controller is impossible using any communication channel. Use **setconf**(310) only within the controller initialization sequence. Avoid attempts to communicate with the controller and the motor **enable** command or motion commands while **setconf**(310) is in progress.

8.11.3 Addressing Scenarios

There are four scenarios of assigning TCP/IP addresses to the controller:

1. Static Addressing
   - Configure the appropriate TCP/IP address in the **TCPIP** variable and store it in the flash.
   - After start-up, the controller adopts the address and answers to the corresponding telegrams.

   For using static addressing in a local network, the system administrator needs to reserve this address to avoid identical addresses in the network segment.
2. Quasi-dynamic Addressing
   - Configure any non-zero TCP/IP address in the **TCPIP** variable and store it in the flash.
   - After start-up, use **setconf** with a non-zero address argument to assign an actual address to the controller.

   This approach makes sense, if the controller is assigned with an address dependent on some condition available only at run time. For example, the controller may select its TCP/IP address on the basis of digital input states.
3. Dynamic Addressing
   - Configure a zero TCP/IP address in the **TCPIP** variable and store it in the flash.
   - After start-up, the controller initiates DHCP communication with the DHCP server and obtains the TCP/IP address and other network information.

   To use this method, the DHCP server should be accessible at the time of controller start-up.

   If the DHCP server is not accessible, the controller repeats the request several times. If all attempts fail, the controller disables Ethernet channel support and continues initialization.
4. Delayed Dynamic Addressing

   Use this method, if the DHCP server is not available at the time of controller start-up. It may occur, for example, if the controller and the computer running DHCP server are activated at the same time, and nobody knows which is ready first.
   - Configure zero TCP/IP address in the **TCPIP** variable and store it in the flash.
   - Create an initialization routine in one of the buffers, and store it in the flash:

```
AUTOEXEC:
WHILE getconf(310,0) = 0
      setconf(310,0,0)
END
! continue initialization
```

   As a result, the controller waits at the beginning of initialization until the DHCP server succeeds to in supplying a TCP/IP address.

8.12 Non-Default Connections

This section covers handling non-default connections.

Note	
	*It should be noted that many applications require switching between non-default or default connections within the process of operations. You can return to a default connection by setting the* **MFLAGS.#DEFCON** *(bit 17, Default Connection) to 1. This automatically resets the connect formula, updates dependence and equates corresponding* **APOS** *to* **RPOS** *(see* ***Chapter 7 - Connection to the Plant****).*

8.12.1 ROFFS Variable

The **ROFFS** variable is an 8 element arrary, one element per axis, that reads the offset calculated by the controller in the connect formula. As long as the motor is in the default connection (**MFLAGS.#DEFCON** = 1), offset **ROFFS** is zero. However, once you have specified a **connect** (see **Section 8.12.3 - connect Command**) formula:

```
connect RPOS(axis) = F(…)
```

the controller calculates offset **ROFFS(axis)** to prevent any immediate change of **RPOS(axis)** that may cause a jump of the motor. Then the controller calculates formula:

RPOS(axis) = F(…) + ROFFS(axis)

for each controller cycle.

The controller recalculates the offset to prevent motor jump when any of the following commands is executed:

- **connect**
- **set**
- **enable**
- **disable**
- **kill**

ROFFS reads the current value of the offset. Watching the offset value facilitates development and debugging of application with complex kinematics.

8.12.2 DAPOS Variable

The **DAPOS** variable is an 8 element arrary, one element per axis, that reads the Axis Position synchronous with the Reference Position. The variable supplements the existing **APOS** variable. The problem of the **APOS** variable is that the axis position is not synchronous with the **RPOS** and **FPOS**. For this reason watching **APOS** against **RPOS** or **FPOS** in the SPiiPlus MMI Application Studio **Scope** is inconvenient.

DAPOS reads the same values of axis position, but synchronously with **RPOS** and **FPOS**. Using synchronous axis position facilitates analysis and debugging of the **CONNECT** formula.

Use only **DAPOS** for watching the axis position in the **Scope**.

Use only **APOS** in the **connect** formula and in ACSPL+ program.

8.12.3 connect Command

The **connect** command defines the relation between motors and axes.

Note	*The* **connect** *command cannot be executed as long as the default connection bit (***MFLAGS.#DEFCON***) is raised.*

Syntax:

connect RPOS(axis) = expression

Where **RPOS(axis)** is an axis **RPOS** variable (for example, **RPOS(0)**), which receives the value of the **expression**.

For more information about **RPOS** and other common motion variables refer to the *SPiiPlus Command & Variable Reference Guide*.

The **connect** command is not an **assignment** command (see **Section 3.8.1 - Assignment Command**). It does not simply assign the result of the formula on the right side to the axis **RPOS**. The formula is not evaluated when the **connect** command is executed (which would be the case for an **assignment** command); instead, the formula is stored and then evaluated by the controller every controller cycle to calculate the corresponding **RPOS**.

After power-up the controller always starts with the default connection. The default connection means the following for each axis:

- Bit **MFLAGS.#DEFCON** is raised.
- The default connect formula is defined as **connect RPOS = APOS**.
- **APOS** and **RPOS** are linked, i.e., explicit (through the **set** command see **Section 4.1.5 - set Command**) or implicit change of one of these variables causes the same change in the other one.

Once an application resets **MFLAGS.#DEFCON**, it can then execute a **connect** and (typically) a **depends** command. At this point, the motor is considered to be in non-default connection.

Consider the following examples:

The commands

```
MFLAGS(1).#DEFCON = 0
connect RPOS(1) = APOS(0)
depends 1, 0
```

connect the 1 axis motor position to the 0 axis reference. If the 0 axis motor is also connected to the 0 axis reference, this provides gantry-like motion of two motors.

The command

```
ptp  0, 1000
```

will provide synchronous motion of both 0 and 1 axes motors.

The command:

```
connect RPOS(0) = APOS(0) + AIN(1)
```

connects the 0 axis motor position to the 0 axis reference plus analog input 1. In this case the 0 axis provides a motion and the analog input (for example, an external sensor) supplies a correction for the 0 axis motor.

The following commands

```
MFLAGS(2).#DEFCON = 0
connect RPOS(2) = APOS(2) + APOS(3)
depends 2, (2,3)
```

connect the 2 axis motor to the sum of 2 and 3 axes. The axes can each execute an independent motion, with the 2 axis motor following the sum of the two motions. Or the axes can participate in a single multi-axis motion.

The following illustrate uses of the **connect** command.

Note	
	*The* **connect** *command is normally followed by a* **depends** *command (see* ***Section 8.12.4 - depends Command****).*

❑ **Using a Non-Default Connection**

Listed below are some of the tasks that can be resolved using the appropriate connect formula:

- Introduce a gear ratio between a logical axis and a physical motor.
- Compensate for encoder errors and backlash.

- Compensate for non-orthogonality of machine slides.
- Compensate for undesired mutual interference between machine coordinates.
- Implement gantry axes
- Define the physical motion as a sum of a logical motion and a compensating signal.
- Define the physical motion as a sum of two or more logical motions.
- Inverse kinematics, such as programming in Cartesian coordinates a machine that actually has polar kinematics .

Typically, the **connect** command for a specific motor is executed only once in an ACSPL+ application.

A typical location for a **connect** command is after the homing process in the code that follows an **AUTOEXEC** label (see **Section 3.1.4 - Names: Variable and Label**). The following pseudocode executes homing of X and Y axes and configures them as a gantry pair that follows the motion on the X axis:

`AUTOEXEC:`	The controller automatically starts the program from the **AUTOEXEC** label after power-up.
`. . .`	Execute homing of 0 and 1 axes.
`MFLAGS1.#DEFCON=0`	Reset the **#DEFCON** bit.
`connect RPOS(1)=APOS(0)`	Set gantry-like connection (1 axis motor follows 0 axis).
`depends 1,0`	Specify dependence (1 axis motor depends on 0 axis).
`set APOS(0)=0,RPOS(0)=0,RPOS(1)=0`	Set origin.
`. . .`	Continue.

A more sophisticated application may require changing the connection in the middle of operations. The controller applies no limitations regarding when a connection can be changed. In a typical case, changing connection requires three commands:

`connect RPOS(0)=…`	Specify connection of 0 axis motor.
`depends 0,…`	Specify dependence of 0 axis motor.
`set APOS(0)=…,RPOS(0)=…`	Set origin of 0 axis and 0 axis motor.

To return to default connection use the following commands:

`connect RPOS(0)=APOS(0)`	The 0 axis motor will follow the 0 axis.
`MFLAGS(0).#DEFCON=1`	Set the **#DEFCON** bit.
`set RPOS(0)=…`	Set origin of 0 axis motor (if **#DEFCON=1**, **APOS** is set to the same value).

❑ Offset in Connect Formula

If a motor is in a non-default connection, the **APOS** and **RPOS** variables are not linked and may contain different values.

The **connect** command specifies a basic formula that the controller uses to calculate the **RPOS**. However, in the process of **RPOS** calculation the controller also adds an implicit offset which is not specified in the connect formula.

The controller calculates the offset automatically and recalculates it in the following circumstances:

- Execution of **connect RPOS=…** command for the motor.
- Execution of **set RPOS=…** or **set FPOS=…** command for the motor.
- Execution of **set APOS=…** command for any axis that the motor depends on.
- Execution of **enable** command for the motor.

When a **connect** command is executed, the offset is adjusted so that the **RPOS** specified on the left side of the connect formula and any **APOS** specified on the right side retain their current values. For this reason the **connect** command can be executed while the motor is enabled and does not cause a motor jump. Using this implicit offset the controller ignores any explicit offset specified in the **connect** formula. For example, the following commands have exactly the same effect:

```
connect RPOS(0) = 0.5*APOS(0) + 1000
```

and

```
connect RPOS(0) = 0.5*APOS(0) + 2000
```

because the explicit offset is ignored.

When an **enable** command is executed, the offset is adjusted so that the connection formula calculation yields the desired value (**RPOS** retains its value).

The **set RPOS=…** or **set FPOS=…** command immediately changes the values of **RPOS** and **FPOS**. The offset is recalculated so that the connection formula calculation yields the new desired value.

The **set APOS=…** command immediately changes the value of axis position **APOS**. In order to retain the current values of all **RPOS** components, the controller recalculates the offsets of all motors that depend on the axis.

8.12.4 depends Command

Description

The **depends** command complements the connect command, specifying dependence between a motor and axes.

Note	
	*The* **depends** *command cannot be executed as long as the default connection bit (***MFLAGS.#DEFCON***) is raised.*

Syntax

depends *dependent_axis, axes_specification*

Comments

The **dependent_axis** argument specifies an axis and the **axes_specification** argument specifies one or more axes on which the motor depends.

Typically, a **depends** command will follow a **connect** command. Whereas a **connect** command can define a mathematical correspondence between a motor's reference position (**RPOS**) and one or more axis positions (**APOS**), a **depends** command specifies the logical dependence between a motor and axes.

The **depends** command is necessary because generally the controller is not capable of deriving the dependence information from the **connect** formula alone. For this reason, once a **connect** command is executed, the controller resets the dependence information of the motor; the motor depends only on the corresponding axis.

Dependence information, as specified using a **depends** command, is required in the following cases. If the dependence information is not provided correctly, the controller may display strange behavior.

- A motor/axis query (for example, **?X**) returns the non-default dependence for that motor.
- When initiating a motion, the controller verifies if each motor dependent on the axes involved is enabled. If one or more motors are disabled, the motion does not start.
- If in the process of motion a motor is disabled or killed due to a fault or due to a **disable** of **kill** command, the controller immediately terminates all motions involving the axes that the motor depends on.
- Once a **set APOS=…** command is executed, the controller adjusts offsets in the connection formula of the motors that depend on the specified axis.

8.12.5 The match Function

Description

The **match** function calculates the axis position that matches current Reference Position of the same axis with zero offset.

Syntax

match *(axis, from, to)*

Arguments

axis	Specifies the axis to be matched.
from/to	Restricts the range within which the matching value is searched.

Comments

The function is useful in the case of non-default connections if a motor depends only on the same axis, a typical example is error compensation. For example, the following connection:

```
connect RPOS(1) = APOS(1) + mapby1(APOS(1), ErrorTable)
depends 1,1
```

defines 1-to-1 dependence. In this case, the command:

```
set APOS(1) = match(1, -1000, 1000)
```

can be used to find matching **APOS(1)** in the range -1000 to +1000 and to set offset **ROFFS1** to zero. In mathematical terms, the function finds the root of equation:

RPOS = F(x)

where **RPOS** is the current value of the **RPOS** variable and **F(x)** is the connect formula specified by you with **APOS** substituted for x.

The function succeeds if the unique root exists in the specified range. If there are several roots in the range, the function returns one of them. If the root does not exist, the function result is incorrect. It is your responsibility to restrict the range so that the function provides proper result.

8.13 Input Shaping

Input Shaping is a feed-forward control technique for reducing vibrations in computer controlled machines. The SPiiPlus controller implements an Input Shaping algorithm patented by Convolve, Inc.

Input Shaping support is an optional feature in the SPiiPlus controller. To use the feature, specify the option in the controller order code, as described in the product data sheet.

8.13.1 The inshape Function

Description

The **inshape** function implements the input shaping algorithm and its result is a dynamic output signal equal to the convolution of the input signal and the convolution pulses.

Syntax

real inshape*(real Input_val, int NP, real T_array, real Amp_array, real Buf)*

Arguments

Input_val	Input signal.
NP	Number of convolution pulses.
T_array	One-dimensional array specifying the times of each convolution in milliseconds.
Amp_array	One-dimensional array specifying the amplitudes of each convolution pulse.
Buf	One-dimensional array designating a buffer for internal use.

Comments

For correct calculation, the function must be called each controller cycle.

Vectors **T_array** and **Amp_array** define characteristics of the convolution pulses. If the vector size is greater than **NP**, only the first **NP** elements are used for the function calculation.

Vector **T_array** contains real numbers, so fractional numbers can be specified. However, the position of each pulse is rounded to a multiple of the controller cycle. If the controller cycle is one millisecond, the numbers in **T_array** are rounded to integers.

The elements of **T_array** must be arranged in ascending order.

Buf is used for the internal calculation in the function. You do not need to fill the buffer before the function call. The size of the vector must be not less than the maximum time of convolution pulses divided by the controller cycle. As **T_array** elements are sorted in ascending order, the condition can be expressed as:

Size(Buf) = T_array[NP-1]/CTIME+1

where **CTIME** is the controller cycle in milliseconds.

inshape detects the following error conditions:

- The size of vector **T_array** is less than **NP**.
- The size of vector **Amp_array** is less than **NP**.
- The size of **Buf** is not enough.

Note	
	*You have to take a few precautions working with the axis that implements input shaping:* 1. *Do not use* **kill** *for the axis. Use* **halt** *to terminate a motion.* 2. *Do not use a* **kill** *response to the faults. Re-define any kill response to a kill-disable response for the axis.* 3. *Execute any required* **set** *commands before input shaping is activated.* 4. *Use* **inshape** *in the* **connect** *command for the required axis. Execute* **connect** *every time an axis is enabled.*

Example

The following example illustrates implementation of input shaping for the 1 axis:

```
global real CnvT(4), CnvA(4), CnvB(500)!Definitions
safetyconf 1,#RL,"KD"; safetyconf Y,#LL,"KD"
safetyconf 1,#AL,"KD"
safetyconf 1,#PROG,"KD"          !Redefine Limit Switches, Acceleration and
                                 !Program fault responses to kill-disable.
                                 !In the same manner, redefine all kill-
                                 !type responses to kill-disable.
write CnvT
write CnvA
read CnvT
read CnvA                        !Read convolution data from the flash
                                 !memory.
MFLAGS(1).DEFCON=0
set APOS(1)=0, RPOS(1)=0
…
stop
```

```
MFLAGS(1).#DEFCON=0                 !Reset default connection flag
set APOS(1)=0, RPOS(1)=0            !Set origin - if required
...                                 !Program actions
on MST(1).#ENABLED                  !Execute auto routine once 1 axis is
                                    !enabled
connect RPOS(1)=inshape(APOS(1),4,CnvT,CnvA,CnvB)
                                    !Define connection
depends 1,1                         !Define Dependance
ret
```

8.13.2 Using the Convolve Web Site

This section describes the procedures used for designing Input Shapers™ using the Convolve, Incorporated web site for designing Input Shapers™ located at: **http://inputshaping.convolve.com**.

Access to the web site requires a user name and password.

Note	
	*Use the name and password that you received from ACS Motion Control along with the controller.*

The first time you log into the web site, you will be required to review the licensing agreement for use of the site. For each subsequent log-in you will be reminded of the licensing agreement and remaining number of trials until expiration of the account will be displayed.

8.13.3 Data Entry Dialog

The **Data Entry** dialog, see **Figure 24**, is the primary means for entering the parameters to design an Input Shaper™. The **Data Entry** dialog displays the current **User ID**, **Name**, and **Date**. Use the form fields as follows:

- **Constraint** - Set to **Digital Positive Mixed Constr.**. This gives you the maximum freedom for selecting other design parameters.
- **Output Format** - Set to **Standard Format**.
- **Axis** - Set the required axis
- **Frequency or DeltaT** - Set **Delta**T value according to the controller parameter **CTIME** (Cycle Time). **CTIME** defines the controller cycle in milliseconds, and **DeltaT** should be defined in seconds; therefore, if **CTIME**=1, set **DeltaT**=0.001. **Frequency** is automatically computed.
- **Amplitude Sum** - Set = 100000
- **Measured Frequency and Damping** entries - Enter from one up to six vibration frequencies and the associated damping for each frequency. The vibration frequencies are modeled as second order systems, which are normally characterized by a natural frequency and damping ratio (or zeta).

The damped natural frequency will be less than the natural frequency by the factor,

$$\sqrt{1-\text{zeta}^2}$$

The frequencies are referred to as "measured frequencies" because the program performs the calculations to convert them to "undamped natural frequencies".

- **Hardware ID**, **Location**, **Tooling ID** and **Notes** - These are optional fields that can be used to document a configuration for a specific machine. Text entered into these fields is copied into the output file as a comment.
- **Robust** check box - This box should normally be selected to design a "robust" Input Shaper™ for a range of ±15% of the nominal frequency. This box can be left unchecked when the range of frequency variation is less than ±5%. Non-robust Input Shapers™ are 50% shorter in duration than a robust Input Shapers™. Please refer to the discussion on Insensitivity Curves below for more information.

When all of the data has been entered, press the **Calculate** to start the calculation. It can take several seconds to complete the calculation.

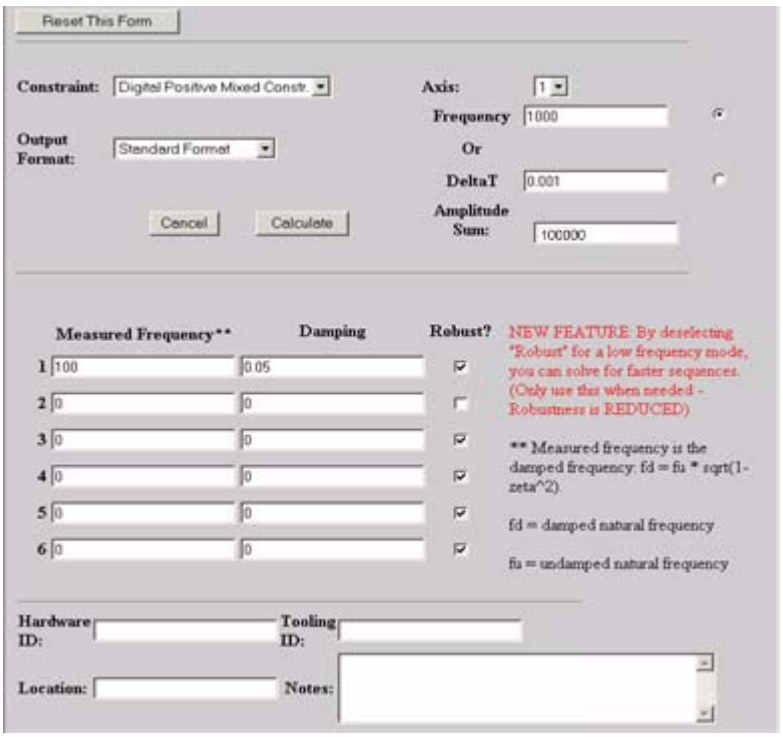


Figure 24 Data Entry Dialog

When the calculations are complete, the screen in **Figure 25** appears.

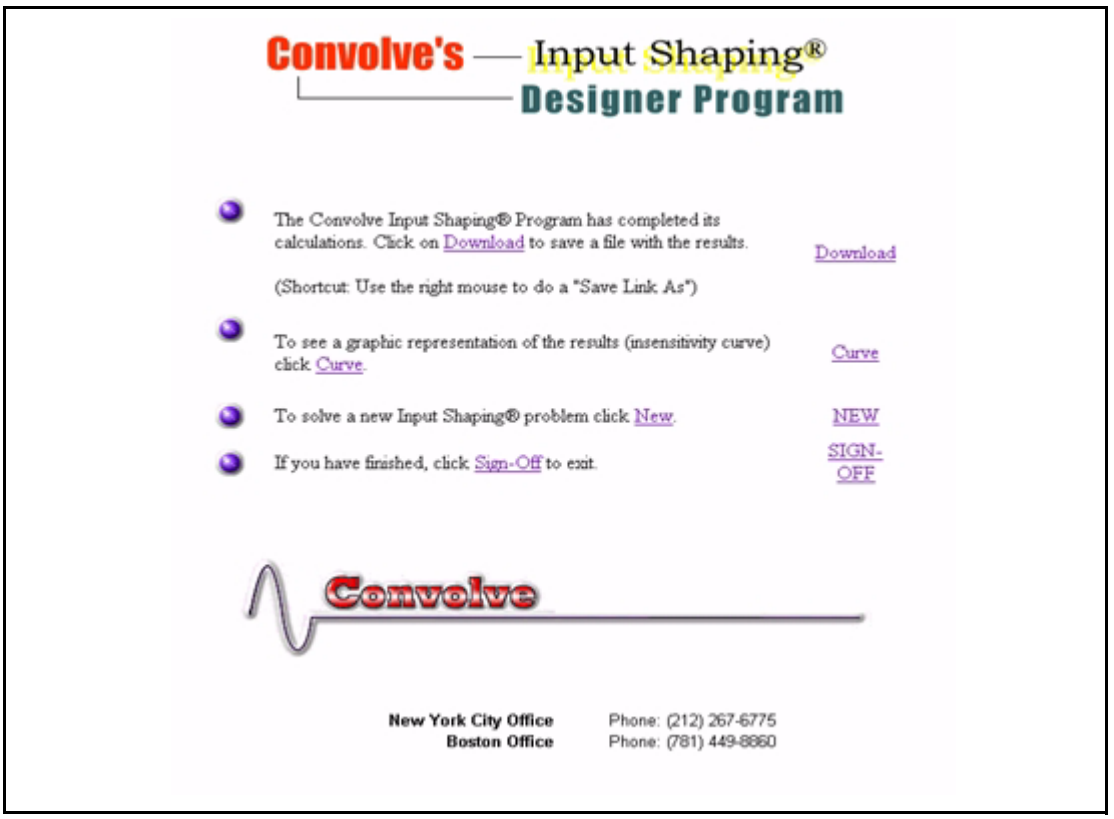


Figure 25 Screen at the Conclusion of Calculation

Click **Download** to access another window that contains the results of the calculation. A sample output file is shown below for the design of an Input Shaper with the following parameters:

- Measured Vibration Frequency - 100 Hz
- Damping - 0.05
- Delta T for trajectory generation - 0.001 second
- Amplitude sum - 100000
- Robust was selected - (Constraint = 1)

Figure 26 illustrates this output.

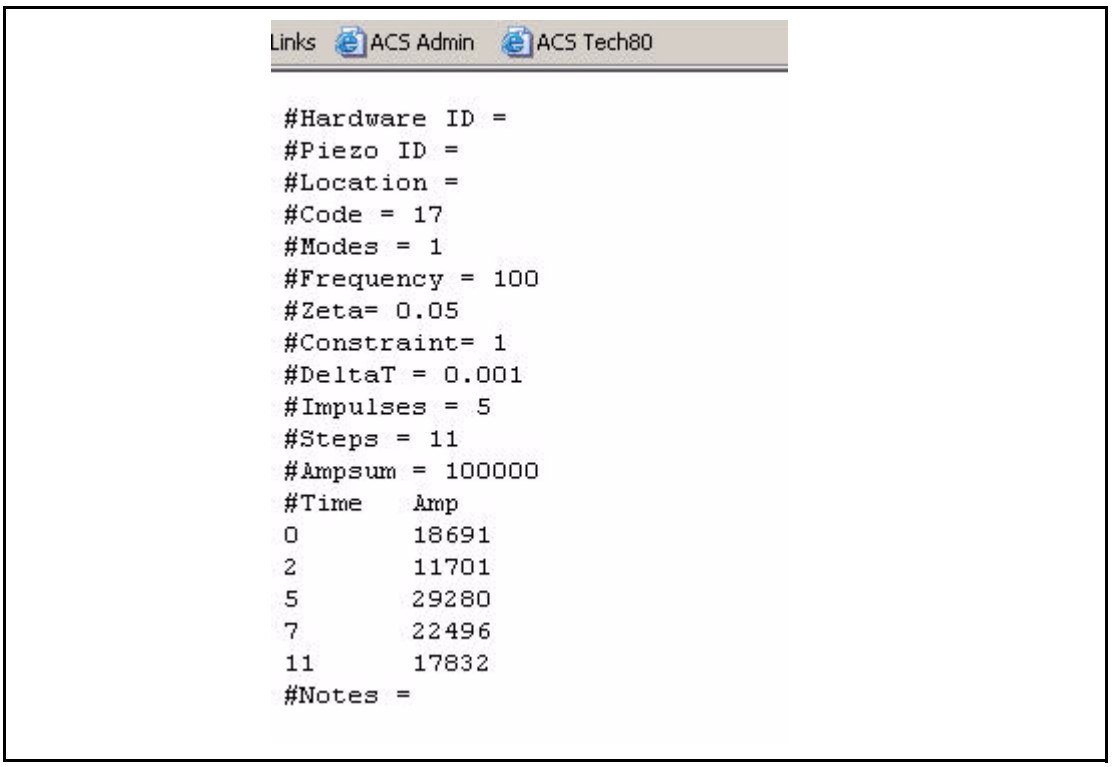


Figure 26 Window Accessed by Download

The output from the calculation is five impulses that characterize the Input Shaper™. The delays are specified in terms of digital time steps and for **Delta T** =.001 second, the delays would be at 0, 0.002, 0.005, 0.007 and 0.011 seconds.

The reported Time values should be used in the ***T*** argument of the **inshape** function. The reported **Amp** values are multiplied by 100000. The following example shows how the inshape arguments should be initialized to use the reported values:

```
global real CnvT(5), CnvA(5), CnvB(12)
                                !Definitions
CnvT(0)=0; CnvT(1)=2; CnvT(2)=5;CnvT(3)=7; CnvT(4)=11;
                                !Initialize CnvT array
CnvA(0)=0.18693; CnvA(1)=0.11701; CnvA(2)=0.29279;CnvA(3)=0.22495;
CnvA(4)=0.17832;                !Initialize CnvA array
...                             !Program actions
stop
on MST(1).#ENABLED              !Execute auto routine once the 1 axis is
                                !enabled
```

```
MFLAGS(1).#DEFCON=0                !Reset default connection flag
connect RPOS(1)=inshape(APOS(1),5,CnvT,CnvA,CnvB)
                                   !Define connection
depends 1,1                        !Define Dependance
ret
```

The results from the calculation can be saved directly by using the web browser file command **File →Save as ….** Save the results as a text file (*.txt), so Input Shaper™ coefficients can be downloaded into a controller. The same file will sent by email to the email account specified when the account was created.

Click the **Curve** option (shown in **Figure 25**) to display the useful frequency range for the Input Shaper™. A new window, **Figure 27**, opens to display the **Insensitivity Curve**. The following plot is the **Insensitivity Curve** for the example shown above:

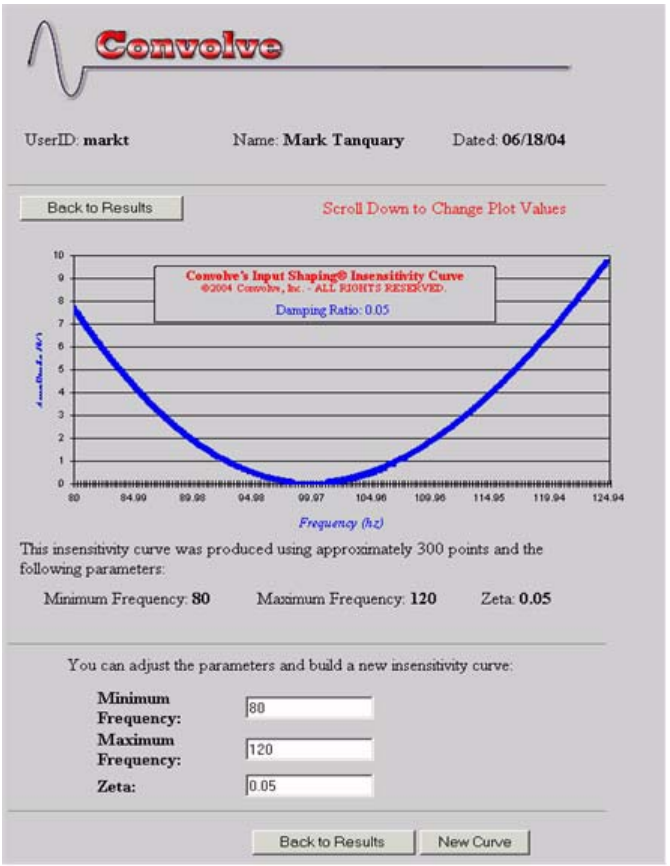


Figure 27 Insensitivity Curve Illustration

The **Insensitivity Curve** displays the theoretical amount of residual vibration that results after applying the Input Shaper to the system. The horizontal axis is the vibration frequency of the system and the vertical axis is the percentage of remaining or residual vibration. When the residual vibration = 0, for a perfect linear system, there should be no vibration after applying the Input Shaper™. In the case shown above, the residual vibration should be 0 at a frequency of 100 Hz, which was the design frequency.

If the frequency of the actual system is not equal to 100 Hz, there will be residual vibration present. This level of vibration is expressed on a percentage basis. If the residual vibration equals 100%, then the no vibration reduction has occurred. For the Input Shaper shown above, the system vibration frequency could vary between 85 to 115 Hz and the residual vibration should be less than 5% (or when compared to the case of not using the Input Shaper, the vibrations should be reduced by 95%).

Note	
	*It should be remembered that the vibration reduction is expressed for a "perfect" linear system. There are some examples of perfect, linear systems in the real world that do achieve close to 100% cancellation. (One good example of a "perfect" system is a simulation. Input Shaping usually performs very well in simulation.) However for most physical systems, the vibration reduction will usually be in the range of 95 to 98% of the original vibrations.*

The frequency range for the **Insensitivity Curve** can be changed by selecting new minimum and maximum frequency values and then clicking **New Curve**. The damping ratio can also be changed to determine the effect of changes in the estimated damping of the system. In general, changes in damping in the range from 0.005 to 0.1 have a limited effect on the **Insensitivity Curve**. When the system damping ratio is greater than 0.1, the effectiveness of the Input Shaper™ will be improved by specifying a damping ratio that is close to that of the actual system.

The **Insensitivity Curve** can also be used to examine the effect of selecting whether or not an Input Shaper should be designed using the **Robust** selection. If **Robust** is not selected for a particular frequency, the useful range of the Input Shaper will be reduced. For example the following **Insensitivity Curve**, **Figure 28**, is for the same Input Shaper™ without **Robust**:

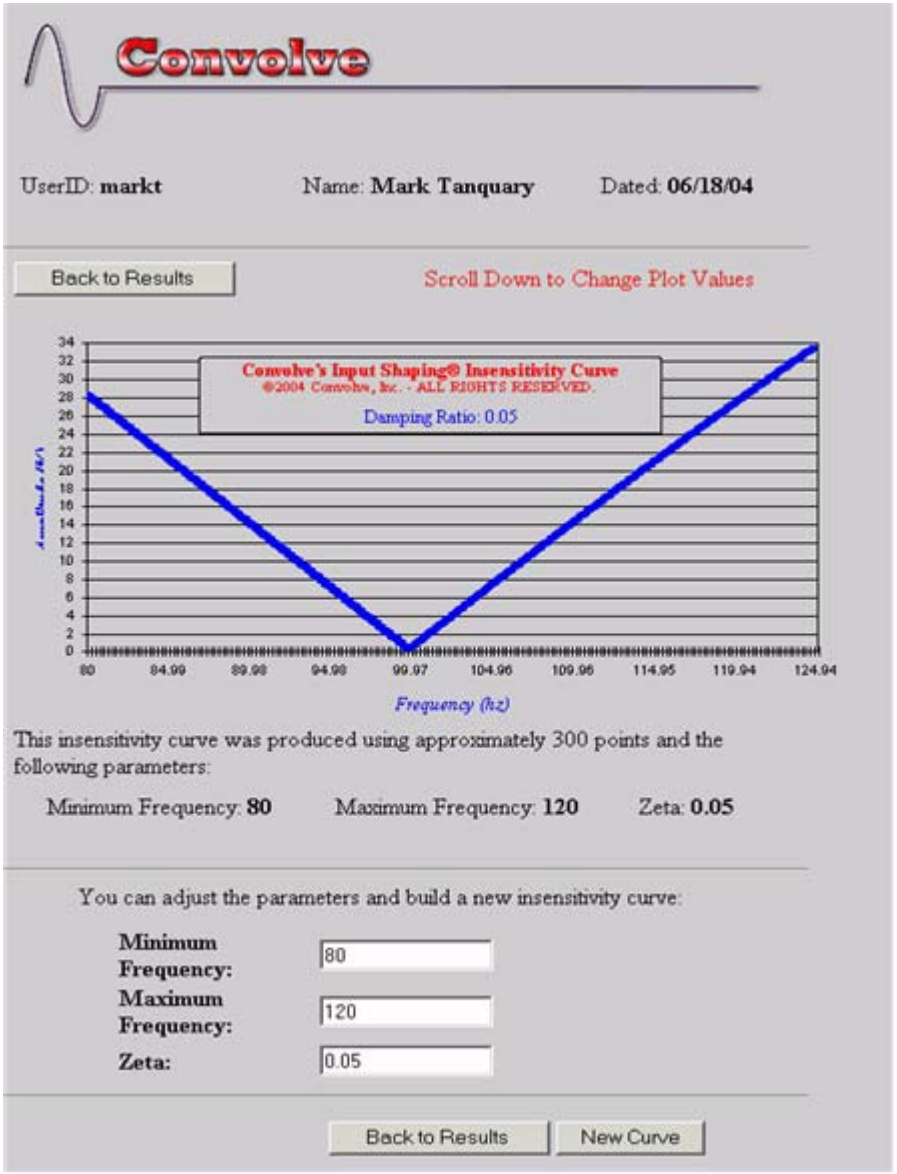


Figure 28 Insensitivity Curve without Robust

In this case, the frequency range for 95% cancellation of the vibration is much smaller, 95 to 105Hz. However the benefit is that the duration of the Input Shaper will be 50% less.

It is possible to use non-robust setting to design an Input Shaper that will be effective over a wider range of frequencies. Two frequencies can be specified to create a frequency band for vibration cancellation.

8.14 DRA Algorithm

The ACS proprietary Disturbance Rejection Algorithm (DRA) is used to improve the disturbance rejection response of the servo, and helps to minimize the position error during the settling phase and shorten the settling time.

The most common use of the algorithm is to improve the settling of systems mounted on passive isolation platforms. Passive isolation is typically used to isolate systems from disturbances transmitted from the floor. They employ a seismic mass supported on a soft spring made of rubber, metal, or air. The spring's damping action absorbs vibrations above the spring's resonance. For this reason, passive isolation manufacturers usually try to lower spring resonant frequency to increase the effective isolation range. When a servo force is applied to generate motion, it also acts on the isolated stationary base, causing it to vibrate. Because the frequency is low (usually below 1 Hz, to 10 Hz) and damping is very light, the isolation system continues vibrating long after the motion profile has ended. This vibration acts as disturbance to the servo system, introduces position error, and extends the settling time.

The Disturbance Rejection algorithm is used to minimize the latter effect and improve the position error during settling. This is demonstrated in **Figure 29** and **Figure 30**. The green graph shows the velocity command (in [mm/sec]) of a linear stage mounted on passive isolation, with a resonant frequency of approximately 5 Hz. The red graph shows the position error with a standard PIV algorithm. The 5Hz disturbance is clearly observed during settling. The disturbance is relatively small (less than 1 micron), yet it may be critical if the required settling window is very small (as an example, the resolution of semiconductor instruments is approaching and in some cases going below 1nm). This disturbance can be minimized by increasing the PIV gains (**SLVKP**, **SLVKI**, **SLPKP** - see *SPiiPlus ACSPL+ Command and Variable Reference Guide*), yet it cannot necessarily be eliminated and if the values of the PIV gains are too high this may lead to marginal stability. A better solution is using the DRA algorithm. As it can be seen in the blue graph the disturbance is fully eliminated.

If tuned properly the algorithm has minimal effect on the servo stability margins.

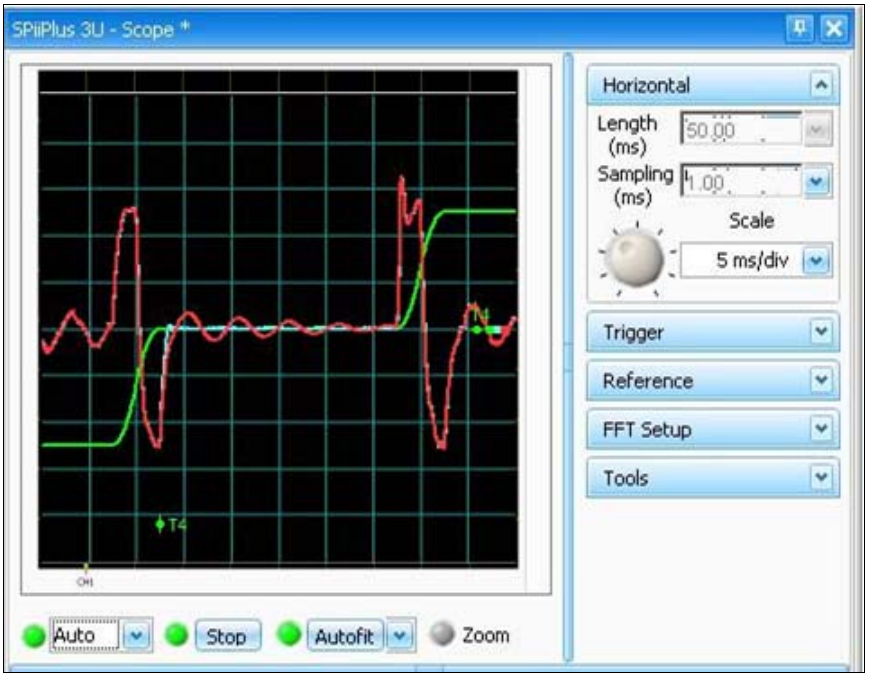


Figure 29 Example 1 of Using DRA

The meanings of the colors of the scope shot are:

- Green - Reference velocity (200 mm/sec per division)
- Red - Position error without DRA (1 division = 2 microns)
- Blue - Position error with DRA (1 division = 2 microns)

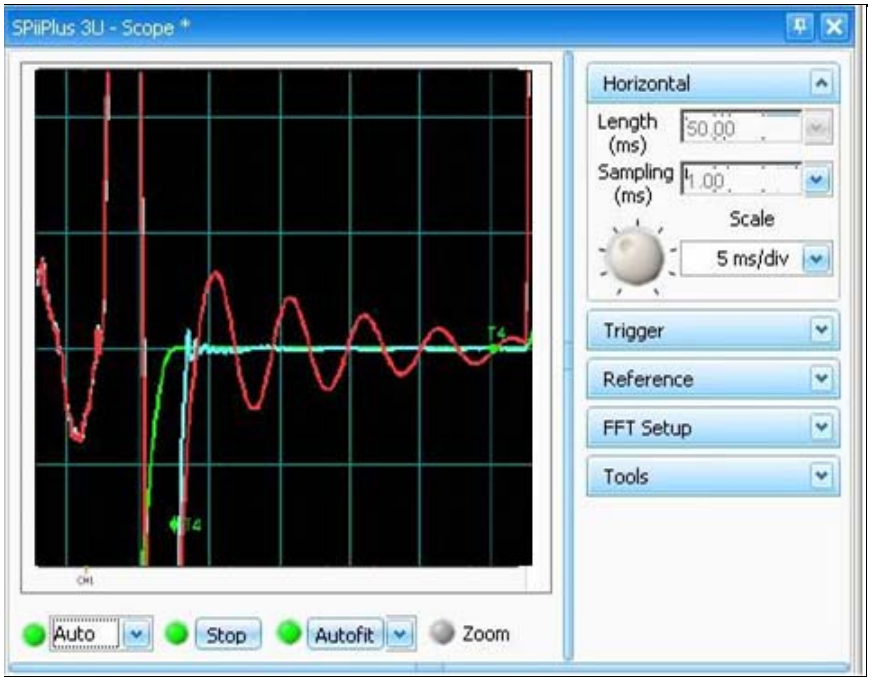


Figure 30 Example 2 of using DRA (zoomed)

The meanings of the colors of the scope shot are:

- Green - Reference velocity (1 division = 100 mm/sec)
- Red - Position error without DRA (1 division = 1 microns)
- Blue - Position error with DRA (1 division =1 microns)

DRA has two parameters that have to be tuned:

- **SLDRA** - This is a frequency specified in [Hz]. It should typically be set to 1-2 times the crossover frequency of the open loop FRF. In the example below (**Figure 31**) the open loop crossover frequency was about 100Hz, so SLDRA was set to the same value.
- **SLDRX** - This parameter stands for maximum DRA correction and specified in units/sec. As a rule-of-thumb, it should be set according to the maximal periodical velocity error during the settling process. This can be deduced by observing the feedback velocity (FVEL) variable in the SPiiPlus MMI Application Studio **Scope**, when SLDRX = 0. In **Figure 30**, it is about 0.01 mm/sec. Based on the time domain response the value should be further optimized to achieve the optimal response with minimal overshoot and minimal settling time.

In order to disable DRA both parameters are set to zero (default).

Note	
	*DRA is usually not very effective if the vibration frequency is relatively high (>10Hz), or the system bandwidth is very low.*

Excessive values of SLRA, and SLDRX may cause servo instability, ringing and increased standstill jitter. In such cases the parameters should be significantly reduced. With good settings, you should usually be able to double the parameters without getting instability.

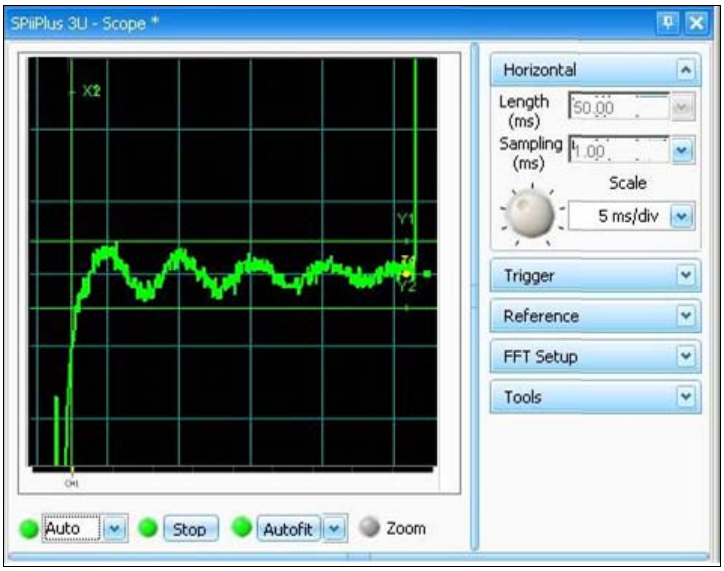


Figure 31 Example of Velocity Error

Velocity error (1 division = 0.02 mm/sec) during settling process of a linear axis. The maximal value of the periodical error is used to determine the SLDRX parameter.

8.15 BI-Quad Filter

A Bi-Quad filter is added to the velocity loop control in addition to the existing 2nd order Low-pass and Notch filters.

The Bi-Quad filter is the most general 2nd order filter. It has two poles and two zeros. It can be thought of as a high-pass filter in series with a low-pass filter. The transfer function of the Bi-Quad filter is as follows:

$$H(s) = \frac{(s/\omega_N)^2 + 2\zeta_N(s/\omega_N)+1}{(s/\omega_D)^2 + 2\zeta_D(s/\omega_D)+1}$$

Where:

- ω_N and ω_D are the numerator (high-pass filter zero) and denominator (low-pass filter pole) frequencies, respectively.
- ζ_N and ζ_D are the numerator and denominator damping ratios, respectively.

The corresponding ACSPL+ parameters are:

- **SLVB0NF**, **SLVB0DF** - numerator and denominator frequencies in [Hz]. Range: 0.1 - 4000 [Hz].
- **SLVB0ND**, **SLVB0DD** - numerator and denominator damping ratios. Range: 0.01 - 1.
- MFLAGS bit 16 is used to activate the filter (by default the filter is off).

The Bi-Quad filter can be used to compensate mechanical resonances, improve stability margins and system bandwidth.

The Bi-Quad filter can be configured as an additional Notch using the following formulas:

- Set the numerator and denominator frequencies equal to the Notch frequency in[Hz]:

  $$SLVB0NF = SLVB0DF = SLVNFRQ[Hz]$$

- Set the numerator damping ratio equal to half the ratio between the Notch width and Notch frequency:

  $$SLVB0ND = \frac{SLVNWID[Hz]}{2 \times SLVNFRQ[Hz]}$$

  Maximal recommended ratio between width and frequency= 1/3.

- Set the denominator damping ratio equal the numerator damping ratio times the Notch attenuation (in absolute value):

  $$SLVB0DD = SLVB0ND * SLVNATT$$

Below there are several examples that demonstrate the generality of the Bi-Quad filter.

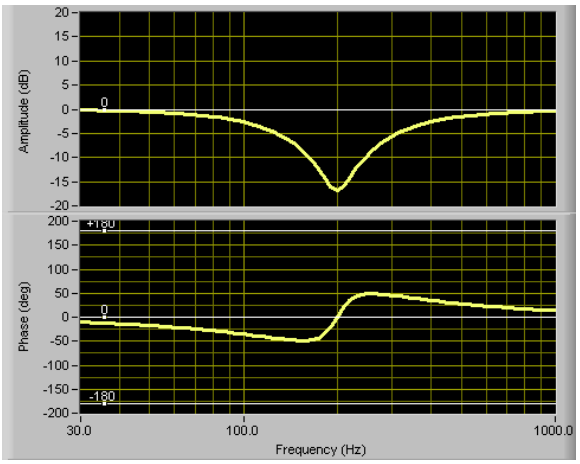


Figure 32 Bi-Quad Configured as a Notch Filter

SLVB0NF = **SLVB0DF**

SLVB0ND<**SLVB0DD**

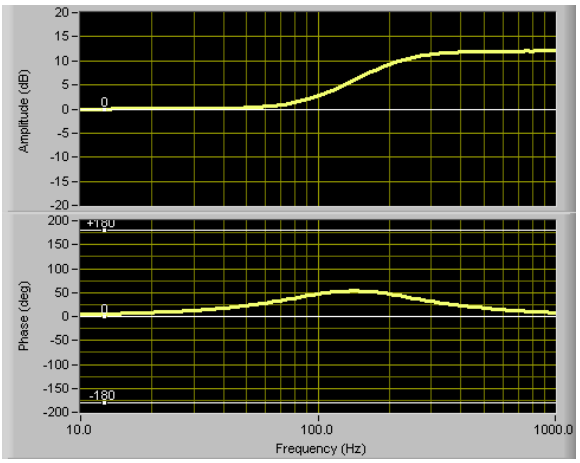


Figure 33 Bi-Quad Configured as a 2nd Order Lead Filter

This kind of filter is used to improve the phase margin

SLVB0NF < **SLVB0DF**

In the example the damping ratios are equal:

SLVB0ND=**SLVB0DD** =0.707

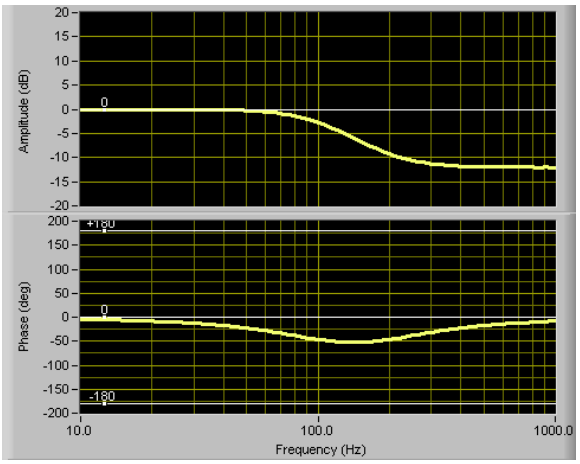


Figure 34 Bi-Quad Configured as a 2nd Order Lag Filter

This kind of filter is used to improve the gain margin

SLVB0NF > **SLVB0DF**

In the example the damping ratios are equal:

SLVB0ND=**SLVB0DD** =0.707

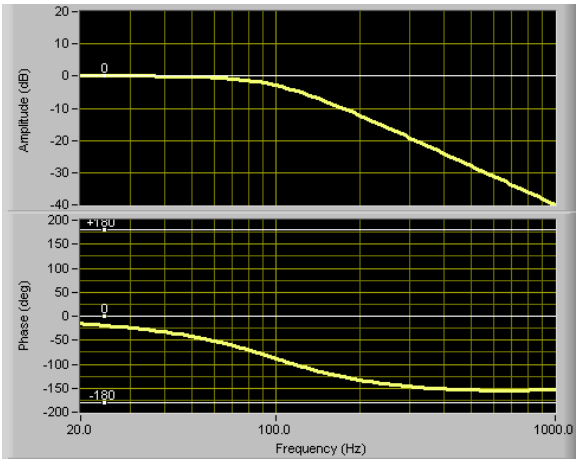


Figure 35 Bi-Quad Configured as a 2nd Order Low Pass Filter

SLVB0NF > **SLVB0DF**

8.16 Feedback Routing

The new routing implementation in NT allows any axis to use any encoder input available. Routing is effected through using the **SLPROUT**(position feedback), **SLVROUT** (velocity feedback) and **SLCROUT** (commutation position feedback) variables (for details of these variables see *SPiiPlus Command & Variable Reference Guide*).

Feedback is displayed using the **FPOS** (associated with SLPROUT), **FVEL** (associated with SLVROUT), and **FACC** (associated with SLCROUT) variables.

An encoder error in an encoder pointed to by **SLPROUT** will cause an encoder error in the axis. An encoder error in an encoder pointed to by either **SLVROUT** or **SLCROUT** will cause an encoder2 error to be reported.

Index and Mark inputs are also routed according to SLPROUT. Routing two axes to a single input will cause undefined index and mark behavior and should be avoided.

The firmware will clear all encoders associated (SLPROUT, SLCROUT, SLVROUT) with an axis following an encoder reset.

Note	
	*Encoder variables (***E_TYPE***,* **E_SCMUL***,* **SCCOFFS**, **SCSOFFS**, **SCGAIN**, *and* **SCPHASE***) are connected to the encoder feedback, not an axi; therefor they are unaffected by routing.*

9 Generic EtherCAT Master

This chapter describes the generic interface of EtherCAT master functionality for the SpiiPlus NT family. This interface allows configuring and controlling any EtherCAT-compliant slave device via ACSPL+ commands and variables. Some devices, like qualified motion drives and I/O, will have a special interfaces in addition; however the generic one is the base and can be used for them too.

9.1 Stack Behaviour

On the controller's start-up, the stack looks for the C:\ECAT.XML file. This file describes the network setup completely according to EtherCAT standard. The stack performs the following sequence of actions, if any of them is not successful, the sequence is stopped and the stack reports failure to go operational.

1. Scan the network and verify the scanned results via the XML file. In case of inconsistency stop
2. Initialize each slave
3. Initialize DC transmission
4. Set slaves to OP state
5. Initiate Master-Bus synchronization (if applicable) and wait for synchronization
6. Set Master state to OP

9.2 Interface Description

9.2.1 ACSPL+ Variables

9.2.1.1 ECST - EtherCAT State

The **ECST** is a one-byte variable the bits of which provide indications of the status of the EtherCAT. The status stored in the bits is detailed in **Table 25**.

Table 25 ECST Bits

Bit	Designator	Description
0	#SCAN	The scan process was performed successfully.
1	#CONFIG	There is no deviation between XML and actual setup.
2	#INITOK	All bus devices are successfully set to INIT state.
3	#CONNECTED	Indicates valid Ethernet cable connection to the master.
4	#INSYNC	If DCM is used, indicates synchronization between master and the bus.
5	#OP	The EtherCAT bus is operational.
6	#DCSYNC	Distributed clocks are synchronized.

Note	
	*All bits (except* **#INSYNC** *in some cases) should be true for proper bus functioning, for monitoring the bus state, checking the* ***#OP*** *bit is sufficient. Any bus error will reset the* **#OP** *bit.*

9.2.1.2 ECERR

Any EtherCAT error sets **ECST.#OP** to false and the error code is latched in the **ECERR** variable. The only way to reset the error state and to clear the **ECERR** value is by calling **ECRESTART** function.

The EtherCAT error codes are detailed in **Table 26**.

Table 26 EtherCAT Error Codes

Error Code	Description
6000	General EtherCAT error.
6001	EtherCAT cable not connected.
6002	EtherCAT master is in incorrect state.
6003	Not all EtherCAT frames can be processed.
6004	EtherCAT Slave error.
6005	EtherCAT initialization failure.
6006	EtherCAT cannot complete the operation.
6007	EtherCAT work count error.
6008	Not all EtherCAT slaves are operational.
6009	EtherCAT protocol timeout.
6010	Slave initialization failed.
6011	Bus configuration mismatch

9.2.2 #ETHERCAT

The #ETHERCAT command is available for gaining complete information about the connected EtherCAT slaves: The command is entered through the SPiiPlus MMI Application Studio **Communication Terminal**. The command provides the following:

- ❑ Slave number
- ❑ Vendor ID
- ❑ Product ID
- ❑ Revision
- ❑ Serial number
- ❑ EtherCAT physical address

- DC support
- Mailbox support

Following the display of the above general data, a list of network variables is displayed. Each variable is described with:

- Name (as in XML)
- Offset inside the telegram (magic number that is used for mapping)
- IN or OUT description
- Data size

9.3 EtherCAT Functions

9.3.1 Mapping Functions

9.3.1.1 ECIN

The **ECIN** function is used for mapping input variables to the EtherCAT network.

Syntax:

ecin(int offset, Varname)

Where:

offset	Internal EtherCAT offset of network variable (which can be seen by running the **#ETHERCAT** command).
Varname	Valid name of ACSPL+ variable, global or standard.

Comments:

Once the function is called successfully, the Firmware copies the value of the network input variable at the corresponding EtherCAT offset into the ACSPL+ variable, every controller cycle.

There is no restriction on number of mapped network variables.

The mapping is allowed only when stack is operational, that is, **ECST.#OP** is true.

In the event of incorrect parameters or stack state, the function will produce the corresponding runtime error.

9.3.1.2 ECOUT

The **ECOUT** function is used for mapping output variables to the EtherCAT network.

Syntax:

ecout(int offset, Varname)

Where:

offset	Internal EtherCAT offset of network variable (which can be seen by running the **#ETHERCAT** command).
Varname	Valid name of ACSPL+ variable, global or standard.

Comments:

The Firmware copies the value of ACSPL+ variable into the network output variable at the corresponding EtherCAT offset, every controller cycle.

There is no restriction on number of mapped network variables.

The mapping is allowed only when stack is operational, that is, **ECST.#OP** is true.

In the event of incorrect parameters or stack state, the function will produce the corresponding runtime error.

9.3.1.3 ECUNMAP

The **ECUNMAP** is used for resetting all previous mapping defined by **ECIN** and **ECOUT**.

Syntax:

ecunmap

Comments:

The function call is legal only when stack is operational, that is, **ECST.#OP** is true.

9.3.2 CoE Functions

CoE functions are required for SDO transfers in CoE. SDOs are part of the cyclic EtherCAT data transfer. It is impossible to define a generic function for any kind of mailbox transfer, such as protocols like EoE, FoE and VoE have their own definitions. So CoE is supported first.

Note	*The* **#ETHERCAT** *command can be used to check if a slave has Mailbox support.*

9.3.2.1 COEWRITE

The **COEWRITE** function is used to write a value into a given slave.

Syntax:

coewrite[/size] (int slave,int Index,int Subindex,real Value)

Where:

size	1, 2 or 4 - the number of bytes in the OD or /f for floating.
slave	Slave number (can be obtained from the **#ETHERCAT** command).
index	Index in the OD.
subindex	Subindex in the OD.
Value	Value to be written.

Comments:

In case of wrong parameters, the corresponding runtime error will be generated. The function cannot be used in the **Communication Terminal**. The function delays the buffer execution on its line until it is successful or fails the whole buffer with timeout or other error.

9.3.2.2 COEREAD

The **COEREAD** function is used to read a value from a given slave.

Syntax:

real coeread[/size] (int slave,int Index,int Subindex)

Where:

size	1, 2 or 4 - the number of bytes in the OD or /f for floating.
slave	Slave number (can be obtained from the **#ETHERCAT** command).
index	Index in the OD.
subindex	Subindex in the OD.

Comments:

The function returns the received value or fails with runtime error.

In case of wrong parameters, the corresponding runtime error will be generated. The function cannot be used in the **Communication Terminal**. The function delays the buffer execution on its line until it is successful or fails the whole buffer with timeout or other error.

10 Errors & Diagnostics

Error detection can occur in three circumstances:

- An erroneous command is received through any communication channel
- An error occurs while the controller executes an ACSPL+ program
- The controller kills a motion or disables a motor because a fault was detected

This chapter provides a description of how error codes are generated. For the complete list of error codes see *SPiiPlus Command & Variable Reference Guide*.

10.1 Error Codes

Each error in the controller has a corresponding 4-digit error code. Regardless of how an error is detected and reported, you can request the error description from the controller. A request for error description consists of two question marks followed by the error code. For example, the following communication may occur when the controller is in protected mode:

`VEL(0) = 10000`	Attempt of assignment to a protected variable.
`?2085`	The controller reports an error.
`??2085`	Request for error description.
`Protection violation`	Error description

10.1.1 Error Code Ranges

Table 27 provides a breakdown of the error code ranges.

Table 27 SPiiPlus Error Code Ranges (page 1 of 2)

Range	Type	Description
0 - 999	Errors detected by SPiiPlus C Library	Errors are detected by C Library without communication with the controller. The controller itself never returns error codes in this range. A host application that calls a C Library function can receive this code if a function call failed. For explanation of the errors see the *SPiiPlus C Library Reference Guide*.
1000 - 1999	Errors in terminal commands	Errors in terminal commands are reported immediately in the prompt that is displayed in response to the command.
2000 - 2999	ACSPL+ compilation errors	ACSPL+ compilation errors are reported either immediately when the erroneous line is inserted, or subsequently, when the controller attempts to compile an ACSPL+ program. If a program in a buffer undergoes compilation and an error is detected, the error code is stored in the corresponding element of the **PERR** array and the erroneous line number is stored in the corresponding element of the **PERL** array

Table 27 SPiiPlus Error Code Ranges (page 2 of 2)

Range	Type	Description
3000 - 3999	ACSPL+ termination codes	Codes from 3000 to 3019 do not indicate an error. For example, code 3002 reports that the user has terminated the program. Codes from 3020 to 3999 indicate run-time errors. If an error occurs in immediate execution of ACSPL+ command, the error is indicated immediately in the prompt. If an error occurs when an ACSPL+ program is executed in a buffer, no immediate indication is provided. Instead, the error code and the line number are stored in the corresponding elements of the **PERR** and **PERL** arrays.
5000 - 5999	Motion termination codes and motor disable codes	Codes from 5000 to 5008 do not indicate an error. They report normal motion termination. Codes from 5009 and higher appear when a motion is terminated or a motor is disabled due to a fault detected by the controller. When a motion terminates abnormally, or a motor is disabled , the error code is stored in the **MERR** variable.
6000 - 6999	EtherCAT code	Stored in the **ECERR** variable, and when occurs, sets **ECST.#OP** to false.
≥ 9000	User-defined codes	The user can execute commands **kill**, **killall**, **disable**, **disableall** with an argument that specifies a user-defined error code. The specified error code is stored in the **MERR** variable.

10.2 Error Indication

10.2.1 Errors in Received Commands

If the controller receives a correct command, it responds with the normal prompt (colon).

If a command cannot be executed for any reason, the controller responds with the error prompt. The error prompt contains the "**?**" character and 4-digit error code. For example:

```
VEL(0) = 1000
:                                   Normal prompt - the command was successful
FPOS(0) = 0
?2020                               Error - attempt of assignment to a read-only variable
```

10.2.2 Errors in ACSPL+ Programs

If the controller executes an ACSPL+ program and an error occurs due to the program, the controller does the following:

- ❑ Aborts the erroneous program
- ❑ Stores the error code in the corresponding element of the **PERR** array

- Stores the line number that caused the error in the corresponding element of the **PERL** array
- Activates the **#PROG** fault. The default controller response to the fault is to kill all executed motions.

No error prompt is issued. You must analyze the state of the buffer in order to detect program errors. For example:

`?1`	Query the state of buffer 1
`Buffer 1: 78 lines, run-time error 3077 in line 37`	
`?PERR(1)`	The state also can be monitored through variable **PERR**
`3077`	

10.2.3 Motion Termination Codes

A motion executed in the controller can terminate for different reasons:

- The motion comes to its final point - normal termination
- You interrupt the motion with a **halt** or **kill** command
- The controller detects a fault that requires motion kill

In all cases the controller stores the termination reason in the corresponding element of the **AERR** (Axis Error) variable. No error prompt is issued. You have to analyze the state of the axis in order to detect motion termination. For example:

`?$0`	Query of the state of axis 0.
`Motor 0(X): enabled, idle, in position` **`Axis 0(X): motion was killed by user`**	
`?AERR(0)`	The state also can be monitored through the **AERR** variable.
`5002`	The termination code - motion was killed by user.
`?$13`	Query of the state of axis 13.
`Motor 13(Axis13): enabled, idle, in position` **`Motor 13(Axis13): motion failed, reason 5011`**	
`?AERR(13)`	The state also can be monitored through the **AERR** variable.
`5011`	The termination code - Left Limit fault

10.2.4 Motion Termination and Motor Disable Codes

A motion executed in the controller can terminate for the different reasons:

- The motion comes to its final point - normal termination
- You or the ACSPL+ program interrupts the motion with a **halt**, **kill**, or **killall** command
- A motor involved in the motion is disabled for any reason
- The controller detects a fault that requires motion kill

The controller disables a motor for the following reasons:

- You or the ACSPL+ program executes a **disable** or **disableall** command
- The controller detects a fault that requires motor disable

If a motor is disabled or a motion is killed due to a fault, the controller stores the reason in the corresponding element of the **MERR** (Motor Error) variable.

You can also specify a termination code for commands **kill**, **killall**, **disable** or **disableall**. If one of the commands is used with a non-zero reason argument, the specified code is stored in one or more corresponding elements of **MERR** as a termination/disable code.

In a case where no error prompt is issued, you must analyze the state of the motor in order to detect whether the motor was disabled or a motion was killed abnormally. For example:

`?Z`	Query of the state of motor 2
`Motor 2(Z): disabled` **`  Disable reason: 5023 - Critical Position Error`** **`Axis 2(Z): motion was killed because a motor was disabled`**	
`?MERR(2)`	The state also can be monitored through variable **MERR**
`5023`	The error code - motor was disabled because of critical position error fault

10.2.5 Getting Extended Drive Fault Status

Upon receiption of a Drive Alarm signal, the controller stores a general Drive Alarm code (5019) in the **MERR** variable. The extended Drive Fault status code can be obtained by executing the **getconf(246, Axis)** function (see *SPiiPlus Command & Variable Reference Guide*).

The following fault codes are returned:

- 5061 – Short circuit
- 5064 – Power supply too high
- 5065 – Temperature too high
- 5069 – Power down
- 5071 – Drive not ready
- 5072 – Over current

Using the **getconf** funtion, the faults 5064, 5065, 5069, 5071 can be read before **enable** command is executed. In order to clear the Drive Fault status code, use the **setconf** function: **setconf(246, Axis, 0)**. This function clears the fault status on all axes that relate to the DDM3U Motor Drive that handles the specified axis.

11 Application Examples

This chapter contains practical examples for use in SPiiPlus applications.

For further examples, registered users may go to **www.AcsMotionControl.com** and download pertinent documents (selected from the **Categories** dropdown list on the page).

11.1 Encoder Error Compensation With Constant Step

Assume an axis was calibrated with a laser interferometer. The calibration process includes positioning to the points equally spaced according to the encoder feedback, and measuring the exact positions by the laser interferometer. The calibration points start from the coordinate 10000 and follow each 1000 counts. The last calibration point is 20000.

The calibration produced the following table:

Feedback	10000	11000	12000	13000	14000	15000	16000	17000	18000	19000	20000
Actual	10012	10998	11985	12981	13997	15007	16013	17023	18005	18993	19991
Error	+12	-2	-15	-19	-3	+7	+13	+23	+5	-7	-9

Only the third row is entered to a controller variable and is stored in the flash **X_ERROR** file. Details of the calibration routine, which can be implemented using ACSPL+, are not discussed here.

An application that uses the file may provide an initialization routine like this:

```
real X_ERR(11)                                  Declare real array X_ERR with 11
                                                members.
AUTOEXEC:                                       ACSPL+ label initializing the routine on
                                                start-up.
write X_ERR                                     Write to and read from the calibration table
read X_ERR                                      in the flash memory.
connect RPOS(0)0 = APOS(0) - map (APOS(0), X_ERR, 10000, 1000)
stop                                            Finish initialization
```

The **connect** function specifies that the reference position be calculated by subtracting the interpolated error from the desired position so that the actual value will be closer to the desired value.

11.2 Encoder Error Compensation With Arbitrary Step

Assume in the above example that the calibration routine does not calculates an error, but writes the first two lines from the above table to the **X_CALIBR** flash file. The table is stored as an array with 2 rows and 11 columns.

In this case the application can implement an initialization routine like this:

`real X_CAL(2)(11)`	Declare real 2x11 matrix **X_CAL**.
`AUTOEXEC:`	ACSPL+ label initializing the routine on start-up.
`write X_CAL` `read X_CAL`	Write to and read from calibration table from the flash memory.
`connect RPOS(0) = mapby2(APOS(0), X_CAL)`	
`stop`	Finish initialization

And the **X_CALIBR** file can also contain a table with non-uniform points.

11.3 Backlash Compensation

Assume that the 0 axis has a backlash of 20 counts. The following **connect** command compensates for the backlash:

```
connect RPOS(0) = APOS(0) + 10*dsign(RVEL(0), 0, 0)
```

By using this **connect**, the value added to desired position changes immediately when the direction of the motion changes. In many cases such jumps in the desired position are harmful. In this case the third parameter in the **dsign** function (see *SPiiPlus Command & Variable Reference Guide*) can be used to gradually implement the backlash compensation. In the following example the backlash compensation is introduced by small steps, so that the compensation growth to the entire value by 20 milliseconds:

```
connect RPOS(0) = APOS(0) + 10*dsign(RVEL(0), 0, 20)
```

If the X axis executes master-slave motion slaved to some physical value like encoder feedback, the **RVEL** value contains noise that can cause undesirable switching of the backlash sign. In this case the second parameter of the **dsign** function can be used to introduce antibouncing effect. In the following example the backlash compensation changes its sign only if **RVEL** holds its new sign for more than 10 milliseconds:

```
connect RPOS(0) = APOS(0) + 10*dsign(RVEL(0), 10, 20)
```

11.4 Compensation of Encoder Error and Backlash

An arbitrary expression can be used as an argument in the dsign function and in the map-functions. It ensures combining different compensation function and other required transformations in one **connect** command. The following example combines error and backlash compensations from the above examples:

```
real X_CAL(2)(11)                                  Declare real 2x11 matrix X_CAL.
AUTOEXEC:                                          ACSPL+ label initializing the routine on
                                                   start-up.
write X_CAL                                        Write to and read from calibration table from
read X_CAL                                         the flash memory.
connect RPOS(0) = mapby2(APOS(0)+10*dsign(RVEL(0), 10, 20), X_CAL)
stop                                               Finish initialization
```

11.5 Cam Motion

Assume that the 1 axis must provide a cam motion following the 0 axis. The **CAMTABLE** matrix in the nonvolatile (flash) memory contains 2 rows and 1000 columns. Each column contains an X coordinate (for axis 0) in the first row and the corresponding Y coordinate (for axis 1) in the second row. The X coordinates in the first row can be spaced either equally or non-equally.

The following fragment initializes the cam motion:

```
global real CAMTABLE(2)(1000)                      Declare 2x1000 matrix CAMTABLE.
AUTOEXEC:
write CAMTABLE                                     Write to and read from calibration table
read CAMTABLE,CAMTABLE                             from the flash memory.
master MPOS(1) = mapby1(FPOS(0), CAMTABLE)         Define master value via CAMTABLE
slave Y                                            Start master-slave motion
```

11.6 Joystick

Assume that a joystick that controls the motion of an X,Y table is connected to analog inputs **AIN(0)**, **AIN(1)**. The velocity of each coordinate must be proportional to the corresponding analog input. Analog input lower than 20 counts must be ignored to avoid motion due to analog drift or bias. The X (axis 0) motion is limited in the range from -500 to 100000 counts. The Y (axis 1) motion is limited in the range from -500 to 500000 counts.

The following program fragment initializes the joystick motion:

```
real JK
JK = 10                                                  Joystick factor
master MPOS(0) = intgr(AIN(0), 20, -500, 100000)  Define 0 master
master MPOS(1) = intgr(AIN(1), 20, -500, 500000)  Define 1 master
slave Y                                                  Start master-slave motion
```


ACS MotionControl
Your Competitive Advantage