SPiiPlus C Library Reference

Programmer's Guide

February 2023

Document Revision: 3.13.01

ACS MOTION CONTROL

SPiiPlus C Library Reference

Release Date: February 2023

PATENTS

Israel Patent No. 235022
US Patent Application No. 14/532,023
Europe Patent application No.15187586.1
Japan Patent Application No.: 2015-193179
Chinese Patent Application No.: 201510639732.X
Taiwan(R.O.C.) Patent Application No. 104132118
Korean Patent Application No. 10-2015-0137612

www.acsmotioncontrol.com

support@acsmotioncontrol.com

sales@acsmotioncontrol.com

Related Documentation

Documents listed in the following table provide additional information related to this document.

Authorized users can download the latest versions of the documents from ACS Downloads.

Document	Description
SPiiPlus Setup Guide	Communication, configuration and adjustment procedures for SPiiPlus motion control products.
SPiiPlus Command & Variable Reference Guide	Command and variables of high level language for programming SPiiPlus controllers.
SPiiPlus COM Library Reference Guide	COM Methods, Properties, and Events for Communication with the Controller
SPiiPlus Utilities User Guide	The SPiiPlus Utilities described in this document are: > SPiiPlus User Mode Driver (UMD) > SPiiPlus Upgrader > SPiiPlus Simulator
PEG and MARK Operations Application Notes	A guide to the PEG and MARK operations for the SPiiPlus family of controllers.
SPiiPlus ADK Suite v2.50 Release Notes	Describes new features and changes introduced since the last SPiiPlus ADK Suite version 2.40 release.

Conventions Used in this Guide

Text Formats

Format	Description
Bold	Names of GUI objects or commands
BOLD + UPPERCASE	ACSPL+ variables and commands
`Monospace + grey background`	Code example
Italic	Names of other documents
Blue	Hyperlink
[ ]	In commands indicates optional item(s)
|	In commands indicates either/or items

Flagged Text

	**Note** - includes additional information or programming tips.
	**Caution** - describes a condition that may result in damage to equipment.
	**Warning** - describes a condition that may result in serious bodily injury or death.
	**Model** - highlights a specification, procedure, condition, or statement that depends on the product model
	**Advanced** - indicates a topic for advanced users.

Revision History

Date	Revision	Description
February 2023	3.13.01	New Release, document acsc_GetEthernetCardsExt
November 2022	3.13	New version, removed obsolete PEG functions, add new EtherCAT functions
September 2022	3.12.01	Corrected acsc_SaveApplication example
May 2022	3.12	New Version Release, NURBS, SPATH, PEG functions
November 2021	3.11.01	New Version Release
July 2020	3.01	Add note about starvation parameter in extended motion Remove deprecated functions
June 2020	3.00	Added SPiiPlusSC errors to table
March 2019	2.70	Removed Program Management functions controlling breakpoints meant for internal use only
July 2018	2.60	Replaced "X" in programming example with an axis number, as necessary. Updated extended segmented motion functions: > acsc_ExtendedSegmentArc1 > acsc_ExtendedSegmentArc2 Updated Position Event Generation (PEG) Functions: > acsc_PegIncNT > acsc_PegRandomNT
January 2018	2.50.10	Corrected program example for acsc_SegmentArc2Ext Corrected program example for acsc_SegmentArc1Ext Corrected program example for acsc_ BlendedSegmentMotion Corrected program example for acsc_BlendedArc1 Corrected program example for acsc_ BlendedSegmentMotion

Date	Revision	Description
December 2017	2.50	Updated extended segmented motion Added blended segmented motion Added motion flag definitions Added acsc_CommutExt
September 2017	2.40.10	Updated number of buffers to 63
June 2017	2.40	Updated for SPiiPlus ADK Suite v2.30: Data collection function acsc_WaitCollectEnd is replaced with the function acsc_WaitCollectEndExt. Added new bits for ECST
August 2016	2.30	Updated for SPiiPlus ADK Suite v2.30: Removed functions: acsc_OpenCommDirect Added functions: acsc_OpenCommSimulator & acsc_CloseSimulator Corrected syntax from ASCS_COMM_AUTORECOVER_HW_ERROR to ACSC_COMM_AUTORECOVER_HW_ERROR
September 2014	01	First Release

Table of Contents

List Of Figures

List of Tables

1. Introduction

1.1 Document Scope

The SPiiPlus C Library supports the creation of a user application that operates on a PC host computer and communicates with SPiiPlus motion controllers. The SPiiPlus C Library implements a rich set of controller operations and conceals from the application the complexity of low-level communication and synchronization with the controller.

2. SPiiPlus C Library Overview

2.1 Operation Environment

The SPiiPlus C Library supports the following Microsoft® Windows® environments:

- Windows 7 SP1 (x86 and x64)
- Windows 8 (x86 and x64)
- Windows 8.1 (x86 and x64)
- Windows 10
- Windows Server 2008 R2 SP1 (x64)
- Windows Server 2012 R2 (x64)
- Windows Server 2016 R2 (x64)
- Windows Server 2019 (x64)

Windows XP and Windows Server 2003 are no longer supported.

2.2 Communication Log

2.2.1 Run-Time Logging

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.

2.2.2 Log Types

The user 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** – there is a background thread that takes care of periodic file updates. It reads the binary buffer and performs text formatting to the file.

2.3 C Library Concept

The C Library is a software package that allows Host-based applications to communicate with the SPiiPlus controller in order to program, send commands, and query controller status.

The C Library includes user-mode and kernel-mode drivers that perform various communication tasks.

The host application is provided with a robust C Function API to make calls the C Library which in turn communicates with the SPiiPlus Controller through the controller drivers. The controller then returns the reply to the to the caller application.

The host application may contact C Library from remote location by setting its IP address using the acsc_SetServerExtLogin function.

Up to four host applications may communicate with the controller simultaneously via a single physical connection.

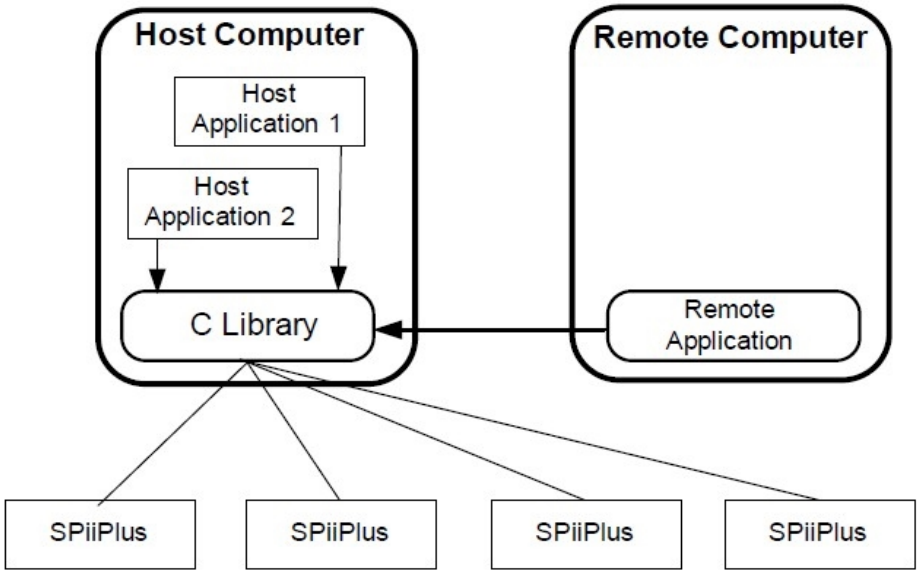

Figure 3-1. C Library Concept

2.4 Communication Channels

The SPiiPlus C Library supports all communication channels provided by SPiiPlus motion controllers:

- Serial (RS-232)
- Ethernet (point-to-point and Network)
- PCI Bus

2.5 Controller Simulation

The SPiiPlus Utilities include an controller simulator application that operates on the same PC as the user application. The Simulator provides for execution of a user application without the physical controller for debugging and demonstration purposes. For details see the *SPiiPlus Utilities User Guide*.

2.6 Programming Languages

The library directly supports development of C/C++ applications. Visual Basic®, C# or other languages can also be used with a little additional effort. For languages other than C/C++, the SPiiPlus COM library is recommended.

2.7 Supplied Components

The library includes a DLL, a device driver, an import library, and a header file for C/C++ compilers.

2.8 Highlights

- **Unified support of all communication channels (Serial, Ethernet, PCI Bus)**

All functions except **acsc_OpenComm*** ** functions are identical for all communication channels. The user application remains substantially the same and works through any of the available communication channels.

> **Controller simulator as an additional communication channels**

All library functions can work with the Simulator exactly as with the actual controller. The user application activates the simulator by opening a special communication channel. The user is not required to change his application in order to communicate with the Simulator.

> **Support of multi-threaded user application**

The user application can consist of several threads. Each thread can call SPiiPlus C Library functions simultaneously. The library also provides special functions for the synchronization SPiiPlus C functions called from concurrent threads.

> **Automatic synchronization and mutual exclusion of concurrent threads**

Both waiting and non-waiting calls of SPiiPlus C functions can be used from different threads without any blocking or affect one to another. The library provides automatic synchronization and mutual exclusion of concurrent threads so the threads are not delayed one by another. Each thread operates with its maximum available rate.

> **Support of concurrent multiple communication channels to one controller**

Usually different communication channels are connected to different controllers. However, two or more communication channels can be connected to one controller.

The User Mode Driver (UMD) supports up to 10 different communication channels, both local and remote, and for every channel up to four "logical" connections can be opened. For example, two serial communication channels can be opened to the same controller (they are opened as "logical" channels) along with two PCI channels. The same is true for TCP/IP network/point-to-point connections.

> **Acknowledgement for each controller command**

The library automatically checks the status of each command sent by the user application to the controller. The user application can check the status to confirm that the command was received successfully. This applies for both waiting and non-waiting calls.

> **Communication history**

The library supports the storage of all messages sent to and received from the controller in a memory buffer. The application can retrieve the full or partial contents of the buffer and can clear the history buffer.

> **Separate processing of unsolicited messages**

Most messages sent from the controller to the host are responses to the host commands. However, the controller can send unsolicited messages, for example, because of executing the disp command. The library separates the unsolicited messages from the overall message flow and provides special function for handling unsolicited messages.

> **Rich set of functions for setting and reading parameters, motion, program management, I/O ports, safety controls, and other**.

> **Two calling modes**

Most library functions can be called in either waiting or non-waiting mode. In waiting mode, the calling thread does not continue until the controller acknowledges the command execution. In non-waiting mode, a function returns immediately and the actual work of sending the command and receiving acknowledgement is performed by the internal thread of the library.

> **Debug Tools**

The library provides different tools that facilitate debugging of the user application. The simulator and the communication history mentioned above are the primary debugging tools. The user can also open a log file that stores all communications between the application and the controller.

> **Setting user callback functions for predefined events**

The possibility exists to set a callback function that will be called when a specified event occurs in the controller. This lets you define a constant reaction by the user host application to events inside the controller without polling the controller status (see Callbacks).

> **Wait-for-Condition Functions**

To facilitate user programming, the library includes functions that delay the calling thread until a specific condition is satisfied. Some of the functions periodically pole the relevant controller status until the condition is true, or the time out expired. Some of these functions are based on the callback mechanism, see Callbacks. The functions with this option are:

> acsc_WaitMotionEnd
> acsc_WaitLogicalMotionEnd
> acsc_WaitProgramEnd
> acsc_WaitInput

These functions will use the callback mechanism if the callback to the relevant event is set, otherwise polling is used.

2.9 Use of Functions

Each library function performs a specific controller operation. To perform its task, the function sends one or more commands to the controller and validates the controller responses.

Because the SPiiPlus C functions follow the C syntax and have self-explaining names, the application developer is not required to be an expert in ACSPL+ language. However, the most time-critical part of an application often needs to be executed in the controller and not in the host. This part still requires ACSPL+ programming.

To use the SPiiPlus C Library functions from C/C++ environment, it is necessary to include the header file ACSC.h and the import library file ACSC_x86.lib or ACSC_x64.lib, whichever is appropriate, to the project.

An example of a function is the following that implements a motion to the specified point:

```
int acsc_ToPoint(HANDLE Handle, int Flags, int Axis, double Point, ACSC_
WAITBLOCK* Wait)
```

Where:

> **Handle** is a communication handle returned by one of the **acsc_OpenComm*** ** functions.
> **Flags** are a bit-mapped parameter that can include one or more motion flags.

For example:

ACSC_AMF_WAIT	Plan the motion, but don't start until the **acsc_Go** function is called.
ACSC_AMF_RELATIVE	The **Point** value is relative to the end-point of the previous motion. If the flag is not specified, the **Point** specifies an absolute coordinate.

- **Axis** is an axis constant (see Axis Definitions) of the motion.
- **Point** is a coordinate of the target point.
- **Wait** is used for non-waiting calls. Non-waiting calls are discussed in the next section.

2.10 Callbacks

There is an option to define an automatic response in the user application to several events inside the controller. The user specifies a function that will be called when certain event occurs. This approach helps user application to avoid polling of the controller status and only to execute the defined reaction when it is needed.

The library may set several callbacks in the same time. Every one of them runs in its own thread and doesn't delay the others.

Callbacks are supported in all communication channels. The library hides the difference from the application, so that the application handles the callbacks in all channels in the same way. The events that may have a callback functions are:

- Hardware detected events
  - PEG
  - MARK1 and MARK2
  - Emergency Stop
- Software detected events
  - Physical motion end
  - Logical motion end
  - Motion failure
  - Motor failure
  - ACSPL+ program end
  - ACSPL+ line execution
  - ACSPL + "interrupt" command execution
  - Digital input goes high
  - Motion start
  - Motion profile phase change
  - Trigger function detects true trigger condition
  - Controller sent complete message on a communication channel

2.10.1 Timing

When working with PCI bus, the callbacks are initiated through physical interrupts generated by the controller. In the Simulator, the interrupt mechanism is emulated with OS mechanisms. In all other kinds of communication, the controller sends an alert message over the communication channel in order to inform the host about the event.

Although the implementation is transparent, the timing is different varies for each communication channel as follows:

- In PCI communication, the callbacks are based upon PCI interrupts and response is very fast (sub-millisecond level).
- In all other channels, callback operation includes sending/receiving a message that requires much more time. Specific figures depend on the communication channel rate.

From the viewpoint of the Callback Mechanism, all communication channels are functionally equivalent, but differ in timing.

2.10.2 Hardware Interrupts

Hardware events (Emergency Stop, PEG and MARK) are detected by the controllers HW and an interrupt on PCI bus is generated automatically, while on other communication channels those events are recognized by the firmware and only then an alert message may be sent. That is why there is a difference in the definition of the Event condition for different communication channels.

Callback	Condition of PCI Interrupt	Condition of Alert Message (all channels except PCI)
Emergency stop	The interrupt is generated on positive or negative edge of the input ES signal.The edge is selected by **S_SAFINI.#ES** bit.	The message is sent when the **S_FAULT.#ES** bit changes from zero to one. The message is disabled if **S_FMASK.#ES** is zero.
Mark 1 and Mark 2	The interrupt is generated on positive edge of the corresponding Mark signal.	The message is sent when the corresponding **IST.#MARK** or **IST.#MARK2** bit changes from zero to one.
PEG	The interrupt is generated on negative edge of PEG pulse.	The message is sent when corresponding **AST.#PEG** bit changes from one to zero.

2.11 Dual-Port RAM (DPRAM)

DPRAM is not supported in SPiiPlus products.

The DPRAM is a memory block that is accessible from the host and from the controller. This feature provides fast data exchange between the host and the controller.

The SPiiPlus controller provides 1024 bytes of dual-port ram memory (DPRAM). Relative address range of DPRAM is from byte 0 to byte 0x3FF.

The first 128 bytes (relative addresses from 0 to 0x080) are reserved for system use. The rest of the memory is free for the user needs.

The DPRAM functions are available with any communication channel, however it is important to remember that only PCI bus communication provide real physical access to controllers DPRAM and works very fast (sub-millisecond level).

In all other channels, the DPRAM operation is simulated. Each operation includes communication with the controller. Specific figures depend on the communication channel rate.

Using DPRAM communication in a non-PCI communication channel is recommended if an application is primarily intended for PCI channel, but requires full compatibility with other communication channels.

2.12 Shared Memory

Shared Memory is applicable only to the SPiiPlusSC. For details of the SPiiPlus SC see the *SPiiPlusSC Motion Controller User Guide*.

Shared Memory refers to a 100 KByte section of the memory where variables used by both the SPiiPlus SC RTOS processes and the Windows processes are stored. Access to the shared memory by the Windows applications does not affect the RTOS execution. Special C read and write functions have been incorporated to access the Shared Memory (see Shared Memory Functions).

2.13 Non-Waiting Calls

There are three possible approaches regarding when a library function returns control to the calling thread:

> Waiting call

The function waits for the controller response and then returns. For many commands, the controller response does not signal the completion of the operation. The controller response only acknowledges that the controller accepted the command and started the process of its execution. For example, the controller responds to a motion command when it has planned the motion, but has not executed yet.

> Non-waiting call

The library function initiates transmission of the command to the controller and returns immediately without waiting for the controller response. An internal library thread sends the command to the controller and retrieves the result. To get the result of operation the application calls the acsc_WaitForAsyncCall function.

> Non-waiting call with neglect of operation results

The same as the previous call, only the library does not retrieve the controller response. This mode can be useful when the application ignores the controller responses.

Most library functions can be called in either waiting or non-waiting mode. The pointer **Wait** to the **ACSC_WAITBLOCK** structure provides the selection between waiting and non-waiting modes as follows:

> Zero Wait (NULL character) defines a waiting call. The function does not return until the controller response is received.

Do not use '0' as the Null character.

> If **Wait** is a valid pointer, the call is non-waiting and the function returns immediately.

> If **Wait** is ACSC_IGNORE, the call is non-waiting and will neglect of the operation result.

ACSC_WAITBLOCK is defined as follows:

Structure: ACSC_WAITBLOCK {HANDLE Event; int Ret;};

When a thread activates a non-waiting call, the library passes the request to an internal thread that sends the command to the controller and then monitors the controller responses. When the controller responds to the command, the internal thread stores the response in the internal buffers. The calling thread can retrieve the controller response with help of the acsc_WaitForAsyncCall function and validate the completion result in the **Ret** member of the structure.Up to 256 non-waiting calls can be activated before any **acsc_WaitForAsyncCall** is called. It is important to understand that **acsc_WaitForAsyncCall** must be called for every non-waiting call. Otherwise, the response will be stored forever in the library's internal buffers. A call, which is called when more then 256 calls are already activated is delayed for a certain time and waits until **acsc_WaitForAsyncCall** is called by one of the previous calls. If the time expires, an **ACSC_COMMANDSQUEUEFULL** error is returned.

If you work with multiple non-waiting calls and the **ACSC_COMMANDSQUEUEFULL** error pops up all the time, the structure of your application is too demanding. This means that you are trying to activate more than 256 calls without retrieving the results.

If the error message pops up occasionally, try increasing the timeout.

Time-out is controlled by acsc_GetQueueOverflowTimeout and acsc_SetQueueOverflowTimeout functions.

The following example shows how to perform waiting and non-waiting calls. In this example the acsc_WaitForAsyncCall function was used. Any function that has **Wait** as a parameter can be used to perform waiting and non-waiting calls.

```
#include "ACSC.h"
Char* cmd = "?$\r";                    // get motors state
char buf[101];
int Received;
ACSC_WAITBLOCK wait;
// example of the waiting call of acsc_Transaction
```

```
if (!acsc_Transaction( Handle,          // communication handle
                       cmd,             // pointer to the buffer that
                                        // contains command to be executed
                       strlen(cmd),     // size of this buffer
                       &Received,       // number of characters that were
                                        //actually received
                       NULL             // waiting call
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
// example of non-wainig call of acsc_Transaction if (acsc_Transaction
(Handle,cmd,strlen(cmd),buf,100,&Received,&wait))
{
       // something doing here
       ….
       // retrieve controller response
       if (acsc_WaitForAsyncCall(Handle, buf, &Received, &wait, 5000))
{
               buf[Received] = '\0';
               printf("Motors state: %s\n", buf);
       }
       else
       {
               acsc_GetErrorString(Handle, wait.Ret, buf, 100, &Received);
               buf[Received] = '\0';
               printf("error: %s\n", buf);
       }
}
else
{
printf("transaction error: %d\n", acsc_GetLastError());
}
// Example of non-waiting call of acsc_Transaction with neglect of the
// operation result. Function does not wait for the controller response.
// The call of acsc_WaitForAsyncCall has no sense because it does not
// return the controller response for this calling mode.
If (acsc_Transaction(  Handle,cmd,strlen(cmd),buf,
                       100, &Received, ACSC_IGNORE))
{
       printf("transaction error: %d\n", acsc_GetLastError());
}
```

3. Using the SPiiPlus C Library

3.1 Library Structure

The C Library is built from several levels, from Kernel-mode drivers on one end, to high level C function APIs on the other, and include:

- **ACSPCI32.SYS** (for 32-bit)**, ACSPCI64.SYS** (for 64-bit), **WINDRVR6.SYS** – Kernel-mode drivers for low-level communication support. These drivers are automatically installed and registered when the user installs the SPiiPlus software package. These drivers are required for communication with the controller through the PCI bus.
- **ACSCSRV.EXE –** User-mode driver for high-level communication support. When this driver is active, there is an icon in the notification area at the bottom-right corner of the screen. This driver is necessary for all communication channels. The driver is automatically installed and registered when the user installs the SPiiPlus software package**.**
- **ACSCL_X86.DLL** – Dynamic Link Library that contains the API functions. The DLL is installed in the SYSTEM32 directory, so it is accessible to all host applications. It is designed to operate in a 32-bit environment.
- **ACSCL_X64.DLL** – Dynamic Link Library that contains the API functions. The DLL is installed in the SYSTEM32 directory, so it is accessible to all host applications. It is designed to operate in a 64-bit environment.
- **ACSCL_X86.LIB** – Static LIB file required for a C/C++ project to access the DLL functions. It is designed to operate in a 32-bit environment.
- **ACSCL_X64.LIB** – Static LIB file required for a C/C++ project to access the DLL functions. It is designed to operate in a 64-bit environment.

A 32-bit software development should link against **ACSCL_X86.LIB**.

A 64-bit software development should link against **ACSCL_X64.LIB**

- **ACSC.H –** C header file with API functions and Constant declarations.

3.2 Building C/C++ Applications

To facilitate using the C Library in user applications, the installation includes the **ACSC.H**, and **ACSC_x86.LIB** or **ACSC_x64.LIB** files. The files are not required for running the C Library, and are only used for building user applications.

In order to use the C Library functions in your application, proceed as follows:

1. Copy files from Program Files\ACS Motion Control\SPiiPlus …\ACSC to the project directory. Include file ACSCL_X86.LIB, or ACSCL_X86.LIB (as appropriate), in your C/C++ project.
2. Include file ACSC.H in each project file where the functions must be called. Use the statement #include "ACSC.H" (see the example program in Non-Waiting Calls).
3. Once the application that includes ACSCL_X86.LIB (or ACSCL_X64.LIB) file is activated, it locates file ACSCL_X86.DLL (or ACSCL_X64.DLL), so the appropriate.DLL file must be

accessible to the application. The library installation puts the file in the SYSTEM32 directory, where it can be found by any application.

3.3 Redistribution of User Application

A user application that calls C Library functions can be immediately executed on a computer where the SPiiPlus software package was previously installed.

If the application is executed on a computer without the SPiiPlus software package installation, the user must install several library and support files. This process is called "redistribution".

3.3.1 Redistributed Files

The files for redistribution are found in "Program Files\ACS Motion Control\SPiiPlus …****Redist," and include:

- **ACSCL_X86.DLL** – ACS Motion Control© C Library API for 32-bit environment
- **ACSCL_X64.DLL** – ACS Motion Control© C Library API for 64-bit environment
- **MFC90.DLL, MSVCR90.DLL**, **Microsoft.VC90.CRT.manifest**, **Microsoft.VC90.MFC.manifest** – Microsoft® C Runtime Libraries
- **WINDRVR6.SYS** –Jungo© WinDriverKernel Mode Device Driver
- **WDAPI1011.DLL** – Jungo© WinDriver Library used by **ACSCSRV.EXE**
- **ACSPCI32.SYS** – ACS Motion Control© kernel-mode PCI Device Driver (for 32-bit environment)
- **ACSPCI64.SYS** – ACS Motion Control© kernel-mode PCI Device Driver (for 64-bit environment)
- **ACS.ESTOP.EXE** – ACS Motion Control© SPiiPlus Emergency Stop
- **ACSCSRV.EXE** – ACS Motion Control© SPiiPlus User Mode Driver (UMD)
- **ACS.AUTOINSTALLER.EXE** – ACS Motion Control© SPiiPlus PCI Driver AutoInstaller

3.3.2 File Destinations

The files should be copied to various places, depends on the Operating System.

These files are common to all Microsoft Windows versions. Place these files in the relevant Windows system directory:

- ACSCL_X86.DLL, or
- ACSCL_X64.DLL

The following files differ with each Windows version:

Place the following files in the SYSTEM32\DRIVERS:

- ACSPCI32.SYS, or
- ACSPCI64.SYS

In order to provide WinDriver information for Windows "Plug and Play," redistribute INF files as follows:

- ACSPCI.INF
- WINDRVR6.INF

The **INF** files are required only for registration process; they should be placed in well-known destinations.

Copy **ACSCSRV.EXE**, **ACS.EStop.exe**, **WDAPI1011.DLL, MFC90.DLL**, **MSVCR90.DLL**, **Microsoft.VC90.CRT.manifest**, and **Microsoft.VC90.MFC.manifest** to target machine. The exact place is not important; however, it is convenient to put it in the application directory. The main thing is to configure Windows to launch **ACSCSRV.EXE** on start-up. The preferred way to do so is to make an addition to the registry key as follows:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run a new
string value named "ACSCSRV"
```

The string should contain the full path to the location of **ACSCSRV.EXE**.

On start-up Windows will start the UMD on the current machine for each user that logs in.

3.3.3 Kernel Mode Driver Registration

The easiest way to perform Kernel Mode driver installation and removal is to use **ACS.AutoInstaller.exe**. Manual installation can be performed as follows:

- **32-bit Systems**

If you want to install or remove drivers on a 32-bit system, use files from the "x86" folder.

First, you have to remove old drivers from the registry. Execute the following commands on the target machine and reboot it:

- wdreg.exe -name "Spii" -file Spii stop
- wdreg.exe -name "Spii" -file Spii DELETE
- wdreg.exe -name WinDriver stop
- wdreg.exe -name WinDriver DELETE

Delete these files from the following directories on the target machine:

WINDRVR.SYS, SPII.SYS from C:\WINDOWS\SYSTEM32\DRIVERS.

Place these files in the following directories on the target machine:

ACSPCI32.SYS to C:\WINDOWS\SYSTEM32\DRIVERS.

Put wdreg.exe, difxapi.dll, windrvr6.inf, windrvr6.sys, wd1000.cat, acspci.inf, ACSPCI32.sys, acsx86.cat in a folder on the target machine.

To install drivers to the registry execute the following commands:

- wdreg.exe -inf acspci.inf disable
- wdreg.exe -inf windrvr6.inf disable

- wdreg.exe -inf windrvr6.inf install
- wdreg.exe -name ACSPCI32 install
- wdreg.exe -inf acspci.inf install
- wdreg.exe -inf windrvr6.inf enable
- wdreg.exe -inf acspci.inf enable

To remove drivers from the registry execute the following commands:

- wdreg.exe -inf acspci.inf disable
- wdreg.exe -inf windrvr6.inf disable
- wdreg.exe -name ACSPCI32 uninstall
- wdreg.exe -inf acspci.inf uninstall
- wdreg.exe -inf windrvr6.inf uninstall

- **64-bit Systems**

If you want to install or remove drivers on a 64-bit system, use files from "x64" folder. Place these files to the following directories on the target machine:

ACSPCI64.SYS to C:\WINDOWS\SYSTEM32\DRIVERS.

Put wdreg.exe, difxapi.dll, windrvr6.inf, windrvr6.sys, wd1000.cat, acspci.inf, ACSPCI64.sys, acsamd64.cat in a folder on the target machine.

To install drivers to the registry execute the following commands:

- wdreg.exe -inf acspci.inf disable
- wdreg.exe -inf windrvr6.inf disable
- wdreg.exe -inf windrvr6.inf install
- wdreg.exe -name ACSPCI64 install
- wdreg.exe -inf acspci.inf install
- wdreg.exe -inf windrvr6.inf enable
- wdreg.exe -inf acspci.inf enable

To remove drivers from the registry execute the following commands on the target machine:

- wdreg.exe -inf acspci.inf disable
- wdreg.exe -inf windrvr6.inf disable
- wdreg.exe -name ACSPCI64 uninstall
- wdreg.exe -inf acspci.inf uninstall
- wdreg.exe -inf windrvr6.inf uninstall

4. C Library Functions

This chapter describes each of the functions available in the SPiiPlus C Library. The functions are arranged in functional groups.

For each function there is a:

- **Description** - A short description of the use of the function
- **Syntax** - The calling syntax
- **Arguments** - List and definition of function arguments
- **Return value** - A description of the value, if any, that the function returns
- **Comments** - Where relevant, additional information on the function
- **Example** - Short code example in C language

The functions are grouped in the following categories:

4.1 Communication Functions

The C Library Communication Functions are:

Table 5-1. Communication Functions

Function	Description
acsc_OpenCommSerial	Opens communication via serial port.
acsc_ OpenCommEthernetTCP	Opens communication with the controller via Ethernet using TCP protocol.
acsc_ OpenCommEthernetUDP	Opens communication with the controller via Ethernet using the UDP protocol.
acsc_OpenCommSimulator	Starts up the Simulator and opens communication with it.
acsc_CloseSimulator	Closes the simulator.
acsc_OpenCommPCI	Opens communication with the SPiiPlus PCI via PCI Bus.
acsc_GetPCICards	Retrieves information about the installed SPiiPlus PCI card's.
acsc_CloseComm	Closes communication (for all kinds of communication).

Function	Description
acsc_Transaction	Executes one transaction with the controller, i.e., sends the request and receives the controller response.
acsc_Command	Sends a command to the controller and analyzes the controller response.
acsc_WaitForAsyncCall	Waits for completion of asynchronous call and retrieves a data.
acsc_CancelOperation	Cancels any asynchronous (non-waiting) call or all operations with the specified communication handle.
acsc_GetEthernetCardsExt	Detects available controllers through a standard Ethernet connection.
acsc_SetServerExtLogin	Defines remote communication server and provides login data.
acsc_GetConnectionsList	Retrieves all currently opened connections on the active server and their details.
acsc_GetConnectionInfo	Retrieve the details of opened communication channel.
acsc_TerminateConnection	Terminates a given communication channel (connection) of the active server.

4.1.1 acsc_OpenCommSerial

Description

The function opens communication with the controller via a serial port.

Syntax

HANDLE acsc_OpenCommSerial(int Channel, int Rate)

Arguments

Channel	Communication channel: 1 corresponds to COM1, 2 – to COM2, etc.
Rate	Communication rate in bits per second (baud). This parameter must be equal to the controller variable **IOBAUD** for the successful link with the controller. If **ACSC_AUTO** constant is passed, the function will automatically determine the baud rate.

Return Value

If the function succeeds, the return value is a valid communication handle. The handle must be used in all subsequent function calls that refer to the open communication channel.

If the function fails, the return value is **ACSC_INVALID (-1)**.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

After a channel is open, any SPiiPlus C function works with the channel irrespective of the physical nature of the channel.

Example

```
HANDLE Handle = OpenCommSerial(1,       // Communication port COM1
        115200                          // Baud rate 115200
        );
if (Handle == ACSC_INVALID)
{
        printf("error opening communication: %d\n", acsc_GetLastError());
}
```

4.1.2 acsc_OpenCommEthernetTCP

Description

The function opens communication with the controller via Ethernet using TCP protocol.

Syntax

int acsc_OpenCommEthernetTCP(char* Address, int Port)

Arguments

Address	Pointer to a null-terminated character string that contains the network address of the controller in symbolic or TCP/IP dotted form.
Port	You can use either: ACSC_SOCKET_STREAM_PORT or Define it using the ACSPL+ variable: **TCPPORT** (see *SPiiPlus ACSPL+ Command & Variable Reference Guide*)

Return Value

If the function succeeds, the return value is a valid communication handle. The handle must be used in all subsequent function calls that refer to the open communication channel.

If the function fails, the return value is **ACSC_INVALID (-1)**.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

None.

Example

```
HANDLE hComm=(HANDLE)-1;
int Port = 703;
hComm = acsc_OpenCommEthernetTCP("10.0.0.1",Port);
if (hComm == ACSC_INVALID)
{
        printf("Error while opening communication: %d\n",
        acsc_GetLastError());
        return -1;
}
```

4.1.3 acsc_OpenCommEthernetUDP

Description

The function opens communication with the controller via Ethernet using the UDP protocol.

Syntax

int acsc_OpenCommEthernetUPD(char* Address, int Port)

Arguments

Address	Pointer to a null-terminated character string that contains the network address of the controller in symbolic or TCP/IP dotted form.
Port	You can use either: Ethernet Communication Options or Define it using the ACSPL+ variable: **UDPPORT** (see *SPiiPlus ACSPL+ Command & Variable Reference Guide*)

Return Value

If the function succeeds, the return value is a valid communication handle. The handle must be used in all subsequent function calls that refer to the open communication channel.

If the function fails, the return value is **ACSC_INVALID (-1)**.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

None

Example

```
HANDLE hComm=(HANDLE)-1;
int Port = 704;
hComm = acsc_OpenCommEthernetUDP("10.0.0.1",Port);
if (hComm == ACSC_INVALID)
```

```
{
        printf("Error while opening communication: %d\n",
        acsc_GetLastError());
        return -1;
}
```

4.1.4 acsc_OpenCommSimulator

Description

The function executes the SPiiPlus stand-alone simulator via SPiiPlus User ModeDriver in the case that it is not running.

The function connects to the simulator via TCP/IP protocol.

Syntax

HANDLE acsc_OpenCommSimulator()

Return value

If the function succeeds, then the return value is a HANDLE that is used as a standing connection to the simulator.

If the function fails, then the return value is ACSC_INVALID (-1) and an appropriate error is set.

Error codes (in case of failure)

ACSC_SIMULATOR_RUN_EXT – if the ports set for simulator in UMD are taken by another application.

ACSC_SIMULATOR_NOT_SET – in case a simulator execution attempt was made without setting a simulator executable and default simulator executable was not found.

Extended error information can be obtained by calling **acsc_GetLastError**.

Example

```
// open communication (and open if needed) with SPiiPlus simulator
HANDLE Handle = acsc_OpenCommSimulator();
if (Handle == ACSC_INVALID)
{
      printf("error opening communication: %d\n", acsc_GetLastError());
}
```

4.1.5 acsc_CloseSimulator

Description

The function Stops the SPiiPlus simulator via UMD in the case that it is running.

Syntax

int acsc_CloseSimulator()

Return value

If the function succeeds,: return value = 1.

In the case of function failure: return value = 0.

Extended error information can be obtained by calling **acsc_GetLastError**.

Error codes (in case of failure)

ACSC_SIMULATOR_NOT_RUN – an attempt to stop simulator was made without it running

4.1.6 acsc_OpenCommPCI

Description

The function opens a communication channel with the controller via PCI Bus.

Up to 4-communication channels can be open simultaneously with the same SPiiPlus card through the PCI Bus.

Syntax

HANDLE acsc_OpenCommPCI(int SlotNumber)

Arguments

SlotNumber	Number of the slot of the controller card. If **SlotNumber** is ACSC_NONE, the function opens communication with the first found controller card.

Return Value

If the function succeeds, the return value is a valid communication handle. The handle must be used in all subsequent function calls that refer to the open communication channel.

If the function fails, the return value is ACSC_INVALID (-1).

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

To open PCI communication the host PC, one or more controller cards must be inserted into the computer PCI Bus.

After a channel is open, any SPiiPlus C function works with the channel irrespective of the physical nature of the channel.

Example

```
// open communication with the first found controller card
HANDLE Handle = acsc_OpenCommPCI(ACSC_NONE);
if (Handle == ACSC_INVALID)
{
        printf("error opening communication: %d\n", acsc_GetLastError());
}
```

4.1.7 acsc_GetPCICards

Description

The function retrieves information about the controller cards inserted in the computer PCI Bus.

Syntax

int acsc_GetPCICards(ACSC_PCI_SLOT* Cards, int Count, int* ObtainedCards)

Arguments

Cards	Pointer to the array of the **ACSC_PCI_SLOT** elements.
Count	Number of elements in the array pointed to by **Cards.**
ObtainedCards	Number of cards that were actually detected.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function scans the PCI Bus for the inserted controller cards and fills the **Cards** array with information about the detected cards.

The Structure **ACSC_PCI_SLOT** is defined as follows:

Structure: **ACSC_PCI_SLOT {int BusNumber; int SlotNumber; int Function;};**

Where:

- BusNumber = bus number
- SlotNumber = slot number of the controller card
- Function = PCI function of the controller card

Within these arguments, **SlotNumber** can be used in the acsc_OpenCommPCI call to open communication with a specific card. Other members have no use in the SPiiPlus C Library.

If no controller cards are detected, the function assigns the **ObtainedCards** with zero and does not fill the **Cards** array. If one or more controller cards are detected, the function assigns the **ObtainedCards** with a number of detected cards and places one **ACSC_PCI_SLOT** structure per each detected card into the **Cards** array.

If the size of **Cards** array specified by the **Count** parameter is less than the number of detected cards, the function assigns the **ObtainedCards** with a number of actually detected cards, but fills only the **Count** elements of the **Cards** array.

Example

```
ACSC_PCI_SLOT CardsList[16];
int DetectedCardsCount;
if (acsc_GetPCICards(CardsList, // pointer to the declared array
                                // to save the detected
                                // cards information
        16,                     // size of this array
        &DetectedCardsCount     // number of cards that were
                                // actually detected
        ))
{
        printf("Found %d SB1218PCI cards\n", DetectedCardsCount);
}
else
{
        printf("error while scanning PCI bus: %d\n", acsc_GetLastError());
}
```

4.1.8 acsc_SetServerExtLogin

Description

The function defines the User Mode Driver (UMD) host IP address, and port along with passing login data.

Syntax

int acsc_SetServerExtLogin(char *IP, int Port, char *Username, char *Password,
 char *Domain)

Arguments

IP	IP address where to look for server
Port	Port number to connect over
Username	Null terminated valid username
Password	Null terminated valid password
Domain	Null terminated domain or workgroup

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

Use the function only if the application needs to establish communication with a controller through a remote computer.

If the function is not called, by default, all connections are established through the local computer. Only a controller connected to the local computer by a serial cable, PCI bus or Ethernet can be accessed.

The function sets the IP address and port number of the User Mode Driver (UMD) host and logs the user in. Once the function is called all **acsc_OpenCommxxx** calls will attempt to establish communication via the UMD that was specified in the most recent **acsc_SetServrExtLogin** call. In order to establish communication via a different UMD host **acsc_SetServrExtLogin** has to be called again.

Applications can simultaneously communicate through several communication servers. Use the following pattern to open communication channels through several servers:

```
Handle01= acsc_OpenComxxx(…);    // Open all channels with controllers
                                 // connected to the local computer
Handle02= acsc_OpenComxxx(…)
acsc_SetServrExtLogin(IP1, Port#,..);   // Set Server1
Handle11=acsc_OpenComxxx(…);     // Open all channels with controllers
                                 // connected to Server1
Handle12= acsc_OpenComxxx(…)
acsc_SetServrExtLogin(IP2, Port#,..);   // Set Server2
Handle21= acsc_OpenComxxx(…);    // Open all channels with controllers
                                 // connected to Server2
Handle22= acsc_OpenComxxx(…)
```

Example

```
acsc_SetServerExtLogin("10.0.0.13",7777,"Mickey","viva11","ACS-Motion")
Handle=acsc_OpenCommPCI(-1)//Will attempt to find active UMD running on
host
                        //"10.0.0.13" and listenning on Port 7777
                        //on the host attempting to login as "Mickey"
                        //and open communication with PCI card at that host
```

4.1.9 acsc_CloseComm

Description

The function closes communication via the specified communication channel.

Syntax

int acsc_CloseComm(HANDLE Handle)

Arguments

Handle	Communication handle

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function closes the communication channel and releases all system resources related to the channel. If the function closes communication with the Simulator, the function also terminates the Simulator.

Each **acsc_OpenComm*** ** call in the application must have the corresponding **acsc_CloseComm** call in order to return the resources to the system.

Example

```
if (!acsc_CloseComm(Handle))
{
        printf("error closing communication: %d\n", acsc_GetLastError());
}
```

4.1.10 acsc_Transaction

Description

The function executes one transaction with the controller, i.e. it sends a command and receives a controller response.

Syntax

int acsc_Transaction (HANDLE Handle, char* OutBuf, int OutCount, char* InBuf,
int InCount, int* Received, ACSC_WAITBLOCK* Wait)

Any ASCII command being sent to the controller must end with the '\r' (13) character, otherwise it will not be recognized as valid.

Arguments

Handle	Communication handle
OutBuf	Output buffer that contains the command to be sent
OutCount	Number of characters in the command
InBuf	Input buffer that receives controller response
InCount	Size of the input buffer
Received	Number of characters that were actually received

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The full operation of transaction includes the following steps:

1. Send **OutCount** characters from **OutBuf** to the controller**.**
2. Waits until the controller response is received or the timeout occurs. In the case of timeout, set **Received** to zero, store error value and return.
3. Store the controller response in **InBuf**. If the controller response is longer than **InCount**, store only **InCount** characters.
4. Writes to **Received** the exact number of the characters stored in **InBuf**.
5. Analyzes the controller response, and set the error value if the response indicates an error.

If the **Wait** argument is NULL, the call is waiting and the function does not return until the full operation is finished.

If the **Wait** argument points to a valid ACSC_WAITBLOCK structure, the call is non-waiting and the function returns immediately after the first step. An internal library thread executes the rest of the operation. The calling thread can validate if the operation finished and can retrieve the operation result using the acsc_WaitForAsyncCall function.

Example

```
char* cmd = "?SN\r";                    // get controller serial number
char buf[100];
int Received, ret;
ACSC_WAITBLOCK wait;
// example of the waiting call of acsc_Transaction
if (!acsc_Transaction( Handle,  // communication handle
        cmd,                    // pointer to the buffer that contains
                                // executedcontroller's command
        strlen(cmd),            // size of this buffer
```

```
        buf,                    // input buffer that receives controller
                                // response
        100,                    // size of this buffer
        &Received,              // number of characters that were actually
                                // received
        NULL                    // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
// example of non-wainig call of acsc_Transaction
if (acsc_Transaction(Handle, cmd, strlen(cmd), buf, 100, &Received,
&wait))
{
        // something doing here
        ….
        // waiting for the controller’s response 5 sec
if (acsc_WaitForAsyncCall(      Handle, buf, &Received, &wait, 5000))
{
        buf[Received] = ‘\0’;
        printf(“Controller serial number: %s\n”, buf);
}
```

4.1.11 acsc_Command

Description

The function sends a command to the controller and analyzes the controller response.

Syntax

int acsc_Command (HANDLE Handle, char* OutBuf, int OutCount,
ACSC_WAITBLOCK* Wait)

Any ASCII command being sent to the controller must end with '\r' (13) character, otherwise it will not be recognized as valid.

Arguments

Handle	Communication handle
OutBuf	Output buffer that contains the request to be sent
OutCount	Number of characters in the request
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received.

	If **Wait** points to a valid ACSC_WAITBLOCK structure, the function acsc_WaitForAsyncCall returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function is similar to acsc_Transaction except that the controller response is not transferred to the calling thread. The function is used mainly for the commands that the controller responds to with a prompt. In this case, the exact characters that constitute the prompt are irrelevant for the calling thread. The function provides analysis of the prompt, and if the operation fails, the calling thread can obtain the error code.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_Command
char* cmd = "enable X\r";
if (!acsc_Command(( Handle,     // communication handle
        cmd,                    // pointer to the buffer that contains
                                // executed controller's command
        strlen(cmd),            // size of this buffer
        NULL                    // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.1.12 acsc_WaitForAsyncCall

Description

The function waits for completion of asynchronous call and retrieves a data.

Syntax

int acsc_WaitForAsyncCall(HANDLE Handle, void* Buf, int* Received,
ACSC_WAITBLOCK* Wait, int Timeout)

Arguments

Handle	Communication handle.
Buf	Pointer to the buffer that receives controller response. This parameter must be the same pointer that was specified for asynchronous call of SPiiPlus C function. If the SPiiPlus C function does not accept a buffer as a parameter, Buf has to be NULL pointer.
Received	Number of characters that were actually received.
Wait	Pointer to the same ACSC_WAITBLOCK structure that was specified for asynchronous call of SPiiPlus C function.
Timeout	Maximum waiting time in milliseconds. If **Timeout** is INFINITE, the function's timeout interval never elapses.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero. The **Ret** field of **Wait** contains the error code that the non-waiting call caused. If **Wait.Ret** is zero, the call succeeded: no errors occurred.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function waits for completion of asynchronous call, corresponds to the **Wait** parameter, and retrieves controller response to the buffer pointed by **Buf**. The **Wait** and **Buf** must be the same pointers passed to SPiiPlus C function when asynchronous call was initiated.

If the call of SPiiPlus C function was successful, the function retrieves controller response to the buffer **Buf**. The **Received** parameter will contain the number of actually received characters.

If the call of SPiiPlus C function does not return a response (for example: acsc_Enable, acsc_Jog, etc.) **Buf** has to be NULL.

If the call of SPiiPlus C function returned the error, the function retrieves this error code in the **Ret** member of the **Wait** parameter.

If the SPiiPlus C function has not been completed in Timeout milliseconds, the function aborts specified asynchronous call and returns ACSC_TIMEOUT error.

If the call of SPiiPlus C function has been aborted by the acsc_CancelOperationfunction, the function returns **ACSC_OPERATIONABORTED** error.

Example

```
char* cmd = "?VR\r";    // get firmware version
char buf[101];
int Received;
ACSC_WAITBLOCK wait;
if (!acsc_Transaction(Handle, cmd, strlen(cmd), buf, 100, &Received,
        &wait))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
else
{
                                // call is pending
if (acsc_WaitForAsyncCall(Handle,// communication handle
        buf,                    // pointer to the same buffer, that
                                // was specified for acsc_Transaction
        &Received,              // received bytes
        &wait,                  // pointer to the same structure, that was
                                // specified for acsc_Transaction
        500                     // 500 ms
        ))
}
{
        buf[Received] = '\0';
        printf("Firmware version: %s\n", buf);
}
else
{
        acsc_GetErrorString(Handle, wait.Ret, buf, 100, &Received);
        buf[Received] = '\0';
        printf("error: %s\n", buf);
}
}
```

4.1.13 acsc_CancelOperation

Description

The function cancels any asynchronous (non-waiting) call or all operations with the specified communication handle.

Syntax

int acsc_CancelOperation(HANDLE Handle, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle
Wait	Pointer to the ACSC_WAITBLOCK structure that was passed to the function that initiated the asynchronous (non-waiting) call.

Return Value

If the function succeeds, the return value is non-zero. The corresponding asynchronous call was successfully canceled.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

If **Wait** points to a valid ACSC_WAITBLOCK structure, the function cancels the corresponding call with the error ACSC_OPERATIONABORTED. If the corresponding call was not found the error ACSC_ CANCELOPERATIONERROR will be returned by acsc_GetLastErrorfunction.

If **Wait** is NULL, the function cancels all of the waiting and non-waiting calls for the specified communication handle.

Example

```
// cancels all of the waiting and non-waiting calls
if (!acsc_CancelOperation(Handle, NULL))
{
        printf("Cancel operation error: %d\n", acsc_GetLastError());
}
```

4.1.14 acsc_GetEthernetCardsExt

Description

The function detects available controllers through a standard Ethernet connection and retrieves the IP address, serial number, part number, and firmware version, which are stored in an **ACSC_ CONTROLLER_INFO** struct. By default, the function searches the local network segment. The default setting can be changed, as described below.

Syntax

int acsc_GetEthernetCardsExt(ACSC_CONTROLLER_INFO * ControllerInfo, unsigned long ControllerInfoSize, int Max, int*Ncontrollers, unsigned long BroadcastAddress)

Arguments

ControllerInfo	ACSC_CONTROLLER_INFO array. Fills an array of **ACSC_CONTROLLER_INFO** structs containing: > IP address > Serial number > Part number > Firmware version
ControllerInfoSize	Size of **ACSC_CONTROLLER_INFO** struct
Max	Size of ControllerInfo array

NControllers	The number of controllers actually detected
BroadcastAddress	IP address for broadcasting. Normally should be **ACSC_NONE**. See comments below for more details.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

Function Execution

The function executes as follows:

1. Broadcasts a message to the network
2. Collects all received replies
3. Filters out all replies sent by nodes other than the SPiiPlus controller
4. The function then stores controller's IP addresses in the IP address field in the ControllerInfo array and assigns the number of detected controllers to Ncontrollers.
5. Broadcasts a private messages to each ip address field in ControllerInfo array. The messages ask for the controller serial number, part number and firmware version, then the function stores them in the relevant field and returns.

If the size of the **IPaddresses** array appears too small for all detected controllers (**Ncontrollers** > **Max**), only **Max** addresses are stored in the array.

If the size of the **ACSC_CONTROLLER_INFO** struct is smaller then the offset of the struct fields, return **ACSC_INVALIDPARAMETERS**.

In order to convert the **in_addr** structure to a dot-delineated string, use the **inet_ntoa()** Microsoft Windows function.

Broadcasting Considerations

If **BroadcastAddress** needs to be specified as other than **ACSC_NONE**, use the **inet_addr** Microsoft Windows function to convert the desired broadcast IP address from a dot-delineated string to an integer parameter.

The function uses the following broadcasting message:

```
"ACS-TECH80\r"
```

The target port for broadcasting is 700. A node that doesn't support port 700 simply does not see the broadcast message. If a node other than the controller supports port 700, the node receives the message. However, since the message format is meaningful only for SPiiPlus controllers, any other node that receives the message either ignores it or responds with an error message that is filtered out by the function.

Normally, the user specifies **ACSC_NONE** in the **BroadcastAddress** parameter. In this case, the function uses broadcast address 255.255.255.255. The address causes broadcasting in a local segment of the network. If the host computer is connected to several local segments (multi-home node), the broadcast message is sent to all connected segments.

The user may wish to specify explicit broadcast addresses in the following cases:

- To reduce the broadcast area. In a multi-home node, the specific address can restrict broadcasting to one of the connected segments.
- To extend the broadcast area. If a wide area network (WAN) is supported, a proper broadcasting address can broadcast to the entire WAN, or part of the WAN.

For information about how to use broadcast addresses, refer to the network documentation.

The function uses the following message to get controller serial number:

```
"?SN\r"
```

The function uses the following message to get controller part number:

```
"?PN\r"
```

The function uses the following message to get controller firmware version:

```
"?VR\r"
```

If the **inet_ntoa** function does not work properly, try to add "Ws2_32.lib" in Linker>Input> Additional Dependencies. Alternatively you can add “#pragma comment(lib, "Ws2_32.lib")” to one of the source files in your project.

Example

```
ACSC_CONTROLLER_INFO ControllerInfo[100];
int Ncontrollers;
char *DotStr;
char Version[100];
int I;
if (!acsc_GetEthernetCardsExt(ControllerInfo,100, &Ncontrollers,ACSC_
NONE, sizeof(ACSC_CONTROLLER_INFO)))
{
printf("Error %d\n", acsc_GetLastError());
}
for(I=0;I<Ncontrollers;I++)
{
DotStr=inet_ntoa(ControllerInfo [I].IpAddress);
printf("Controller IP address: %s\n", DotStr);
printf("Controller serial number: %s\n", ControllerInfo [I].
SrialNumber);
printf("Controller part number: %s\n", ControllerInfo [I].PartNumber);
printf("Controller firmware version: %s\n", ControllerInfo [I].Version);
}
```

4.1.15 acsc_GetConnectionsList

Description

The function retrieves all currently opened connections on the active server and their details.

Syntax

int acsc_GetConnectionsList(ACSC_CONNECTION_DESC* ConnectionsList,
int MaxNumConnections, int* NumConnections)

Arguments

ConnectionsList	Pointer to array of connections. Each connection is defined by the ACSC_CONNECTION_DESC structure. The user application is responsible for allocating memory for this array.
MaxNumConnections	Number of ConnectionsList array elements.
NumConnections	Actual number of retrieved connections.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

If the size of the **ConnectionsList** array appears too small for all detected connections (Connections > **MaxNumConnections**), only **MaxNumConnections** connections are stored in the array.

This function can be used to check if there are some unused connections that remain from an application that did not close the communication channel or were not gracefully closed (terminated or killed).

Each connection from returned list can be terminated only by the acsc_TerminateConnection function.

Using the acsc_SetServerExtLogin function makes a previously returned connections list irrelevant because it changes the active server.

Example

```
int NConnections;
ACSC_CONNECTION_DESC Connections[100];
int I;
char app_name[]="needed_app.exe";
if (!acsc_GetConnectionsList(Connections, 100, &NConnections))
{
printf("Error %d\n", acsc_GetLastError());
}
for(I=0;I<NConnections;I++)
{
if ((strcmp(Connections[I].Application, app_name)==0) &&
        (OpenProcess(0,0,Connections[I].ProcessId) == NULL))
                                // Check if process is
                                //still running by OpenProcess() function,
                                //only if it was executed on local PC.
{
if (!acsc_TerminateConnection(&(Connections[i])))
{
printf("Error closing communication of %s application: %d\n",
Connections[I].Application, acsc_GetLastError());
}
else
{
printf("Communication of %s application is successfully closed!\n",
Connections[I].Application);
}
        }
}
```

4.1.16 acsc_GetConnectionInfo

Description

The function is used to retrieve the details of opened communication channel.

Syntax

int acsc_GetConnectionInfo(HANDLE Handle,
ACSC_CONNECTION_INFO* ConnectionInfo)

Arguments

Handle	Communication handle.
ConnectionInfo	Details of referenced communication channel.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
ACSC_CONNECTION_INFO ConnectionInfo;
if (!acsc_GetConnectionInfo(Handle, &ConnectionInfo))
{
        printf("acsc_GetConnectionInfo(): Error Occurred - %d\n",
        acsc_GetLastError());
        return;
}
```

4.1.17 acsc_TerminateConnection

Description

The function terminates a given communication channel (connection) of the active server.

Syntax

int acsc_TerminateConnection(ACSC_CONNECTION_DESC* Connection)

Arguments

Connection	Pointer to array of connections. Each connection is defined by the ACSC_CONNECTION_DESC structure. The acsc_GetConnectionsList function is responsible for initializing this structure.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

This function can be used to terminate an unused connection that remains from an application that did not close the communication channel or was not gracefully closed (terminated or killed). The **Connection** parameter should be passed as it was retrieved by acsc_GetConnectionsList function.

Using the acsc_SetServerExtLogin makes a previously returned connections list irrelevant because it changes the active server.

Example

```
int NConnections;
ACSC_CONNECTION_DESC Connections[100];
int I;
char app_name[]="needed_app.exe";
acsc_SetServerExtLogin("10.0.0.13",7777,"Greg","MyPassword","ACS-
```

```
Motion”);
if (!acsc_GetConnectionsList(Connections, 100, &NConnections))
{
printf("Error %d\n", acsc_GetLastError());
}
for(I=0;I<NConnections;I++)
{
if ((strcmp(Connections[I].Application, app_name)==0) &&
               (OpenProcess(Connections[I].ProcessId) == NULL))
                                  // Check if process is still
                                  //running by OpenProcess() function, only
                                  //if it was executed on local PC.
{
if (!acsc_TerminateConnection(&(Connections[i])))
{
printf("Error closing communication of %s application: %d\n",
Connections[I].Application, acsc_GetLastError());
}
else
{
printf("Communication of %s application is successfully closed!\n",
Connections[I].Application);
}
        }
}
```

4.2 Service Communication Functions

The Service Communication functions are:

Table 5-2. Service Communication Functions

Function	Description
acsc_GetCommOptions	Retrieves the communication options.
acsc_GetDefaultTimeout	Retrieves default communication time-out.
acsc_GetErrorString	Retrieves the explanation of an error code.
acsc_GetLastError	Retrieves the last error code.
acsc_GetLibraryVersion	Retrieves the SPiiPlus C Library version number.
acsc_GetTimeout	Retrieves communication time-out.
acsc_SetIterations	Sets the number of iterations of one transaction.
acsc_SetCommOptions	Sets the communication options.
acsc_SetTimeout	Sets communication time-out.

Function	Description
acsc_SetQueueOverflowTimeout	Sets the Queue Overflow Time-out.
acsc_GetQueueOverflowTimeout	Retrieves the Queue Overflow Time-out.

4.2.1 acsc_GetCommOptions

Description

The function retrieves the communication options.

Syntax

int acsc_GetCommOptions(HANDLE Handle, unsigned int* Options)

Arguments

Handle	Communication handle
Options	Current communication options Bit-mapped parameter that can include the following flag: ACSC_COMM_USE_CHECKSUM. The communication mode when each command sends to the controller with checksum and the controller responds with checksum.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the current communication options. To set the communication option call acsc_SetCommOptions.

Example

```
//example of using acsc_GetCommOptions
unsigned int Options;
if (!acsc_GetCommOptions(Handle, &Options))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.2.2 acsc_GetDefaultTimeout

Description

The function retrieves default communication timeout.

Syntax

int acsc_GetDefaultTimeout(HANDLE Handle)

Arguments

Handle	Communication handle

Return Value

If the function succeeds, the return value is the default time-out value in milliseconds.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The value of the default time-out depends on the type of established communication channel. Time-outalsodepends on the baud rate value for serial communication.

Example

```
int DefTimeout = acsc_GetDefaultTimeout(Handle);
if (DefTimeout == 0)
{
        printf("default timeout receipt error: %d\n", acsc_GetLastError());
}
```

4.2.3 acsc_GetErrorString

Description

The function retrieves the explanation of an error code.

Syntax

int acsc_GetErrorString(HANDLE Handle, int ErrorCode, char* ErrorStr, int Count,
int* Received)

Arguments

Handle	Communication handle
ErrorCode	Error code.
ErrorStr	Pointer to the buffer that receives the text explanation of **ErrorCode**.
Count	Size of the buffer pointed by **ErrorStr**
Received	Number of characters that were actually received

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the string that contains the text explanation of the error code returned by the acsc_GetLastError, acsc_GetMotorError, and acsc_GetProgramError functions.

The function will not copy more than **Count** characters to the **ErrorStr** buffer. If the buffer is too small, the error explanation can be truncated.

For the SPiiPlus C Library error codes, the function returns immediately with the text explanation. For the controller's error codes, the function refers to the controller to get the text explanation.

Example

```
char ErrorStr[256];
int ErrorCode, Received;
ErrorCode = acsc_GetLastError();
if (acsc_GetErrorString(Handle, // communication handle
        ErrorCode,              // error code
        ErrorStr,               // buffer for the error explanation
        255,                    // available buffer length
        &Received               // number of actually received bytes
        )
{
ErrorStr[Received] = '\0';
printf("function returned error: %d (%s)\n", ErrorCode, ErrorStr);
}
```

4.2.4 acsc_GetLastError

Description

The function returns the calling thread's last-error code value. The last-error code is maintained on a per-thread basis. Multiple threads do not overwrite each other's last-error code.

Syntax

int acsc_GetLastError()

Arguments

This function has no arguments.

Return Value

The return value is the calling thread's last-error code value.

Comments

It is necessary to call **acsc_GetLastError** immediately when some functions return zero value. This is because all of the functions rewrite the error code value when they are called.

For a complete List of Error Codes, see Error Codes.

Example

```
printf("function returned error: %d\n", acsc_GetLastError());
```

4.2.5 acsc_GetLibraryVersion

Description

The function retrieves the SPiiPlus C Library version number.

Syntax

unsigned int acsc_GetLibraryVersion()

Arguments

This function has no arguments.

Return Value

The return value is the 32-bit unsigned integer value, which specifies the binary version number.

Comments

The SPiiPlus C Library version consists of four (or fewer) numbers separated by points: #.#.#.#. The binary version number is represented by 32-bit unsigned integer value. Each byte of this value specifies one number in the following order: high byte of high word – first number, low byte of high word – second number, high byte of low word – third number and low byte of low word – forth number. For example version "2.10" has the following binary representation (hexadecimal format): 0x020A0000.

First two numbers in the string form are obligatory. Any release version of the library consists of two numbers. The third and fourth numbers specify an alpha or beta version, special or private build, or other version.

Example

```
unsigned int Ver = acsc_GetLibraryVersion();
printf("SPiiPlus C Library version is %d.%d.%d.%d\n", HIBYTE(HIWORD
(Ver)),
        LOBYTE(HIWORD(Ver)), HIBYTE(LOWORD(Ver)), LOBYTE(LOWORD(Ver)));
```

4.2.6 acsc_GetTimeout

Description

The function retrieves communication timeout.

Syntax

int acsc_GetTimeout(HANDLE Handle)

Arguments

Handle	Communication handle

Return Value

If the function succeeds, the return value is the current timeout value in milliseconds.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Example

```
int Timeout = acsc_GetTimeout(Handle);
if (Timeout == 0)
{
        printf("timeout receipt error: %d\n", acsc_GetLastError());
}
```

4.2.7 acsc_SetIterations

Description

The function sets the number of iterations in one transaction.

Syntax

int acsc_SetIterations(HANDLE Handle, int Iterations)

Arguments

Handle	Communication handle
Iterations	Number of iterations

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

If, after the transmission of command to the controller, the controller response is not received during the predefined time, the library repeats the transmission of command. The number of those iterations is defined by the **Iterations** parameter for each communication channel independently.

Most of the SPiiPlus C functions perform communication with the controller by transactions (i.e., they send commands and wait for responses) that are based on the acsc_Transaction function. Therefore,the changing of number of iterations can have an influence on the behavior of the user application.

The default the number of iterations for all communication channels is 2.

Example

```
if (!acsc_SetIterations(Handle, 2))
{
```

```
        printf("number of iterations setting error: %d\n",
                        acsc_GetLastError());
}
```

4.2.8 acsc_SetCommOptions

Description

The function sets the communication options.

Syntax

int acsc_SetCommOptions(HANDLE Handle, unsigned int Options)

Arguments

Handle	Communication handle
Options	Communication options to be set Bit-mapped parameter that can include one of the following flags: ACSC_COMM_USE_CHECKSUM: the communication mode used when each command is sent to the controller with checksum and the controller also responds with checksum. ACSC_COMM_AUTORECOVER_HW_ERROR: When a hardware error is detected in the communication channel and this bit is set, the library automatically repeats the transaction, without counting iterations.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function sets the communication options. To get current communication option, call acsc_GetCommOptions.

To add some communication options to the current configuration, modify an **Option** parameter that has been filled in by a call to acsc_GetCommOptions. This ensures that the other communication options will have same values.

Example

```
//example of setting the mode with checksum
unsigned int Options;
acsc_GetCommOptions(Handle, &Options);
Options = Options | ACSC_COMM_USE_CHECKSUM;
acsc_SetCommOptions(Handle, Options);
```

4.2.9 acsc_SetTimeout

Description

The function sets the communication timeout.

Syntax

int acsc_SetTimeout(HANDLE Handle, int Timeout)

Arguments

Handle	Communication handle
Timeout	Maximum waiting time in milliseconds.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function sets the communication timeout.

All of the subsequent waiting calls of the functions will wait for the controller response **Timeout** in milliseconds. If the controller does not respond to the sent command during this time, SPiiPlus C functions return with zero value. In this case, the call of acsc_GetLastError will return the **ACSC_TIMEOUT** error.

Example

```
if (!acsc_SetTimeout(Handle, 5000))
{
        printf("timeout set error: %d\n", acsc_GetLastError());
}
```

4.2.10 acsc_SetQueueOverflowTimeout

Description

The function sets the Queue Overflow Timeout.

Syntax

int acsc_SetQueueOverflowTimeout (HANDLE Handle, int Delay)

Arguments

Handle	Communication handle.
Timeout	Timeout value in milliseconds

Return Value

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function sets Queue Overflow Timeout value in milliseconds. See also Non-Waiting Calls.

Example

```
if (!acsc_ SetQueueOverflowTimeout(Handle,100))
{
        printf("Queue Overflow Timeout setting error: %d\n",
        acsc_GetLastError());
}
```

4.2.11 acsc_GetQueueOverflowTimeout

Description

The function retrieves the Queue Overflow Timeout.

Syntax

int acsc_GetQueueOverflowTimeout(HANDLE Handle)

Arguments

Handle	Communication handle.

Return Value

If the function succeeds, the return value is the current Queue Overflow Timeout value in milliseconds.

If the function fails, the return value is ACSC_NONE.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

See Non-Waiting Calls for an explanation about Queue Overflow Timeout.

Example

```
int QueueTimeout = acsc_GetQueueOverflowTimeout(Handle);
if (QueueTimeout == ACSC_NONE)
{
        printf("Queue Overflow Timeout receipt error: %d\n",
               acsc_GetLastError());
}
```

4.3 ACSPL+ Program Management Functions

The Program Management functions are:

Table 5-3. ACSPL+ Program Management Functions

Function	Description
acsc_AppendBuffer	Appends one or more ACSPL+ lines to the program in the specified program buffer.
acsc_ClearBuffer	Deletes the specified ACSPL+ program lines in the specified program buffer.
acsc_CompileBuffer	Compiles ACSPL+ program in the specified program buffer(s).
acsc_LoadBuffer	Clears the specified program buffer and then loads ACSPL+ program to this buffer.
acsc_ LoadBufferIgnoreServiceLines	Clears the specified program buffer and then loads ACSPL+ program to this buffer.
acsc_LoadBuffersFromFile	Opens a file that contains one or more ACSPL+ programs allocated to several buffers and download the programs to the corresponding buffers.
acsc_RunBuffer	Starts up ACSPL+ program in the specified program buffer.
acsc_StopBuffer	Stops ACSPL+ program in the specified program buffer (s).
acsc_SuspendBuffer	Suspends ACSPL+ program in the specified program buffer(s).
acsc_UploadBuffer	Uploads ACSPL+ program from the specified program buffer.

4.3.1 acsc_AppendBuffer

Description

The function appends one or more ACSPL+ lines to the program in the specified buffer.

Syntax

int acsc_AppendBuffer(HANDLE Handle, int Buffer, char* Program, int Count,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle
Buffer	Buffer number, from 0 to 63 (depending on controller).
Program	Pointer to the buffer contained ACSPL+ program(s).
Count	Number of characters in the buffer pointed by **Program**
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function appends one or more ACSPL+ lines to the program in the specified buffer. If the buffer already contains any program, the new text is appended to the end of the existing program.

No compilation or syntax check is provided during downloading. In fact, any text, not only a correct program, can be inserted into a buffer. In order to compile the program and check its accuracy, the compile command must be executed after downloading.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_AppendBuffer
char buf[256];
strcpy(buf, "!This is a test ACSPL+ program\n" );
strcat(buf, "enable 0\n" );
strcat(buf, "ptp 0, 1000\n" );
strcat(buf, "stop\n" );
if (!acsc_AppendBuffer(Handle,  // communication handle
        0,              // ACSPL+ program buffer number
        buf,            // buffer contained ACSPL+ program(s)
```

```
        strlen(buf),    // size of this buffer
        NULL            // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.3.2 acsc_ClearBuffer

Description

The function deletes the specified ACSPL+ program lines in the specified program buffer.

Syntax

int acsc_ClearBuffer(HANDLE Handle, int Buffer, int FromLine, int ToLine,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Buffer	Buffer number, from 0 to 63 (depending on controller).
FromLine, ToLine	These parameters specify a range of lines to be deleted. **FromLine** starts from 1. If **ToLine** is larger then the total number of lines in the specified program buffer, the range includes the last program line.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function deletes the specified ACSPL+ program lines in the specified program buffer.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_ClearBuffer
if (!acsc_ClearBuffer(Handle,   // communication handle
        0,                      // ACSPL+ program buffer number
        1, 100000,              // delete all lines from line 1 to line 100000
        NULL                    // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.3.3 acsc_CompileBuffer

Description

The function compiles ACSPL+ program in the specified program buffer(s).

Syntax

int acsc_CompileBuffer(HANDLE Handle, int Buffer, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Buffer	Buffer number, from 0 to 63 (depending on controller). Use ACSC_NONE instead of the buffer number, to compile all programs in all buffers.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

The function returns non-zero if it succeeded to perform the compile operation on the buffer, such that the communication channel is OK, the specified buffer is not running and compile operation was performed. However, it does not mean that compilation succeeded. If the return value is zero, compile operation could not be performed by some reason.

Extended error information can be obtained by calling **acsc_GetLastError**.

In order to get information about compilation results, use acsc_ReadInteger to read **PERR** [X], which contains the last error that occurred in buffer X. If **PERR** [X] is zero, the buffer was compiled successfully.

Otherwise, **PERR** [X] tells you about the error that occurred during the compilation.

Comments

The function compiles ACSPL+ program in the specified program buffer or all programs in all buffers if the parameter **Buffer** is ACSC_NONE.

If attempting to compile the D-Buffer, all other buffers will be stopped and put in a non-compiled state.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the program was compiled successfully.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_CompileBuffer
if (!acsc_CompileBuffer(Handle, // communication handle
        0,              // ACSPL+ program buffer number
        NULL            // waiting call
        ))
{
        printf("compilation error: %d\n", acsc_GetLastError());
}
```

4.3.4 acsc_LoadBuffer

Description

The function clears the specified program buffer and then loads ACSPL+ program to this buffer.

Syntax

int acsc_LoadBuffer(HANDLE Handle, int Buffer, char* Program, int Count,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle
Buffer	Buffer number, from 0 to 63 (depending on controller).

Program	Pointer to the buffer contained ACSPL+ program(s).
Count	Number of characters in the buffer pointed by **Program**
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function clears the specified program buffer and then loads ACSPL+ program to this buffer.

No compilation or syntax check is provided during downloading. Any text, not only a correct program, can be inserted into a buffer. In order to compile the program and check its accuracy, the compile command must be executed after downloading.

If Wait points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the Wait item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_LoadBuffer
char buf[256];
strcpy(buf, "!This is a test ACSPL+ program\n" );
strcat(buf, "enable 0\n" );
strcat(buf, "ptp 0, 1000\n" );
strcat(buf, "stop\n" );
if (!acsc_LoadBuffer(Handle, // communication handle
        0,               // ACSPL+ program buffer number
        buf,             // pointer to the buffer containing
                         // ACSPL+ program(s).
        strlen(buf),     // size of this buffer
        NULL             // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.3.5 acsc_LoadBufferIgnoreServiceLines

Description

The function clears the specified program buffer and then loads ACSPL+ to this buffer.

All lines that start with # are ignored.

This function is obsolete and is maintained only for backwards compatibility. When loading files saved from the MultiDebugger and MMI use the acsc_LoadBuffersFromFile function.

Syntax

int acsc_LoadBufferIgnoreServiceLines(HANDLE Handle, int Buffer, char* Program,
int Count, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle
Buffer	Buffer number, from 0 to 63 (depending on controller).
Program	Pointer to the buffer contained ACSPL+ program(s).
Count	Number of characters in the buffer pointed by **Program**

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function clears the specified program buffer and then loads ACSPL+ program to this buffer.

No compilation or syntax check is provided during downloading. Any text, not only a correct program, can be inserted into a buffer. In order to compile the program and check its accuracy, the compile command must be executed after downloading.

You can use this function in order to load program from a file created by SPiiPlus MMI **Program Manager** if it contains a program in only one buffer. If there are programs in more than one buffer, they will all be appended because the separators are ignored.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_LoadBuffer
char buf[256];
strcpy(buf, "!This is a test ACSPL+ program\n" );
strcat(buf, "#Version 3.0 \n" ); //Ignored - won't be loaded
strcat(buf, "enable 0\n" );
strcat(buf, "ptp 0, 1000\n" );
strcat(buf, "stop\n" );
if (!acsc_LoadBufferIgnoreServiceLines(Handle,  // communication handle
        0,      // ACSPL+ program buffer number
        buf,    // pointer to the buffer containing ACSPL+ program(s)
                strlen(buf),    // size of this buffer
                NULL            // waiting call
                ))
```

```
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.3.6 acsc_LoadBuffersFromFile

Description

The function opens a file that contains one or more ACSPL+ programs allocated to several buffers and download the programs to the corresponding buffers.

Syntax

int acsc_LoadBuffersFromFile(HANDLE Handle, char *Filename,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle
Filename	Path of the file
Wait	**Wait** has to be ACSC_SYNCHRONOUS, since only synchronous calls are supported for this function.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function analyzes the file, determines which program buffers should be loaded, clears them and then loads ACSPL+ programs to those buffers.

SPiiPlus software tools save ACSPL+ programs in the following format:

```
# Header: Date, Firmware version etc.
#Buf1 (buffer number)
ACSPL+ program of Buf1
#Buf2 (buffer number)
ACSPL+ program of Buf2
#Buf3 (buffer number)
ACSPL+ program of Buf3 etc.
```

The number of buffers in a file may change from 0 to 63 (depending on controller), without any default order.

No compilation or syntax check is provided during downloading. Any text, not only a correct program, can be inserted into a buffer. In order to compile the program and check its accuracy, the compile command must be executed after downloading.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_LoadBuffersFromFile
if (!acsc_LoadBuffersFromFile(Handle,   // communication handle
        "C:\\MMI\\Program.prg",                         //full path
        NULL            // waiting call
        ))
{
        printf("acsc_LoadBuffersFromFile: %d\n", acsc_GetLastError());
}
```

4.3.7 acsc_RunBuffer

Description

The function starts up ACSPL+ program in the specified buffer.

Syntax

int acsc_RunBuffer(HANDLE Handle, int Buffer, char* Label,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Buffer	Buffer number, from 0 to 63 (depending on controller).
Label	Label in the program that the execution starts from. If NULL is specified instead of a pointer, the execution starts from the first line.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function starts up ACSPL+ program in the specified buffer. The execution starts from the specified label, or from the first line if the label is not specified.

If the program was not compiled before, the function first compiles the program and then starts it. If an error was encountered during compilation, the program does not start.

If the program was suspended by the acsc_SuspendBuffer function, the function resumes the program execution from the point where the program was suspended.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the program in the specified buffer was started successfully. The function does not wait for the program end. To wait for the program end, use the acsc_WaitProgramEnd function.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_RunBuffer
if (!acsc_RunBuffer(Handle,     // communication handle
        0,              // ACSPL+ program buffer number
        NULL,           // from the beginning of this buffer
        NULL            // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.3.8 acsc_StopBuffer

Description

The function stops the execution of ACSPL+ program in the specified buffer(s).

Syntax

int acsc_StopBuffer(HANDLE Handle, int Buffer, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Buffer	Buffer number, from 0 to 63 (depending on controller). Use ACSC_NONE instead of the buffer number, to stop the execution of all programs in all buffers.

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function stops ACSPL+ program execution in the specified buffer or in all buffers if the parameter **Buffer** is ACSC_NONE.

The function has no effect if the program in the specified buffer is not running.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
/ example of the waiting call of acsc_StopBuffer
if (!acsc_StopBuffer(Handle,    // communication handle
        0,              // ACSPL+ program buffer number
        NULL            // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.3.9 acsc_SuspendBuffer

Description

The function suspends the execution of ACSPL+ program in the specified program buffer(s).

Syntax

int acsc_SuspendBuffer(HANDLE Handle, int Buffer, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Buffer	Buffer number, from 0 to 63 (depending on controller). Use ACSC_NONE instead of the buffer number, to suspend the execution of all programs in all buffers.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function suspends ACSPL+ program execution in the specified buffer or in all buffers if the parameter **Buffer** is ACSC_NONE. The function has no effect if the program in the specified buffer is not running.

To resume execution of the program in the specified buffer, call the acsc_RunBuffer function.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SuspendBuffer
if (!acsc_SuspendBuffer(Handle, // communication handle
        0,      // ACSPL+ program buffer number
        NULL    // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.3.10 acsc_UploadBuffer

Description

The function uploads ACSPL+ program from the specified program buffer.

Syntax

int acsc_UploadBuffer(HANDLE Handle, int Buffer, int Offset, char* Program,
int Count, int* Received, ACSC_WAITBLOCK* Wait)

Arguments

Example

```
// example of the waiting call of acsc_UploadBuffer
char buf[256];
int Received;
if (!acsc_UploadBuffer(Handle,  // communication handle
        0,      // ACSPL+ program buffer number
        0,      // from the beginning of this buffer
        buf,    // pointer to the buffer that receives uploaded text
        256,    // size of this buffer
        &Received,      // number of characters that were actually uploaded
        NULL    // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.4 Read and Write Variables Functions

The Read and Write Variables functions are:

Table 5-4. Read and Write Variables Functions

Function	Description
acsc_ReadInteger	Reads value from integer variable.
acsc_WriteInteger	Writes value to integer variable.
acsc_ReadReal	Reads value from real variable.
acsc_WriteReal	Writes value to real variable.

4.4.1 acsc_ReadInteger

Description

The function reads value(s) from integer variable.

Syntax

int acsc_ReadInteger(HANDLE Handle, int NBuf, char* Var, int From1, int To1,
int From2, int To2, int* Values, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle
NBuf	Number of program buffer for local variable or ACSC_NONE for global and standard variable.
Var	Pointer to a null-terminated character string that contains a name of the variable.
From1, To1	Index range (first dimension).
From2, To2	Index range (second dimension).
Values	Pointer to the buffer that receives requested values.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function reads specified integer variable or array.

The variable can be a standard controller variable, a user global variable, or a user local variable.

Standard and user global variables have global scope. Therefore, parameter **Nbuf** must be ACSC_NONE (-1) for these classes of variables.

User local variable exists only within a buffer. The buffer number must be specified for user local variable.

If the variable is scalar, all indexes **From1, To1, From2, To2** must be ACSC_NONE. The function reads the requested value and assigns it to the variable pointed by **Values**.

If the variable is a one-dimensional array, **From1**, **To1** must specify the index range and **From2**, **To2** must be ACSC_NONE. Array **Values** must be of size **To1-From1+1** at least. The function reads all requested values from index **From1** to index **To1** inclusively and stores them in the **Values** array.

If the variable is a two-dimensional array, **From1**, **To1** must specify the index range of the first dimension and **From2**, **To2** must specify the index range of the second dimension. Array **Values** must be of size **(To1-From1+1)x(To2-From2+1)** values at least. The function uses the **Values** array in such a way: first, the function reads **To2-From2+1** values from row **From1** and fills the **Values** array elements from 0 to **To2-From2**, then reads **To2-From2+1** values from raw **From1+1** and fills the **Values** array elements from **To2-From2+1** to **2*(To2-From2)+1**, etc.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Values** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_ReadInteger
int AnalogInputs[8];
if (!acsc_ReadInteger(Handle,   // communication handle
        ACSC_NONE,              // standard variable
        "AIN",                  // variable name
        0, 7,                   // first dimension indexes
        ACSC_NONE, ACSC_NONE,   // no second dimension
        AnalogInputs,           // pointer to the buffer
                                // that receives requested
                                // values
        NULL                    // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.4.2 acsc_WriteInteger

Description

The function writes value(s) to integer variable.

Syntax

int acsc_WriteInteger(HANDLE Handle, int NBuf, char* Var, int From1, int To1,
int From2, int To2, int* Values, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle
NBuf	Number of program buffer for local variable or ACSC_NONE for global and standard variable.
Var	Pointer to the null-terminated character string contained name of the variable.
From1, To1	Index range (first dimension).
From2, To2	Index range (second dimension).
Values	Pointer to the buffer contained values that must be written.

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function writes to a specified integer variable or array.

The variable can be a standard controller variable, user global or user local.

Standard and user global variables have global scope. Therefore, parameter **Nbuf** must be ACSC_NONE (-1) for these classes of variables.

User local variable exists only within a buffer. The buffer number must be specified for user local variable.

If the variable is scalar, all indexes **From1, To1, From2, To2** must be ACSC_NONE (-1). The function writes the value pointed by **Values** to the specified variable.

If the variable is a one-dimensional array, **From1**, **To1** must specify the index range and **From2**, **To2** must be ACSC_NONE (-1). Array **Values** must contain **To1-From1+1** values at least. The function writes the values to the specified variable from index **From1** to index **To1** inclusively.

If the variable is a two-dimensional array, **From1**, **To1** must specify the index range of the first dimension and **From2**, **To2** must specify the index range of the second dimension. Array **Values** must contain **(To1-From1+1)x(To2-From2+1)** values at least. The function uses the **Values** array as follows: first, the function retrieves the **Values** elements from 0 to **To2-From2** and writes them to row **From1** of the specified controller variable, then retrieves the **Values** elements from **To2-From2+1** to **2*(To2-From2)+1** and writes them to row **From1+1** of the specified controller variable, etc.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
/ example of the waiting call of acsc_WriteInteger
int AnalogOutputs[4] = { 10, 20, 30, 40 };
if (!acsc_WriteInteger(Handle,  // communication handle
        ACSC_NONE,              // standard variable
        "AOUT",                 // variable name
        0, 3,                   // first dimension indexes
        ACSC_NONE, ACSC_NONE,   // no second dimension
        AnalogOutputs,          // pointer to  the buffer contained values
                                // that must be written
        NULL                    // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.4.3 acsc_ReadReal

Description

The function reads value(s) from a real variable.

Syntax

int acsc_ReadReal(HANDLE Handle, int NBuf, char* Var, int From1, int To1, int From2, int To2, double* Values, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle
NBuf	Number of program buffer for local variable or ACSC_NONE for global and standard variable.
Var	Pointer to the null-terminated character string contained name of the variable.
From1, To1	Index range (first dimension).
From2, To2	Index range (second dimension).
Values	Pointer to the buffer that receives requested values.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function reads specified real variable or array.

The variable can be a standard controller variable, user global variable, or user local variable.

ACSPL+ and user global variables have global scope. Therefore, parameter **Nbuf** must be **ACSC_NONE** (-1) for these classes of variables.

User local variable exists only within a buffer. The buffer number must be specified for user local variable.

If the variable is scalar, all indexes **From1, To1, From2, To2** must be **ACSC_NONE**. The function reads the requested value and assigns it to the variable pointed by **Values**.

If the variable is a one-dimensional array, **From1**, **To1** must specify the index range and **From2**, **To2** must be ACSC_NONE. Array **Values** must be of size **To1-From1+1** at least. The function reads all requested values from index **From1** to index **To1** inclusively and stores them in the **Values** array.

If the variable is a two-dimensional array, **From1**, **To1** must specify the index range of the first dimension and **From2**, **To2** must specify the index range of the second dimension. Array **Values** must be of size **(To1-From1+1)x(To2-From2+1)** values at least. The function uses the **Values** array in such a way: first, the function reads **To2-From2+1** values from row **From1** and fills the **Values** array elements from 0 to **To2-From2**, then reads **To2-From2+1** values from raw **From1+1** and fills the **Values** array elements from **To2-From2+1** to **2*(To2-From2)+1**, etc.

If Wait points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Values** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_ReadReal
double FPositions[8];
if (!acsc_ReadReal(Handle,// communication handle
        ACSC_NONE,      // standard variable
        "FPOS",         // variable name
        0, 7,           // first dimension indexes
        ACSC_NONE, ACSC_NONE,   // no second dimension
        FPositions,             // pointer to the buffer that
                                // receives requested values
        NULL                    // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.4.4 acsc_WriteReal

Description

The function writes value(s) to the real variable.

Syntax

int acsc_WriteReal(HANDLE Handle, int NBuf, char* Var, int From1, int To1,
int From2, int To2, double* Values, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle
NBuf	Number of program buffer for local variable or ACSC_NONE for global and standard variable.
Var	Pointer to the null-terminated character string contained name of the variable.
From1, To1	Index range (first dimension).
From2, To2	Index range (second dimension).
Values	Pointer to the buffer contained values that must be written.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function writes to the specified real variable or array.

The variable can be a standard controller variable, user global or user local.

Standard and user global variables have global scope. Therefore, parameter **Nbuf** must be ACSC_NONE (-1) for these classes of variables.

User local variable exists only within a buffer. The buffer number must be specified for user local variable.

If the variable is scalar, all indexes **From1, To1, From2, To2** must be ACSC_NONE (-1). The function writes the value pointed by **Values** to the specified variable.

If the variable is a one-dimensional array, **From1**, **To1** must specify the index range and **From2**, **To2** must be ACSC_NONE (-1). Array **Values** must contain **To1-From1+1** values at least. The function writes the values to the specified variable from index **From1** to index **To1** inclusively.

If the variable is a two-dimensional array, **From1**, **To1** must specify the index range of the first dimension and **From2**, **To2** must specify the index range of the second dimension. Array **Values** must contain **(To1-From1+1)x(To2-From2+1)** values at least. The function uses the **Values** array in such a way: first, the function retrieves the **Values** elements from 0 to **To2-From2** and writes them to row **From1** of the specified controller variable, then retrieves the **Values** elements from **To2-From2+1** to **2*(To2-From2)+1** and writes them to row **From1+1** of the specified controller variable, etc.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_WriteReal
double Velocity[8] = { 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000 };
if (!acsc_WriteReal(Handle,// communication handle
        ACSC_NONE,      // standard variable
        "VEL",          // variable name
        0, 7,           // first dimension indexes
        ACSC_NONE, ACSC_NONE,   // no second dimension
        Velocity,       // pointer  to  the buffer contained values
                        // to be written
        NULL            // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.4.5 acsc_ReadString

Description

The function retrieves String type values from ACSPL+ standard or user defined String variables or arrays.

Syntax

```
int acsc_ReadString(HANDLE Handle, int NBuf, char* Var, int From1, int
To1, int From2, int To2, char* Values, unsigned char CellSize, ACSC_
WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
NBuf	Number of program buffer for local variable or ACSC_NONE (-1) for global and standard variable.
Var	Pointer to a null-terminated character string that contains a name of the variable.
From1, To1	Index range (first dimension).
From2, To2	Index range (second dimension).
Values (out parameter)	Pointer to the buffer that receives requested values.
CellSize	The size of each cell dedicated to a single value.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

```
// Reading a value from string array.
char reply[2][16];
if (!acsc_ReadString(handle, ACSC_BUFFER_1, "myStringArray", 0, 1, -1, -1, (char*)reply, 16,
NULL))
{
   printf("acsc_ReadString(): Error Occurred - %d\n", acsc_GetLastError());
   printf("Try to run acsc_WriteString test before running this test."); }

// Reading a value from string variable.
if (!acsc_ReadString(handle, ACSC_BUFFER_0, "myStringVariable", 0, 0, -1, -1,(char*)reply,
12, NULL))
{
   printf("acsc_ReadString(): Error Occurred - %d\n", acsc_GetLastError());
   printf("Try to run acsc_WriteString test before running this test.");
}
```

4.4.6 acsc_WriteString

Description

The function writes values to ACSPL standard or user defined String variables or arrays.

Syntax

```
int _ACSCLIB_ WINAPI acsc_WriteString(HANDLE Handle, int NBuf, char* Var,
int From1, int To1,  int From2, int To2, char* Values, unsigned char
CellSize, ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
NBuf	Number of program buffer for local variable or ACSC_NONE (-1) for global and standard variable.
Var	Pointer to a null-terminated character string that contains a name of the variable.
From1, To1	Index range (first dimension).
From2, To2	Index range (second dimension).
Values	Pointer to the buffer that contains the new values.
CellSize	The size of each cell dedicated to a single value.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the **acsc_ WaitForAsyncCall** function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Example

```
char buff[5][20];
for (int i = 0; i < 5; i++) {
    sprintf_s(buff[i], "C_str_%d_boo", i);
}

unsigned long readCount = 0;
int count;
if (!acsc_WriteString(controller, 12, (char*)"myArr", 0, 3, -1, -1, buff
[0], 20, ACSC_SYNCHRONOUS)) {
    printf("acsc_ReadString failed, Error %d \n", acsc_GetLastError());
    acsc_CloseComm(controller);
    return 0;
}
```

4.5 Load/Upload Data To/From Controller Functions

The Load/Upload Data To/From Controller functions are:

Table 5-5. Load File to ACSPL+ Variables Functions

Function	Description
acsc_LoadDataToController	Writes value(s) from text file to SPiiPlus controller (variable or file).
acsc_ UploadDataFromController	Writes value(s) from the SPiiPlus controller (variable or file) to a text file.

4.5.1 acsc_LoadDataToController

Description

The function writes value(s) from text file to SPiiPlus controller (variable or file).

Syntax

int acsc_LoadDataToController(HANDLE Handle,int Dest, char* DestName, int From1,
int To1, int From2, int To2, char* SrcFileName, int SrcNumFormat,
bool bTranspose, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Dest	Number of program buffer for local variable, ACSC_NONE for global and standard variable. ACSC_FILE for loading directly to file on flash memory (only arrays can be written directly into controller files).
DestName	Pointer to the null-terminated character string that contains name of the variable or file.

From1, To1	Index range (first dimension).
From2, To2	Index range (second dimension).
SrcFileName	Filename (including path) of the source text data file.
SrcNumFormat	Format of number(s) in source file. Use: ACSC_INT_BINARY for integer numbers, or ACSC_REAL_BINARY for real numbers
bTranspose	If TRUE, then the array will be transposed before being loaded; otherwise, this parameter has no affect.
Wait	**Wait** has to be ACSC_SYNCHRONOUS, since only synchronous calls are supported for this function.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function writes to a specified variable (scalar/array) or directly to a binary file on the controller's flash memory. The variable can be an ACSPL+ variable, user global or user local.

The input file (pointed to by the **SrcFileName** argument) **must** be in ANSI format, otherwise error 168, **ACSC_INVALID_FILE_FORMAT**, is returned.

ACSPL+ and user global variables have global scope. Therefore, **Dest** must be ACSC_NONE (-1) for these classes of variables. User local variable exists only within a buffer. The buffer number must be specified for user local variable.

If **Dest** is ACSC_NONE (-1) and there is no global variable with the name specified by **DestName**, it would be defined. Arrays will be defined with dimensions (**To1+1**, **To2+1**).

If performing loading directly to a file, **From1**, **To1**, **From2** and **To2** are meaningless.

If the variable is scalar, the **From1**, **To1**, **From2**, and **To2** arguments must be **ACSC_NONE** (-1). The function writes the value from the file specified by **SrcFileName** to the variable specified by **Name**.

If the variable is a one-dimensional array, **From1**, **To1** must specify the index range and the **From2**, **To2** must be ACSC_NONE (-1). The text file, pointed to by the **SrcFileName** argument, must contain To1 to From1+1 values, at least. The function writes the values to the specified variable from the **From1** index to the **To1** index inclusively.

If the variable is a two-dimensional array, **From1**, **To1** must specify the index range of the first dimension and **From2**, **To2** must specify the index range of the second dimension. The text file, pointed to by the **SrcFileName** argument, must contain ((To1-From1+1) x (To2-From2+1)) values, at least; otherwise, an error will occur.

The function uses the text file as follows:

1. The function retrieves the To2-From2+1 values and writes them to row From1 of the specified controller variable
2. Then retrieves next To2-From2+1 values and writes them to row From1+1 of the specified controller variable, etc.

If **bTranspose** is TRUE, the function actions are inverted. It takes **To1-From1+1** values and writes them to column **From2** of the specified controller variable, then retrieves next **To1-From1+1** values and writes them to column **From2+1** of the specified controller variable, etc.

The text file is processed line-by-line; any characters except numbers, dots, commas and exponent 'e' are translated as separators between the numbers. A line that starts with no digits is considered as comment and ignored.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the Wait item until a call to the acsc_WaitForAsyncCall function has been made.

Example

```
// example of the waiting call of acsc_LoadDataToController
if (!acsc_LoadDataToController(Handle, // communication handle
     ACSC_NONE, // standard variable
     "UserArray", // variable name
      0, 99, // first dimension indexes
      -1,-1, // no second dimension
        "UserArr.TXT", // Text file name contained values
                       // that must be written
      ACSC_INT_TYPE, //decimal integers
        FALSE, //no Transpose required
NULL // waiting call
))
{
printf("transaction error: %d\n", acsc_GetLastError());
}
UserArr.TXT has this structure:
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 .....
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 .....
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 .....
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 .....
…
…
```

4.5.2 acsc_UploadDataFromController

Description

This function writes value(s) from the SPiiPlus controller (variable or file) to a text file.

Syntax

int acsc_UploadDataFromController(HANDLE Handle, int Src, char * SrcName,
int SrcNumFormat, int From1, int To1, int From2, int To2, char* DestFileName,
char* DestNumFormat, bool bTranspose, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Src	Number of the program buffer for local variable, ACSC_NONE for global and ASCPL+ variables, and ACSC_FILE for downloading directly from file on flash memory.
SrcName	Pointer to the null-terminated character string that contains name of the Variable or File.
SrcNumFormat	Format of number(s) in the controller. Use: ACSC_INT_BINARY for integer numbers, and ACSC_REAL_BINARY for real numbers
From1, To1	Index range (first dimension).
From2, To2	Index range (second dimension).
DestFileName	Filename (including full path) of the destination text file.
DestNumFormat	Pointer to the null-terminated character string that contains the formatting string that will be used for printing into file, for example, 1:%d\n. Use the string with %d for integers, and %lf for reals.
bTranspose	If TRUE, then the array is transposed before being downloaded; otherwise, this parameter has no effect.
Wait	**Wait** has to be ACSC_SYNCHRONOUS, since only synchronous calls are supported for this function.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function writes data to the specified file from a specified variable (scalar/array) or directly from a binary file in the controller's flash memory.

The variable can be a standard ASCPL+ variable, user global or user local. The ASCPL+ and user global variables have global scope. In this case, the **Src** argument has to be **ACSC_NONE** (-1) for these classes of variables. User local variable exists only within a buffer. The buffer number has to be specified for user local variable.

If the variable (or file) with the name specified by **SrcName** does not exist, an error occurs.

If loading directly from the file, **From1**, **To1**, **From2** and **To2** are meaningless.

If the variable is a scalar, all **From1**, **To1**, **From2** and **To2** values must be ACSC_NONE (-1). The function writes the value from variable specified by **SrcName** to the file specified by **DestFileName**.

If the variable is a one-dimensional array, **From1**, **To1** must specify the index range, and **From2**, **To2** must be ACSC_NONE (-1). The function writes the values from the specified variable from the **From1** value to the **To1** value, inclusively, to the file specified by **DestFileName**.

If the variable is a two-dimensional array, **From1**, **To1** must specify the index range of the first dimension, and **From2**, **To2** must specify the index range of the second dimension. The function uses the variable as follows: first, the function retrieves the To2-From2+1 values from the row specified in **From1** and writes them to the file specified by **DestFileName**. It then retrieves To2-From2+1 values from the From1+1 row and writes them, and so forth.

If **bTranspose** is TRUE, the function actions are inverted. It takes To1-From1+1 values from the row specified by **From2** and writes them to first column of the destination file. Then retrieves the next To1-From1 values from row From2+1 and writes them to the next column of the file.

The destination file's format can be determined by string specified by **DestNumFormat**. This string will be used as argument in *printf function.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the Wait item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_DownloadDataFromController
if (!acsc_UploadDataFromController(Handle, // communication handle
     ACSC_NONE, // standard variable
     "UserArray", // variable name
        ACSC_INT_BINARY, //from binary integers
        0, 99, // first dimension indexes
      -1,-1, // no second dimension
        "UserArr.TXT", // Text file name of destination file
                    // that must be written
"%d",//Format of written file
FALSE,          //no Transpose required
NULL            // waiting call
))
{
printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.6 Multiple Thread Synchronization Functions

The Multiple Thread Synchronization functions are:

Table 5-6. Multiple Thread Synchronization Functions

Function	Description
acsc_CaptureComm	Captures a communication channel.
acsc_ReleaseComm	Releases a communication channel.

4.6.1 acsc_CaptureComm

Description

The function captures a communication channel.

Syntax

int acsc_CaptureComm(HANDLE Handle)

Arguments

Handle	Communication handle

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function captures the communication handle for the calling thread and prevents access to this communication handle from other threads.

If one thread captures the communication handle and another thread calls one of SPiiPlus C Library functions using the same handle, the second thread will be delayed until the first thread executes acsc_ReleaseComm.

The function provides ability to execute a sequence of functions without risk of intervention from other threads.

Example

```
if (!acsc_CaptureComm(Handle))
{
        printf("capture communication error: %d\n", acsc_GetLastError());
```

4.6.2 acsc_ReleaseComm

Description

The function releases a communication channel.

Syntax

int acsc_ReleaseComm(HANDLE Handle)

Arguments

Handle	Communication handle

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function releases the communication handle captured by acsc_CaptureComm and allows other threads to communicate through the channel.

Example

```
if (!acsc_ReleaseComm(Handle))
{
        printf("release communication error: %d\n", acsc_GetLastError());
}
```

4.7 History Buffer Management Functions

The History Buffer Management functions are:

Table 5-7. History Buffer Management Functions

Function	Description
acsc_OpenHistoryBuffer	Opens a history buffer.
acsc_CloseHistoryBuffer	Closes a history buffer.
acsc_GetHistory	Retrieves the contents of the history buffer.

4.7.1 acsc_OpenHistoryBuffer

Description

The function opens a history buffer.

Syntax

LP_ACSC_HISTORYBUFFER acsc_OpenHistoryBuffer(HANDLE Handle, int Size)

Arguments

Handle	Communication handle
Size	Required size of the buffer in bytes

Return Value

The function returns pointer to **ACSC_HISTORYBUFFER** structure.

If the buffer cannot be allocated, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function allocates a history buffer that stores all commands sent to the controller and all responses and unsolicited messages received from the controller.

Only one history buffer can be open for each communication handle.

The buffer works as a cyclic buffer. When the amount of the stored data exceeds the buffer size, the newly stored data overwrites the earliest data in the buffer.

Example

```
LP_ACSC_HISTORYBUFFER lpHistoryBuf = acsc_OpenHistoryBuffer(
                        Handle,         // communication handle
                        10000           // size of the buffer
                        );
if (!lpHistoryBuf)
{
        printf("opening history buffer error: %d\n", acsc_GetLastError());
}
```

4.7.2 acsc_CloseHistoryBuffer

Description

The function closes the history buffer and discards all stored history.

Syntax

int acsc_CloseHistoryBuffer(HANDLE Handle)

Arguments

Handle	Communication handle

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function closes the history buffer and releases the used memory. All information stored in the buffer is discarded.

Example

```
if (!acsc_CloseHistoryBuffer(Handle))
{
        printf("closing history buffer error: %d\n", acsc_GetLastError());
}
```

4.7.3 acsc_GetHistory

Description

The function retrieves the contents of the history buffer.

Syntax

int acsc_GetHistory(HANDLE Handle, char* Buf, int Count, int* Received,
BOOL bClear)

Arguments

Handle	Communication handle
Buf	Pointer to the buffer that receives the communication history
Count	Size of the buffer in bytes
Received	Number of characters that were actually received
bClear	If TRUE, the function clears contents of the history buffer If FALSE, the history buffer content is not cleared.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the communication history from the history buffer and stores it in the buffer pointed by **Buf**.

The communication history includes all commands sent to the controller and all responses and unsolicited messages received from the controller. The amount of received data does not exceed the size of the history buffer. The history buffer works as a cyclic buffer: when the amount of the stored data exceeds the buffer size, the newly stored data overwrites the earliest data in the buffer.

Therefore, as a rule, the retrieved communication history includes only the recently sent commands and receives responses and unsolicited messages. The depth of the retrieved history depends on the history buffer size.

The history data is retrieved in historical order, i.e. the earliest message is stored at the beginning of **Buf**. The first retrieved message in **Buf** can be incomplete, because of being partially overwritten in the history buffer.

If the size of the **Buf** is less than the size of the history buffer, only the most recent part of the stored history is retrieved.

Example

```
int Received;
char Buf[10000];
if (!acsc_GetHistory(Handle, // communication handle
        Buf,    // pointer to the buffer that receives a
                        // communication history
        10000,  // size of this buffer
        &Received,      // number of characters that were actually
                        // received
        TRUE            // clear contents of the history buffer
))
{
        printf("getting history error: %d\n", acsc_GetLastError());
}
```

4.8 Unsolicited Messages Buffer Management Functions

The Unsolicited Messages Buffer Management functions are:

Table 5-8. Unsolicited Messages Buffer Management Functions

Function	Description
acsc_OpenMessageBuffer	Opens an unsolicited messages buffer.
acsc_CloseMessageBuffer	Closes an unsolicited messages buffer.
acsc_GetSingleMessage	Retrieves single message or exits by time-out
acsc_GetMessage	Retrieves unsolicited messages from the buffer.

4.8.1 acsc_OpenMessageBuffer

Description

The function opens an unsolicited messages buffer.

Syntax

LP_ACSC_HISTORYBUFFER acsc_OpenMessageBuffer(HANDLE Handle, int Size)

Arguments

Handle	Communication handle
Size	Required size of the buffer in bytes

Return Value

The function returns pointer to **ACSC_HISTORYBUFFER** structure.

If the buffer cannot be allocated, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function allocates a buffer that stores unsolicited messages from the controller.

Unsolicited messages are messages that the controller sends on its own initiative and not as a response to command. For example, the **DISP** command in an ACSPL+ program forces the controller to send an unsolicited message.

Only one message buffer can be open for each communication handle.

If the message buffer has been open, the library separates unsolicited messages from the controller responses and stores them in the message buffer. In this case, the acsc_GetMessage function retrieves unsolicited messages. If the message buffer is not open, **acsc_Receive** retrieves both replies and unsolicited messages. If the user application does not call **acsc_Receive** or **acsc_GetMessage** and uses only acsc_Transaction and acsc_Command, the unsolicited messages are discarded.

The message buffer works as a FIFO buffer: **acsc_GetMessage** extracts the earliest message stored in the buffer. If **acsc_GetMessage** extracts the messages slower than the controller produces them, buffer overflow can occur, and some messages will be lost. Generally, the greater the buffer, the less likely is buffer overflow to occur.

Example

```
LP_ACSC_HISTORYBUFFER lpMessageBuf = acsc_OpenMessageBuffer(
                        Handle,         // communication handle
                        10000           // size of the buffer
                        );
if (!lpMessageBuf)
{
        printf("opening unsolicited messages buffer error: %d\n",
        acsc_GetLastError());
}
```

4.8.2 acsc_CloseMessageBuffer

Description

The function closes the messages buffer and discards all stored unsolicited messages.

Syntax

int acsc_CloseMessageBuffer(HANDLE Handle)

Arguments

Handle	Communication handle

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function closes the message buffer and releases the used memory. All unsolicited messages stored in the buffer are discarded.

Example

```
if (!acsc_CloseMessageBuffer(Handle))
{
        printf("closing unsolicited messages buffer error: %d\n",
                acsc_GetLastError());
}
```

4.8.3 acsc_GetSingleMessage

Description

The function retrieves single unsolicited message from the buffer. This function only works if you setup a buffer using acsc_OpenMessageBuffer. If there is no message in the buffer, the function waits until the message arrives or timeout expires.

Syntax

int acsc_GetSingleMessage (HANDLE Handle, char *Message,int Count,int *Length, int Timeout)

Arguments

Handle	Communication handle
Message	Pointer to the buffer that receives unsolicited messages, it should be at least 1K.
Count	Size of the buffer to which the Message argument points.
Length	The actual length of the message
Timeout	Timeout in milliseconds

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

If message arrives, **Length** will contain the received message length. If the timeout expires, the function exits with the **ACSC_TIMEOUT** error.

Example

```
int L;
char Buf[10000];
if (!acsc_GetSingleMessage(Handle, // communication handle
        Buf,    // pointer to the buffer that
                // receives unsolicited messages
        10000   //Size of the buffer
        &L,     // number of characters that were actually received
        1000    // Wait for 1 second till a message arrives
        ))
{
        printf("getting unsolicited message error: %d\n",
        acsc_GetLastError());
}
```

4.8.4 acsc_GetMessage

Description

The function retrieves unsolicited messages from the buffer. This function only works if you setup a buffer using acsc_OpenMessageBuffer.

Syntax

int acsc_GetMessage(HANDLE Handle, char* Buf, int Count, int* Received, BOOL bClear)

Arguments

Handle	Communication handle
Buf	Pointer to the buffer that receives unsolicited messages
Count	Size of the buffer in bytes
Received	Number of characters that were actually received
bClear	If TRUE, the function clears the contents of the unsolicited messages buffer after retrieving the message. If FALSE, the unsolicited messages buffer is not cleared.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves all stored unsolicited messages from the message buffer.

The function always returns immediately. If no, unsolicited message is received, the function assigns zero to the **Received** variable.

Parameter **Count** specifies the buffer size. If a received message contains more than **Count** characters, the function transfers to buffer only **Count** characters, assigns **Received** with **Count** value and returns non-zero. The remaining characters of the message are removed from the message buffer.

If the **Count** is equal to or more than the length of the message, the function transfers the whole message to buffer and assigns variable **Received** with a number of characters that were actually transferred.

Example

```
int Received;
char Buf[10000];
if (!acsc_GetMessage(    Handle,        // communication handle
                Buf,            // pointer to the buffer that
                                // receives unsolicited messages
                10000,          // size of this buffer
                &Received,      // number of characters that were
                                // actually received
                TRUE            // clear contents of the
                                // unsolicited messages buffer
        ))
{
        printf("getting unsolicited message error: %d\n",
        acsc_GetLastError());
}
```

4.9 Log File Management Functions

The Log File Management functions are:

Table 5-9. Log File Management Functions

Function	Description
acsc_SetLogFileOptions	Sets log file options.
acsc_OpenLogFile	Opens a log file.
acsc_CloseLogFile	Closes a log file.
acsc_WriteLogFile	Writes to a log file.

Function	Description
acsc_FlushLogFile	Flushes the SPiiPlus UMD (User Mode Driver) internal binary buffer to a specified text file from the C Library application.
acsc_GetLogData	Retrieves the data of firmware log.

4.9.1 acsc_SetLogFileOptions

Description

The function sets the log file options.

Syntax

acsc_SetLogFileOptions(HANDLE Handle,
ACSC_LOG_DETALIZATION_LEVEL Detailization,
ACSC_LOG_DATA_PRESENTATION Presentation)

Arguments

Handle	Communication handle
Detailization	This parameter may be one of the following: Minimum -Value 0: Only communication traffic will be logged. Medium - Value 1: Communication traffic and **some** internal C Lib process data will be logged. Maximum -Value 2: Communication traffic and **all** internal process data will be logged.
Presentation	This parameter may be one of the following: Compact -Value 0: No more than the first ten bytes of each data string will be logged. Non-printing characters will be represented in [Hex ASCII code]. Formatted - Value 1: All the binary data will be logged. Non-printing characters will be represented in [Hex ASCII code]. Full -Value 2: All the binary data will be logged as is.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function configures the log file options. The function may be called before or after the log file is opened.

Example

```
//Example of function acsc_SetLogFileOptions
acsc_SetLogFileOptions(Handle,Maximum,Formatted);
```

4.9.2 acsc_OpenLogFile

Description

The function opens a log file.

Syntax

int acsc_OpenLogFile(HANDLE Handle, char* FileName)

Arguments

Handle	Communication handle
FileName	Pointer to the null-terminated string contained name or path of the log file.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function opens a binary file that stores all communication history.

Only one log file can be open for each communication handle.

If the log file has been open, the library writes all incoming and outgoing messages to the specified file. The messages are written to the file in binary format, i.e., exactly as they are received and sent, including all service bytes.

Unlike the history buffer, the log file cannot be read within the library. The main usage of the log file is for debug purposes.

Example

```
if (!acsc_OpenLogFile(Handle, "acs_comm.log"))
{
        printf("opening log file error: %d\n", acsc_GetLastError());
}
```

4.9.3 acsc_CloseLogFile

Description

The function closes the log file.

Syntax

int acsc_CloseLogFile(HANDLE Handle)

Arguments

Handle	Communication handle

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

An application must always call the **acsc_CloseLogFile** before it exits. Otherwise, the data written to the file might be lost.

Example

```
if (!acsc_CloseLogFile(Handle))
{
        printf("closing log file error: %d\n", acsc_GetLastError());
}
```

4.9.4 acsc_WriteLogFile

Description

The function writes to log file.

Syntax

int acsc_WriteLogFile(HANDLE Handle, char* Buf, int Count)

Arguments

Handle	Communication handle
Buf	Pointer to the buffer that contains the string to be written to log file.
Count	Number of characters in the buffer

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function writes data from a buffer to log file.

The log file has to have been opened previously using acsc_OpenLogFile

Example

```
char* str = "Test string";
if (!acsc_WriteLogFile(Handle,  // communication handle
        str,                    // string to be written
        strlen(str)             // length of this string
        ))
{
        printf("error while writing to log file: %d\n", acsc_GetLastError());
}
```

4.9.5 acsc_FlushLogFile

Description

This function allows flushing the SPiiPlus UMD (User Mode Driver) internal binary buffer to a specified text file from the C Library application.

Syntax

acsc_FlushLogFile(char*filename)

Arguments

filename	String that specifies the file name.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

If Continuous Log is active, the function will fail

Example

```
if (!acsc_FlushLogFile( filename))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.9.6 acsc_GetLogData

Description

The function is used to retrieve the data of firmware log.

Syntax

int acsc_GetLogData(HANDLE Handle, char* Buf, int Count, int* Received,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Buf	Pointer to the buffer that receives the log data.
Count	Size of the buffer in bytes.
Received	Number of characters that were actually received.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
const int LogDataMaxSize = 300 * 100;
char LogData[LogDataMaxSize] = {'\0'};
int Received = 0;
if (!acsc_GetLogData(Handle, LogData, LogDataMaxSize, &Received, NULL))
{
        printf("acsc_GetLogData(): Error Occurred - %d\n",
        acsc_GetLastError());
        return;
}
```

4.10 SPiiPlusSC Log File Management Functions

These functions can only be used with the SPiiPlusSC Motion Controller.

SPiiPlusSC Log File Management functions are:

Table 5-10. SPiiPlusSC Log File Management Functions

Function	Description
acsc_OpenSCLogFile	Opens the SPiiPlusSC log file.
acsc_CloseSCLogFile	Closes the SPiiPlusSC log file.
acsc_WriteSCLogFile	Writes data to the SPiiPlusSC log file.
acsc_FlushSCLogFile	Flushes contents of the SPiiPlusSC log file to a specified text file.

4.10.1 acsc_OpenSCLogFile

Description

The function opens the SPiiPlusSC log file.

Syntax

int acsc_OpenSCLogFile(HANDLE Handle, char* FileName)

Arguments

Handle	Communication handle.
FileName	Pointer to the null-terminated string containing the name or path of the log file.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function opens a binary file that stores all SPiiPlusSC log history.

The messages are written to the file in binary format, i.e., exactly as they are received and sent, including all service bytes.

The main use of the log file is for debug purposes.

Example

```
if (!acsc_OpenSCLogFile(Handle, "acs_sc.log"))
{
        printf("opening SPiiPlus SC log file error: %d\n", acsc_GetLastError());
}
```

4.10.2 acsc_CloseSCLogFile

Description

The function closes the SPiiPlusSC log file.

Syntax

int acsc_CloseSCLogFile(HANDLE Handle)

Arguments

Handle	Communication handle.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

An application must always call the **acsc_CloseSCLogFile** before it exits; otherwise, the data written to the file might be lost.

Example

```
if (!acsc_CloseSCLogFile(Handle))
{
        printf("closing log file error: %d\n", acsc_GetLastError());
}
```

4.10.3 acsc_WriteSCLogFile

Description

The function writes to the SPiiPlusSC log file.

Syntax

int acsc_WriteSCLogFile(HANDLE Handle, char* Buf, int Count)

Arguments

Handle	Communication handle.
Buf	Pointer to the buffer that contains the string to be written to the log file.
Count	Number of characters in the buffer

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function writes data from a buffer to the SPiiPlusSC log file.

The log file has to have been opened previously using acsc_OpenLogFile

Example

```
char* str = "Test string";
if (!acsc_WriteSCLogFile(Handle,// communication handle
        str,                    // string to be written
        strlen(str)             // length of this string
        ))
{
        printf("error while writing to log file: %d\n", acsc_GetLastError());
}
```

4.10.4 acsc_FlushSCLogFile

Description

The function enables flushing the SPiiPlusSC internal binary buffer to a specified text file from the C Library application.

Syntax

int acsc_FlushSCLogFile(char*Filename, BOOL bClear)

Arguments

Filename	String that specifies the file name.
bClear	Can be TRUE or FALSE: TRUE - the function clears contents of the log buffer FALSE - the log buffer content is not cleared

Comments

If Continuous Log is active, the function will fail.

Example

```
if (!acsc_FlushSCLogFile(filename,TRUE))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.11 System Configuration Functions

System Configuration functions are:

Table 5-11. System Configuration Functions

Function	Description
acsc_SetConf	The function writes system configuration data.
acsc_GetConf	The function reads system configuration data.
acsc_GetVolatileMemoryUsage	The function retrieves the percentage of the volatile memory load.
acsc_GetVolatileMemoryTotal	The function retrieves the amount of total volatile memory in bytes.
acsc_GetVolatileMemoryFree	The function retrieves the amount of free volatile memory in bytes.
acsc_ GetNonVolatileMemoryUsage	The function retrieves the percentage of the non-volatile memory load.
acsc_GetNonVolatileMemoryTotal	The function retrieves the amount of total non-volatile memory in bytes.
acsc_GetNonVolatileMemoryFree	The function retrieves the amount of free non-volatile memory in bytes.
acsc_SysInfo	The function returns certain system information based on the argument that is specified.

4.11.1 acsc_SetConf

Description

The function writes system configuration data.

Syntax

int acsc_SetConf(HANDLE Handle, int Key, int Index, double Value,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Key	Configuration key, see Configuration Keys, that specifies the configured feature. Assigns value of ***key*** argument in ACSPL+ **SETCONF** function.
Index	Specifies corresponding axis or buffer number. Assigns value of ***index*** argument in ACSPL+ **SETCONF** function.
Value	Value to write to specified key. Assigns value of ***value*** argument in ACSPL+ **SETCONF** function.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCallfunction to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function writes system configuration data. The Key parameter specifies the feature number and the Index parameter defines axis or buffer to which it should be applied. Use ACSC_CONF_XXX constants in the value field. For complete details of system configuration see the description of the **SETCONF** function in the *SPiiPlus ACSPL+ Programmer's Guide*.

Example

```
// example of the waiting call of acsc_Track
if (!acsc_SetConf(Handle,                          // communication handle
        ACSC_CONF_DIGITAL_SOURCE_KEY,              // 205
        ACSC_AXIS_0,                               // of the axis 0
        0,
        NULL                                       // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.11.2 acsc_GetConf

Description

The function reads system configuration data.

Syntax

int acsc_GetConf(HANDLE Handle, int Key, int Index, double *Value,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Key	Configuration Key, see Configuration Keys, specifies the configured feature. Assigns value of ***key*** argument in ACSPL+ **GETCONF** function.
Index	Specifies corresponding axis or buffer number. Assigns value of ***index*** argument in ACSPL+ **GETCONF** function.
Value	Receives the configuration data
Wait	Pointer to **ACSC_WAITBLOCK** structure. If **Wait** is **ACSC_SYNCHRONOUS**, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function reads system configuration data. The **Key** parameter specifies the feature number and the Index parameter defines axis or buffer to which it should be applied. For detailed description of system configuration see "*SPiiPlus ACSPL+ Programmer's Guide*" for the details of the **GETCONF** function.

Example

```
// example of the waiting call of acsc_Track
if (!acsc_GetConf(      Handle,                         // communication handle
                ACSC_CONF_DIGITAL_SOURCE_KEY,         // 205
                ACSC_AXIS_0,                          // of the 0 axis
                &Value,
                NULL                                  // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.11.3 acsc_GetVolatileMemoryUsage

Description

The function retrieves the percentage of the volatile memory load.

Syntax

int _ACSCLIB_ WINAPI acsc_GetVolatileMemoryUsage(HANDLE Handle, double* Value, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Value	Pointer to a variable that receives the percentage of the volatile memory load
Wait	Pointer to **ACSC_WAITBLOCK** structure. If **Wait** is **ACSC_SYNCHRONOUS**, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is **ACSC_IGNORE**, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
// Example of the waiting call of acsc_GetVolatileMemoryUsage
// The function retrieves the volatile memory load in percentage
double Value;
if (!acsc_GetVolatileMemoryUsage(
        Handle,                    // communication handle
```

```
        &Value,                     // received value
        NULL                         // waiting call
        ))
{
     printf("acsc_GetVolatileMemoryUsage(): Error Occurred - %d\n",
         acsc_GetLastError());
}
```

4.11.4 acsc_GetVolatileMemoryTotal

Description

The function retrieves the amount of total volatile memory in bytes.

Syntax

int _ACSCLIB_ WINAPI acsc_GetVolatileMemoryTotal(HANDLE Handle, double* Value, ACSC_ WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Value	Pointer to a variable that receives the amount of total volatile memory in bytes.
Wait	Pointer to **ACSC_WAITBLOCK** structure. If **Wait** is **ACSC_SYNCHRONOUS**, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is **ACSC_IGNORE**, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
// Example of the waiting call of acsc_GetVolatileMemoryTotal
// The function retrieves the amount of total volatile memory in bytes
double Value;
if (!acsc_GetVolatileMemoryTotal(
        Handle,                     // communication handle
        &Value,                    // received value
        NULL                        // waiting call
        ))
{
     printf("acsc_GetVolatileMemoryTotal(): Error Occurred - %d\n",
          acsc_GetLastError());
}
```

4.11.5 acsc_GetVolatileMemoryFree

Description

The function retrieves the amount of free volatile memory in bytes.

Syntax

int _ACSCLIB_ WINAPI acsc_GetVolatileMemoryFree(HANDLE Handle, double* Value, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Value	Pointer to a variable that receives the amount of free volatile memory in bytes
Wait	Pointer to **ACSC_WAITBLOCK** structure. If **Wait** is **ACSC_SYNCHRONOUS**, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is **ACSC_IGNORE**, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
// Example of the waiting call of acsc_GetVolatileMemoryFree
// The function retrieves the amount of free volatile memory in bytes
double Value;
```

```
if (!acsc_GetVolatileMemoryFree(
        Handle,                      // communication handle
        &Value,                     // received value
        NULL                         // waiting call
        ))
{
     printf("acsc_GetVolatileMemoryFree(): Error Occurred - %d\n",
         acsc_GetLastError());
}
```

4.11.6 acsc_GetNonVolatileMemoryUsage

Description

The function retrieves the percentage of the non-volatile memory load.

Syntax

int _ACSCLIB_ WINAPI acsc_GetNonVolatileMemoryUsage(HANDLE Handle, double* Value, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Value	Pointer to a variable that receives the percentage of the non-volatile memory load.
Wait	Pointer to **ACSC_WAITBLOCK** structure. If **Wait** is **ACSC_SYNCHRONOUS**, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is **ACSC_IGNORE**, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
// Example of the waiting call of acsc_GetNonVolatileMemoryUsage
// The function retrieves the non-volatile memory load in percentage
double Value;
if (!acsc_GetNonVolatileMemoryUsage(
        Handle,                    // communication handle
        &Value,                   // received value
        NULL                       // waiting call
        ))
{
     printf("acsc_GetNonVolatileMemoryUsage(): Error Occurred - %d\n",
         acsc_GetLastError());
}
```

4.11.7 acsc_GetNonVolatileMemoryTotal

Description

The function retrieves the amount of total non-volatile memory in bytes.

Syntax

int _ACSCLIB_ WINAPI acsc_GetNonVolatileMemoryTotal(HANDLE Handle, double* Value, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Value	Pointer to a variable that receives the amount of total non-volatile memory in bytes
Wait	Pointer to **ACSC_WAITBLOCK** structure. If **Wait** is **ACSC_SYNCHRONOUS**, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is **ACSC_IGNORE**, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
// Example of the waiting call of acsc_GetNonVolatileMemoryTotal
// The function retrieves the amount of total non-volatile memory in
```

```
bytes
double Value;
if (!acsc_GetNonVolatileMemoryTotal(
        Handle,                     // communication handle
        &Value,                     // received value
        NULL                        // waiting call
        ))
{
     printf("acsc_GetNonVolatileMemoryTotal(): Error Occurred - %d\n",
         acsc_GetLastError());
}
```

4.11.8 acsc_GetNonVolatileMemoryFree

Description

The function retrieves the amount of free non-volatile memory in bytes.

Syntax

int _ACSCLIB_ WINAPI acsc_GetNonVolatileMemoryFree(HANDLE Handle, double* Value, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Value	Pointer to a variable that receives the amount of total non-volatile memory in bytes
Wait	Pointer to **ACSC_WAITBLOCK** structure. If **Wait** is **ACSC_SYNCHRONOUS**, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is **ACSC_IGNORE**, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
// Example of the waiting call of acsc_GetNonVolatileMemoryFree
// The function retrieves the amount of free non-volatile memory in bytes
double Value;
if (!acsc_GetNonVolatileMemoryFree(
        Handle,                    // communication handle
        &Value,                   // received value
        NULL                       // waiting call
        ))
{
     printf("acsc_GetNonVolatileMemoryFree(): Error Occurred - %d\n",
          acsc_GetLastError());
}
```

4.11.9 acsc_SysInfo

Description

The function returns certain system information based on the argument that is specified.

Syntax

int acsc_SysInfo(HANDLE Handle, int Key, double *Value,
 ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Key	Configuration Key, see System Information Keys, specifies the configured feature. Assigns value of key argument in ACSPL+ **SYSINFO** function.
Value	Receives the configuration data
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
double Value = 0;
if (!acsc_SysInfo(Handle, ACSC_SYS_MODEL_KEY, &Value, NULL))
{
        printf("acsc_SysInfo(): Error Occurred - %d\n", acsc_GetLastError());
        return;
}
```

4.12 Setting and Reading Motion Parameters Functions

The Setting and Reading Motion Parameters functions are:

Table 5-12. Setting and Reading Motion Parameters Functions

Function	Description
acsc_SetVelocity	Defines a value of motion velocity.
acsc_GetVelocity	Retrieves a value of motion velocity.
acsc_SetAcceleration	Defines a value of motion acceleration.
acsc_GetAcceleration	Retrieves a value of motion acceleration.
acsc_SetDeceleration	Defines a value of motion deceleration.
acsc_GetDeceleration	Retrieves a value of motion deceleration.
acsc_SetJerk	Defines a value of motion jerk.
acsc_GetJerk	Retrieves a value of motion jerk.
acsc_SetKillDeceleration	Defines a value of kill deceleration.
acsc_GetKillDeceleration	Retrieves a value of kill deceleration.
acsc_SetVelocityImm	Defines a value of motion velocity. Unlike **acsc_SetVelocity**, the function has immediate effect on any executed and planned motion.
acsc_SetAccelerationImm	Defines a value of motion acceleration. Unlike **acsc_SetAcceleration**, the function has immediate effect on any executed and planned motion.
acsc_SetDecelerationImm	Defines a value of motion deceleration. Unlike **acsc_SetDeceleration**, the function has immediate effect on any executed and planned motion.

Function	Description
acsc_SetJerkImm	Defines a value of motion jerk. Unlike **acsc_SetJerk**, the function has an immediate effect on any executed and planned motion.
acsc_SetKillDecelerationImm	Defines a value of kill deceleration. Unlike **acsc_SetKillDeceleration**, the function has immediate effect on any executed and planned motion.
acsc_SetFPosition	Assigns a current value of feedback position.
acsc_GetFPosition	Retrieves a current value of motor feedback position.
acsc_SetRPosition	Assigns a current value of reference position.
acsc_GetRPosition	Retrieves a current value of reference position.
acsc_GetFVelocity	Retrieves a current value of motor feedback velocity.
acsc_GetRVelocity	Retrieves a current value of reference velocity.

4.12.1 acsc_SetVelocity

Description

The function defines a value of motion velocity.

Syntax

int acsc_SetVelocity(HANDLE Handle, int Axis, double Velocity,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1,etc. For the axis constants see Axis Definitions.
Velocity	The value specifies required motion velocity. The value will be used in the subsequent motions except for the master-slave motions and the motions activated with the ACSC_AMF_VELOCITY flag.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result.

	If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function writes the specified value to the controller.

One value can be specified for each axis.

A single-axis motion uses the value of the corresponding axis. A multi-axis motion uses the value of the leading axis. The leading axis is an axis specified first in the motion command.

The function affects the motions initiated after the function call. The function has no effect on any motion that was started or planned before the function call. To change velocity of an executed or planned motion, use the acsc_SetVelocityImm function.

The function has no effect on the master-slave motions and the motions activated with the ACSC_AMF_VELOCITY flag.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SetVelocity
if (!acsc_SetVelocity(          Handle,         // communication handle
                                ACSC_AXIS_0,    // axis 0
                                10000,          // velocity value
                                NULL            // waiting call
                                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.12.2 acsc_GetVelocity

Description

The function retrieves a value of motion velocity.

Syntax

int acsc_GetVelocity(HANDLE Handle, int Axis, double* Velocity,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1,etc. For the axis constants see Axis Definitions.
Velocity	Pointer to the variable that receives the value of motion velocity.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the value of the motion velocity. The retrieved value is a value defined by a previous call of the acsc_SetVelocity function, or the default value if the function was not called before.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Velocity** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetVelocity
double Velocity;
if (!acsc_GetVelocity(Handle,           // communication handle
                ACSC_AXIS_0,            // axis 0
                &Velocity,              // received value
                NULL                    // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.12.3 acsc_SetAcceleration

Description

The function defines a value of motion acceleration.

Syntax

int acsc_SetAcceleration(HANDLE Handle, int Axis, double Acceleration,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1,etc. For the axis constants see Axis Definitions.
Acceleration	The value specifies required motion acceleration. The value will be used in the subsequent motions except the master-slave motions.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function writes the specified value to the controller.

One value can be specified for each axis.

A single-axis motion uses the value of the corresponding axis. A multi-axis motion uses the value of the leading axis. The leading axis is an axis specified first in the motion command.

The function affects the motions initiated after the function call. The function has no effect on any motion that was started or planned before the function call. To change acceleration of an executed or planned motion, use the acsc_SetAccelerationImm function.

The function has no effect on the master-slave motions.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SetAcceleration
if (!acsc_SetAcceleration(      Handle,         // communication handle
                        ACSC_AXIS_0,            // axis 0
                        100000,                 // acceleration value
                        NULL                    // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.12.4 acsc_GetAcceleration

Description

The function retrieves a value of motion acceleration.

Syntax

int acsc_GetAcceleration(HANDLE Handle, int Axis, double* Acceleration,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1,etc. For the axis constants Axis Definitions.
Acceleration	Pointer to the variable that receives the value of motion acceleration.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the value of the motion acceleration. The retrieved value is a value defined by a previous call of the acsc_SetAcceleration function, or the default value if the function was not called before.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Acceleration** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetAcceleration
double Acceleration;
if (!acsc_GetAcceleration(      Handle,         // communication handle
                        ACSC_AXIS_0,            // axis 0
                        &Acceleration,          // received value
                        NULL                    // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.12.5 acsc_SetDeceleration

Description

The function defines a value of motion deceleration.

Syntax

int acsc_SetDeceleration(HANDLE Handle, int Axis, double Deceleration,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1,etc. For the axis constants see Axis Definitions.
Deceleration	The value specifies a required motion deceleration. The value will be used in the subsequent motions except the master-slave motions.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result.

	If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function writes the specified value to the controller.

One value can be specified for each axis.

A single-axis motion uses the value of the corresponding axis. A multi-axis motion uses the value of the leading axis. The leading axis is an axis specified first in the motion command.

The function affects the motions initiated after the function call. The function has no effect on any motion that was started or planned before the function call. To change deceleration of an executed or planned motion, use the acsc_SetDecelerationImm function.

The function has no effect on the master-slave motions.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SetDeceleration
if (!acsc_SetDeceleration(      Handle,         // communication handle
                ACSC_AXIS_0,                    // axis 0
                100000,                         // deceleration value
                NULL                            // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.12.6 acsc_GetDeceleration

Description

The function retrieves a value of motion deceleration.

Syntax

int acsc_GetDeceleration(HANDLE Handle, int Axis, double* Deceleration,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1,etc. For the axis constants see Axis Definitions
Deceleration	Pointer to the variable that receives the value of motion deceleration.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the value of the motion deceleration. The retrieved value is a value defined by a previous call of the acsc_SetDeceleration function, or the default value if the function was not previously called.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Deceleration** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetDeceleration
double Deceleration;
if (!acsc_GetDeceleration(      Handle,         // communication handle
                ACSC_AXIS_0,                    // axis 0
                &Deceleration,                  // received value
                NULL                            // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.12.7 acsc_SetJerk

Description

The function defines a value of motion jerk.

Syntax

int acsc_SetJerk(HANDLE Handle, int Axis, double Jerk, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
Jerk	The value specifies a required motion jerk. The value will be used in the subsequent motions except for the master-slave motions.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function writes the specified value to the controller.

One value can be specified for each axis.

A single-axis motion uses the value of the corresponding axis. A multi-axis motion uses the value of the leading axis. The leading axis is an axis specified first in the motion command.

The function affects the motions initiated after the function call. The function has no effect on any motion that was started or planned before the function call. To change the jerk of an executed or planned motion, use the acsc_SetJerkImm function.

The function has no effect on the master-slave motions.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SetJerk
if (!acsc_SetJerk(       Handle,                // communication handle
                        ACSC_AXIS_0,            // axis 0
                        1000000,                // jerk value
                        NULL                    // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.12.8 acsc_GetJerk

Description

The function retrieves a value of motion jerk.

Syntax

int acsc_GetJerk(HANDLE Handle, int Axis, double* Jerk,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
Jerk	Pointer to the variable that receives the value of motion jerk.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the value of the motion jerk. The retrieved value is a value defined by a previous call of the acsc_SetJerk function, or the default value if the function was not called before.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Jerk** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetJerk
double Jerk;
if (!acsc_GetJerk(      Handle,          // communication handle
                        ACSC_AXIS_0,     // axis 0
                        &Jerk,           // received value
                        NULL             // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.12.9 acsc_SetKillDeceleration

Description

The function defines a value of motion kill deceleration.

Syntax

int acsc_SetKillDeceleration(HANDLE Handle, int Axis, double KillDeceleration,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
KillDeceleration	The value specifies a required motion kill deceleration. The value will be used in the subsequent motions.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result.

	If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function writes the specified value to the controller.

One value can be specified for each axis.

A single-axis motion uses the value of the corresponding axis. A multi-axis motion uses the value of the leading axis. The leading axis is an axis specified first in the motion command.

The function affects the motions initiated after the function call. The function has no effect on any motion that was started or planned before the function call.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SetKillDeceleration
double KillDeceleration;
if (!acsc_SetKillDeceleration(  Handle,         // communication handle
                                ACSC_AXIS_0,    // axis 0
                                100000,         // kill deceleration value
                                NULL            // waiting call
                                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.12.10 acsc_GetKillDeceleration

Description

The function retrieves a value of motion kill deceleration.

Syntax

int acsc_GetKillDeceleration(HANDLE Handle, int Axis, double* KillDeceleration,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
KillDeceleration	Pointer to the variable that receives the value of motion kill deceleration.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the value of the motion kill deceleration. The retrieved value is a value defined by a previous call of the acsc_SetKillDeceleration function, or the default value if the function was not called before.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **KillDeceleration** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetKillDeceleration
double KillDeceleration;
if (!acsc_GetKillDeceleration(Handle,           // communication handle
                        ACSC_AXIS_0,            // axis 0
                        &KillDeceleration,      // received value
                        NULL                    // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.12.11 acsc_SetVelocityImm

Description

The function defines a value of motion velocity. Unlike acsc_SetVelocity, the function has immediate effect on any executed or planned motion.

Syntax

int acsc_SetVelocityImm(HANDLE Handle, int Axis, double Velocity,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
Velocity	The value specifies required motion velocity.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function writes the specified value to the controller.

One value can be specified for each axis.

A single-axis motion uses the value of the corresponding axis. A multi-axis motion uses the value of the leading axis. The leading axis is an axis specified first in the motion command.

The function affects:

- The currently executed motion. The controller provides a smooth transition from the instant current velocity to the specified new value.
- The waiting motions that were planned before the function call.

> The motions that will be commanded after the function call.

The function has no effect on the master-slave motions.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SetVelocityImm
if (!acsc_SetVelocityImm(Handle,        // communication handle
                ACSC_AXIS_0,            // axis 0
                10000,                  // velocity value
                NULL                    // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.12.12 acsc_SetAccelerationImm

Description

The function defines a value of motion acceleration. Unlike acsc_SetAcceleration, the function has immediate effect on any executed and planned motion.

Syntax

int acsc_SetAccelerationImm(HANDLE Handle, int Axis, double Acceleration,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
Acceleration	The value specifies required motion acceleration.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function writes the specified value to the controller.

One value can be specified for each axis.

A single-axis motion uses the value of the corresponding axis. A multi-axis motion uses the value of the leading axis. The leading axis is an axis specified first in the motion command.

The function affects:

- The currently executed motion.
- The waiting motions that were planned before the function call.
- The motions that will be commanded after the function call.

The function has no effect on the master-slave motions.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SetAccelerationImm
if (!acsc_SetAccelerationImm(Handle,            // communication handle
                        ACSC_AXIS_0,            // axis 0
                        100000,                 // acceleration value
                        NULL                    // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.12.13 acsc_SetDecelerationImm

Description

The function defines a value of motion deceleration. Unlike acsc_SetDeceleration, the function has immediate effect on any executed and planned motion.

Syntax

int acsc_SetDecelerationImm(HANDLE Handle, int Axis, double Deceleration,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.

Deceleration	The value specifies required motion deceleration.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function writes the specified value to the controller.

One value can be specified for each axis.

A single-axis motion uses the value of the corresponding axis. A multi-axis motion uses the value of the leading axis. The leading axis is an axis specified first in the motion command.

The function affects:

- The currently executed motion.
- The waiting motions that were planned before the function call.
- The motions that will be commanded after the function call.

The function has no effect on the master-slave motions.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SetDecelerationImm
if (!acsc_SetDecelerationImm(Handle,           // communication handle
                        ACSC_AXIS_0,           // axis 0
                        100000,                // deceleration value
                        NULL                   // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.12.14 acsc_SetJerkImm

Description

The function defines a value of motion jerk. Unlike acsc_SetJerk, the function has immediate effect on any executed and planned motion.

Syntax

int acsc_SetJerkImm(HANDLE Handle, int Axis, double Jerk,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
Jerk	The value specifies required motion jerk.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function writes the specified value to the controller.

One value can be specified for each axis.

A single-axis motion uses the value of the corresponding axis. A multi-axis motion uses the value of the leading axis. The leading axis is an axis specified first in the motion command.

The function affects:

- The currently executed motion.
- The waiting motions that were planned before the function call.
- The motions that will be commanded after the function call.

The function has no effect on the master-slave motions.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
/ example of the waiting call of acsc_SetJerkImm
if (!acsc_SetJerkImm(Handle,            // communication handle
                        ACSC_AXIS_0,    // axis 0
                        1000000,        // jerk value
                        NULL            // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.12.15 acsc_SetKillDecelerationImm

Description

The function defines a value of motion kill deceleration. Unlike acsc_SetKillDeceleration, the function has an immediate effect on any executed and planned motion.

Syntax

int acsc_SetKillDecelerationImm(HANDLE Handle, int Axis, double Deceleration,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
KillDeceleration	The value specifies the required motion deceleration.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function writes the specified value to the controller.

One value can be specified for each axis.

A single-axis motion uses the value of the corresponding axis. A multi-axis motion uses the value of the leading axis. The leading axis is an axis specified first in the motion command.

The function affects:

- The currently executed motion
- The waiting motions that were planned before the function call
- The motions that will be commanded after the function call

The function has no effect on the master-slave motions.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SetKillDecelerationImm
if (!acsc_SetKillDecelerationImm(Handle,        / communication handle
                        ACSC_AXIS_0,            // axis 0
                        100000,                 // kill deceleration value
                        NULL                    // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.12.16 acsc_SetFPosition

Description

The function assigns a current value of feedback position.

Syntax

int acsc_SetFPosition(HANDLE Handle, int Axis, double FPosition,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
FPosition	The value specifies the current value of feedback position.

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function assigns a current value to the feedback position. No physical motion occurs. The motor remains in the same position; only the internal controller offsets are adjusted so that the periodic calculation of the feedback position will provide the required results.

For more information see the explanation of the **SET** command in the *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SetFPosition
if (!acsc_SetFPosition(Handle,  // communication handle
                               ACSC_AXIS_0,    // axis 0
                               0,              // required feedback position
                               NULL            // waiting call
                               ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.12.17 acsc_GetFPosition

Description

The function retrieves an instant value of the motor feedback position.

Syntax

int acsc_GetFPosition(HANDLE Handle, int Axis, double* FPosition,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
FPosition	Pointer to a variable that receives the instant value of the motor feedback position.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves an instant value of the motor feedback position. The feedback position is a measured position of the motor transferred to user units.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **FPosition** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetFPosition
double FPosition;
if (!acsc_GetFPosition(Handle,          // communication handle
                       ACSC_AXIS_0,     // axis 0
                       &FPosition,      // received value
                       NULL             // waiting call
                       ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.12.18 acsc_SetRPosition

Description

The function assigns a current value of reference position.

Syntax

int acsc_SetRPosition(HANDLE Handle, int Axis, double RPosition,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
Rposition	The value specifies the current value of reference position.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function assigns a current value to the reference position. No physical motion occurs. The motor remains in the same position; only the internal controller offsets are adjusted so that the periodic calculation of the reference position will provide the required results.

For more information see explanation of the **SET** command in the *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SetRPosition
if (!acsc_SetRPosition(Handle,          // communication handle
                ACSC_AXIS_0,            // axis 0
                0,                      // required reference position
                NULL                    // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
```

4.12.19 acsc_GetRPosition

Description

The function retrieves an instant value of the motor reference position.

Syntax

int acsc_GetRPosition(HANDLE Handle, int Axis, double* RPosition,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
Rposition	Pointer to a variable that receives the instant value of the motor reference position.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves an instant value of the motor reference position. The reference position is a value calculated by the controller as a reference for the motor.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **RPosition** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetRPosition
double RPosition;
if (!acsc_GetRPosition(Handle,          // communication handle
                ACSC_AXIS_0,            // axis 0
                &RPosition,             // received value
                NULL                    // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.12.20 acsc_GetFVelocity

Description

The function retrieves an instant value of the motor feedback velocity. Unlike acsc_GetVelocity, the function retrieves the actually measured velocity and not the required value.

Syntax

int acsc_GetFVelocity(HANDLE Handle, int Axis, double* FVelocity,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
FVelocity	Pointer to a variable that receives the instant value of the motor feedback velocity.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves an instant value of the motor feedback velocity. The feedback velocity is a measured velocity of the motor transferred to user units.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **FVelocity** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
/ example of the waiting call of acsc_GetFVelocity
double FVelocity;
if (!acsc_GetFVelocity( Handle,                          // communication handle
                        ACSC_AXIS_0,                     // axis 0
                        &FVelocity,                      // received value
                        NULL                             // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.12.21 acsc_GetRVelocity

Description

The function retrieves an instant value of the motor reference velocity.

Syntax

int acsc_GetRVelocity(HANDLE Handle, int Axis, double* RVelocity,
 ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
RVelocity	Pointer to a variable that receives the instant value of the motor reference velocity.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received.

	If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves an instant value of the motor reference velocity. The reference velocity is a value calculated by the controller in the process of motion generation.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **RVelocity** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetRVelocity
double RVelocity;
if (!acsc_GetRVelocity( Handle,                 // communication handle
                        ACSC_AXIS_0,            // axis 0
                        &RVelocity,             // received value
                        NULL                    // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.13 Axis/Motor Management Functions

The Axis/Motor Management functions are:

Table 5-13. Axis/Motor Management Functions

Function	Description
acsc_CommutExt	Initiates a motor commutation.
acsc_Enable	Activates a motor.
acsc_EnableM	Activates several motors.
acsc_Disable	Shuts off a motor.

Function	Description
acsc_DisableAll	Shuts off all motors.
acsc_DisableExt	Shuts off a motor and defines the disable reason.
acsc_DisableM	Shuts off several motors.
acsc_Group	Creates a coordinate system for a multi-axis motion.
acsc_Split	Breaks down an axis group created before.
acsc_SplitAll	Breaks down all axis groups created before.

4.13.1 acsc_CommutExt

This function replaces the acsc_Commut function which is now obsolete.

Description

This function initiates a motor commutation.

Syntax

Int acsc_CommutExt(HANDLE handle, int Axis, float Current, int Settle, int Slope, ACSC_WAITBLOCK *Wait)

Arguments

Handle	Communication handle
Axes	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc.
Current	Desired excitation current in percentage 0-100, ACSC_NONE for default value.
Settle	Specifies the time it takes for auto commutation to settle, in milliseconds. ACSC_NONE for the default value of 500ms.
Slope	Specifies the time it takes for the current to rise to the desired value, ACSC_ NONE for default value.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result.

	If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Values

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function activates a motor. After the activation, the motor begins to follow the reference and physical motion is available.

Settle can only be set if Current is set. Similarly Slope can only be set if Settle is set.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the Wait item until a call to the acsc_WaitForAsyncCall function.

Example

```
if(!acsc_CommutExt(Handle,// Communication handle
ACSC_AXIS_0, // Commuting axis 0
43, // Commutation current
ACS_NONE, // Use default settle
ACS_NONE, // Use default slope
ACSC_SYNCHRONOUS // Waiting call
)){
printf("commutation error: %d\n",acsc_GetLastError());
}
```

4.13.2 acsc_Enable

Description

The function activates a motor.

Syntax

int acsc_Enable(HANDLE Handle, int Axis, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function activates a motor. After the activation, the motor begins to follow the reference and physical motion is available.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_Enable
if (!acsc_Enable(       Handle,                 // communication handle
                        ACSC_AXIS_0,            // enable of axis 0
                        NULL                    // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.13.3 acsc_EnableM

Description

The function activates several motors.

Syntax

int acsc_EnableM(HANDLE Handle, int* Axes, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains -1 which marks the end of the array. For the axis constants see Axis Definitions.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function activates several motors. After the activation, the motors begin to follow the corresponding reference and physical motions for the specified motors are available.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_EnableM
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, ACSC_AXIS_4, ACSC_AXIS_5, -1 };
if (!acsc_EnableM(      Handle,         // communication handle
                        Axes,           // enable of axes 0, 1, 4, and 5
                        NULL            // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.13.4 acsc_Disable

Description

The function shuts off a motor.

Syntax

int acsc_Disable(HANDLE Handle, int Axis, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the acsc_WaitForAsyncCall function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function shuts off a motor. After shutting off the motor cannot follow the reference and remains at idle.

If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_Disable
if (!acsc_Disable(      Handle,                 // communication handle
                        ACSC_AXIS_0,            // disable of axis 0
                        NULL                    // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.13.5 acsc_DisableAll

Description

The function shuts off all motors.

Syntax

int acsc_DisableAll(HANDLE Handle, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function shuts off all motors. After the shutting off none of motors can follow the corresponding, reference and all motors remain idle.

If no motors are currently enabled, the function has no effect.

If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_DisableAll
if (!acsc_DisableAll(Handle, NULL))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.13.6 acsc_DisableExt

Description

The function shuts off a motor and defines the disable reason.

Syntax

int acsc_DisableExt(HANDLE Handle, int Axis,int Reason,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
Reason	Integer number that defines the reason of disable. The specified value is stored in the MERR variable in the controller and so modifies the state of the disabled motor.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function shuts off a motor. After shutting off the motor cannot follow the reference and remains at idle.

If **Reason** specifies one of the available motor termination codes, the state of the disabled motor will be identical to the state of the motor disabled for the corresponding fault. This provides an enhanced implementation of user-defined fault response.

If the second parameter specifies an arbitrary number, the motor state will be displayed as "Kill/disable reason: <number> - customer code. This provides ability to separate different DISABLE commands in the application.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_DisableExt
#define MY_MOTOR_FAULT 10
if (!acsc_DisableExt(Handle,            // communication handle
                ACSC_AXIS_0,            // disable of axis 0
                MY_MOTOR_FAULT,         // internal customer code
                NULL                    // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.13.7 acsc_DisableM

Description

The function shuts off several specified motors.

Syntax

int acsc_DisableM(HANDLE Handle, int* Axes, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains -1 which marks the end of the array. For the axis constants see Axis Definitions.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function shuts off several motors. After the shutting off, the motors cannot follow the corresponding reference and remain idle.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_DisableM
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, ACSC_AXIS_4, ACSC_AXIS_5, -1 };
if (!acsc_DisableM(     Handle,         // communication handle
                        Axes,           // disable axes 0, 1, 4 and 5
                        NULL            // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.13.8 acsc_Group

Description

The function creates a coordinate system for a multi-axis motion.

Syntax

int acsc_Group(HANDLE Handle, int* Axes, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains -1 which marks the end of the array. For the axis constants see Axis Definitions.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function creates a coordinate system for a multi-axis motion. The first element of the **Axes** array specifies the leading axis. The motion parameters of the leading axis become the default motion parameters for the group.

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. To split the existing group, use acsc_Split function. To split all existing groups, use the acsc_SplitAll function.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_Group
int Axes[] = { A        CSC_AXIS_0, ACSC_AXIS_1, ACSC_AXIS_4, ACSC_AXIS_5, -1 };
if (!acsc_Group(        Handle,          // communication handle
                        Axes,            // create group of axes 0, 1, 4 and 5
                        NULL             // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.13.9 acsc_Split

Description

The function breaks down an axis group created before.

Syntax

int acsc_Split(HANDLE Handle, int* Axes, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains -1 which marks the end of the array. For the axis constants see Axis Definitions.

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function breaks down an axis group created before by the acsc_Group function. The **Axes** parameter must specify the same axes as for the **acsc_Group** function that created the group. After the splitting up the group no longer exists.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_Split
int Axes[] = {  ACSC_AXIS_0, ACSC_AXIS_1, ACSC_AXIS_2, ACSC_AXIS_3,
                        ACSC_AXIS_4, ACSC_AXIS_5, ACSC_AXIS_6,
                        ACSC_AXIS_7, -1 };
if (!acsc_Split(        Handle,         // communication handle
                        Axes,           // split the group of axes 0, 1, 2,
                                        // 3, 4, 5, 6 and 7
                        NULL            // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.13.10 acsc_SplitAll

Description

The function breaks down all axis groups created before.

Syntax

int acsc_SplitAll(HANDLE Handle, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function breaks down all axis groups created before by the acsc_Group function.

The application may call this function to ensure that no axes are grouped. If no groups are currently existed, the function has no effect.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SplitAll
if (!acsc_SplitAll(Handle, NULL))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.14 Motion Management Functions

The Motion Management functions are:

Table 5-14. Motion Management Functions

Function	Description
acsc_Go	Starts up a motion that is waiting in the specified motion queue.
acsc_GoM	Synchronously starts up several motions that are waiting in the specified motion queues.
acsc_Halt	Terminates a motion using the full deceleration profile.
acsc_HaltM	Terminates several motions using the full deceleration profile.
acsc_Kill	Terminates a motion using the reduced deceleration profile.
acsc_KillAll	Terminates all currently executed motions.
acsc_KillM	Terminates several motions using the reduced deceleration profile.
acsc_KillExt	Terminates a motion using reduced deceleration profile and defines the kill reason.
acsc_Break	Terminates a motion immediately and provides a smooth transition to the next motion.
acsc_BreakM	Terminates several motions immediately and provides a smooth transition to the next motions.

4.14.1 acsc_Go

Description

The function starts up a motion waiting in the specified motion queue.

Syntax

int acsc_Go(HANDLE Handle, int Axis, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc. For the axis constants see Axis Definitions.

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

A motion that was planned with ACSC_AMF_WAIT flag does not start until the **acsc_Go** function is executed. Being planned, a motion waits in the appropriate motion queue.

Each axis has a separate motion queue. A single-axis motion waits in the motion queue of the corresponding axis. A multi-axis motion waits in the motion queue of the leading axis. The leading axis is an axis specified first in the motion command.

The **acsc_Go** function initiates the motion waiting in the motion queue of the specified axis. If no motion waits in the motion queue, the function has no effect.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_Go
if (!acsc_Go(Handle,            // communication handle
        ACSC_AXIS_0,            // start up the motion of axis 0
        NULL                    // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.14.2 acsc_GoM

Description

The function synchronously starts up several motions that are waiting in the specified motion queues.

Syntax

int acsc_GoM(HANDLE Handle, int* Axes, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains -1 which marks the end of the array. For the axis constants see Axis Definitions.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

A motion that was planned with ACSC_AMF_WAIT flag does not start until the **acsc_GoM** function is executed. After being planned, a motion waits in the appropriate motion queue.

Each axis has a separate motion queue. A single-axis motion waits in the motion queue of the corresponding axis. A multi-axis motion waits in the motion queue of the leading axis. The leading axis is an axis specified first in the motion command.

The **acsc_GoM** function initiates the motions waiting in the motion queues of the specified axes. If no motion waits in one or more motion queues, the corresponding axes are not affected.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GoM
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, ACSC_AXIS_4, ACSC_AXIS_5, -1 };
if (!acsc_GoM(          Handle,         // communication handle
```

```
                        Axes,           // start up the motion of axes 0, 1,
                                        // 4 and 5
                        NULL            // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.14.3 acsc_Halt

Description

The function terminates a motion using the full deceleration profile.

Syntax

int acsc_Halt(HANDLE Handle, int Axis, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function terminates the executed motion that involves the specified axis. The terminated motion can be either single-axis or multi-axis. Any other motion waiting in the corresponding motion queue is discarded and will not be executed.

If no executed motion involves the specified axis, the function has no effect.

The terminated motion finishes using the full third-order deceleration profile and the motion deceleration value.

If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_Halt
if (!acsc_Halt(Handle,  // communication handle
                        ACSC_AXIS_0,    // terminate the motion of axis 0
                        NULL                    // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.14.4 acsc_HaltM

Description

The function terminates several motions using the full deceleration profile.

Syntax

int acsc_HaltM(HANDLE Handle, int* Axes, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains -1 which marks the end of the array. For the axis constants see Axis Definitions.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function terminates all executed motions that involve the specified axes. The terminated motions can be either single-axis or multi-axis. All other motions waiting in the corresponding motion queues are discarded and will not be executed.

If no executed motion involves a specified axis, the function has no effect on the corresponding axis.

The terminated motions finish using the full third-order deceleration profile and the motion deceleration values.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_HaltM
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, ACSC_AXIS_4, ACSC_AXIS_5, -1 };
if (!acsc_HaltM(        Handle,         // communication handle
                        Axes,           // terminate the motion of axes
                                        //0, 1, 4 and 5
                        NULL            // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.14.5 acsc_Kill

Description

The function acsc_Kill terminates a motion using reduced deceleration profile.

Syntax

int acsc_Kill(HANDLE Handle, int Axis, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function terminates the executed motion that involves the specified axis. The terminated motion can be either single-axis or multi-axis. Any other motion waiting in the corresponding motion queue is discarded and will not be executed.

If no executed motion involves the specified axis, the function has no effect.

The terminated motion finishes with the reduced second-order deceleration profile and the kill deceleration value.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_Kill
if (!acsc_Kill( Handle,         // communication handle
                ACSC_AXIS_0,    // terminate the motion of axis 0
                NULL            // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.14.6 acsc_KillAll

Description

The function terminates all currently executed motions.

Syntax

int acsc_KillAll(HANDLE Handle, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function terminates all currently executed motions. Any other motion waiting in any motion queue is discarded and will not be executed.

If no motion is currently executed, the function has no effect.

The terminated motions finish with the reduced second-order deceleration profile and the kill deceleration values.

If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_KillAll
if (!acsc_KillAll(Handle, NULL))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.14.7 acsc_KillM

Description

The functionterminates several motions using reduced deceleration profile.

Syntax

int acsc_KillM(HANDLE Handle, int* Axes, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains -1 which marks the end of the array. For the axis constants see Axis Definitions.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall unction to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function terminates all executed motions that involve the specified axes. The terminated motions can be either single-axis or multi-axis. All other motions waiting in the corresponding motion queues are discarded and will not be executed.

If no executed motion involves a specified axis, the function has no effect on the corresponding axis.

The terminated motions finish with the reduced second-order deceleration profile and the kill deceleration values.

If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_KillM
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, ACSC_AXIS_4, ACSC_AXIS_5, -1 };
if (!acsc_KillM(        Handle,         // communication handle
                        Axes,           // terminate the motion of axes
                                        // 1, 2, 4, and 5
                        NULL            // waiting call
```

```
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.14.8 acsc_KillExt

Description

The function terminates a motion using reduced deceleration profile and defines the kill reason.

Syntax

int acsc_KillExt(HANDLE Handle, int Axis, int Reason,ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc. For the axis constants see Axis Definitions.
Reason	Integer number that defines the reason of kill. The specified value is stored in the MERR variable in the controller and so modifies the state of the killed motor.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function terminates the executed motion that involves the specified axis. The terminated motion can be either single-axis or multi-axis. Any other motion waiting in the corresponding motion queue is discarded and will not be executed.

If no executed motion involves the specified axis, the function has no effect.

The terminated motion finishes with the reduced second-order deceleration profile and the kill deceleration value.

If **Reason** specifies one of the available motor termination codes, the state of the killed motor will be identical to the state of the motor killed for the corresponding fault. This provides an enhanced implementation of user-defined fault response.

If the second parameter specifies an arbitrary number, the motor state will be displayed as "Kill/disable reason: <number> - customer code. This provides ability to separate different KILL commands in the application.

If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the calling thread must not use or delete the **KillDeceleration** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
/ example of the waiting call of acsc_KillExt
#define MY_MOTOR_FAULT 10
if (!acsc_KillExt(      Handle,                 // communication handle
                        ACSC_AXIS_0,            // terminate the motion of
                                                // axis 0
                        MY_MOTOR_FAULT,         // internal customer code
                        NULL                    // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.14.9 acsc_Break

Description

The function terminates a motion immediately and provides a smooth transition to the next motion.

Syntax

int acsc_Break(HANDLE Handle, int Axis, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc. For the axis constants see Axis Definitions.

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function acsc_WaitForAsyncCall returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function terminates the executed motion that involves the specified axis only if the next motion is waiting in the corresponding motion queue. The terminated motion can be either single-axis or multi-axis.

If the motion queue contains no waiting motion, the break command is not executed immediately. The current motion continues instead until the next motion is planned to the same motion queue. Only then is the break command executed.

If no executed motion involves the specified axis, or the motion finishes before the next motion is planned, the function has no effect.

When executing the break command, the controller terminates the motion immediately without any deceleration profile. The controller builds instead a smooth third-order transition profile to the next motion.

Use caution when implementing the break command with a multi-axis motion, because the controller provides a smooth transition profile of the vector velocity. In a single-axis motion, this ensures a smooth axis velocity. However, in a multi-axis motion an axis velocity can change abruptly if the terminated and next motions are not tangent to the junction point. To avoid jerk, the terminated and next motion must be tangent or nearly tangent in the junction point.

If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the using of acsc_Break
// start up the motion of axis 0 to point 10000
acsc_ToPoint(Handle, 0, ACSC_AXIS_0, 10000, NULL);
                        // delay 200 ms
Sleep(200);             // Windows API function
```

```
acsc_Break(Handle, ACSC_AXIS_0, NULL);
                        // change the end point to point -20000 on the fly
acsc_ToPoint(Handle, 0, ACSC_AXIS_0, -20000, NULL);
```

4.14.10 acsc_BreakM

Description

The function terminates several motions immediately and provides a smooth transition to the next motions.

Syntax

int acsc_BreakM(HANDLE Handle, int* Axes, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function acsc_WaitForAsyncCall returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function terminates the executed motions that involve the specified axes. Only those motions are terminated that have the next motion waiting in the corresponding motion queue. The terminated motions can be either single-axis or multi-axis.

If a motion queue contains no waiting motion, the break command does not immediately affect the corresponding axis. The current motion continues instead until the next motion is planned to the same motion queue. Only then, the break command is executed.

If no executed motion involves the specified axis, or the corresponding motion finishes before the next motion is planned, the function does not affect the axis.

When executing the break command, the controller terminates the motion immediately without any deceleration profile. Instead, the controller builds a smooth third-order transition profile to the next motion.

Use caution when implementing the break command with a multi-axis motion, because the controller provides a smooth transition profile of the vector velocity. In a single-axis motion, this ensures a smooth axis velocity, but in a multi-axis motion, an axis velocity can change abruptly if the terminated and next motions are not tangent in the junction point. To avoid jerk, the terminated and next motion must be tangent or nearly tangent in the junction point.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_BreakM
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, -1 };
double Points[] = { 10000, 10000 };
                // start up the motion of axis XY to point (10000, 10000)
acsc_ToPointM(Handle, 0, Axes, Points, NULL);
                // delay 200 ms
Sleep(200);     // Windows API function
acsc_BreakM(Handle, Axes, NULL);
                // change the end point to point (-10000, -10000) on the fly
Points[0] = -10000; Points[1] = -10000;
acsc_ToPointM(Handle, 0, Axes, Points, NULL);
```

4.15 Point-to-Point Motion Functions

The Point-to-Point Motion functions are:

Table 5-15. Point-to-Point Motion Functions

Function	Description
acsc_ToPoint	Initiates a single-axis motion to the specified point.
acsc_ToPointM	Initiates a multi-axis motion to the specified point.
acsc_ExtToPoint	Initiates a single-axis motion to the specified point using the specified velocity or end velocity.
acsc_ExtToPointM	Initiates a multi-axis motion to the specified point using the specified velocity or end velocity.

4.15.1 acsc_ToPoint

Description

The function initiates a single-axis motion to the specified point.

Syntax

int acsc_ToPoint(HANDLE Handle, int Flags, int Axis, double Point,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Flags	Bit-mapped parameter that can include one or more of the following flags: ACSC_AMF_WAIT: plan the motion but don't start it until the function acsc_Go is executed. ACSC_AMF_RELATIVE: the **Point** value is relative to the end point of the previous motion. If the flag is not specified, the **Point** specifies an absolute coordinate.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc. For the axis constants see Axis Definitions.
Point	Coordinate of the target point.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function initiates a single-axis point-to-point motion.

The motion is executed using the required motion velocity and finishes with zero end velocity. The required motion velocity is the velocity specified by the previous call of the acsc_SetVelocity function or the default velocity if the function was not called.

To execute multi-axis point-to-point motion, use acsc_ToPointM. To execute motion with other motion velocity or non-zero end velocity, use **acsc_ToPoint** or acsc_ExtToPointM.

The controller response indicates that the command was accepted and the motion was planned successfully. The function does not wait for the motion end. To wait for the motion end, use acsc_WaitMotionEnd function.

The motion builds the velocity profile using the required values of velocity, acceleration, deceleration, and jerk of the specified axis.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_ToPoint
if (!acsc_ToPoint(Handle,       // communication handle
                0,              // start up immediately the motion
                ACSC_AXIS_0,    // of the axis 0
                10000,          // to the absolute target point 10000
                NULL            // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.15.2 acsc_ToPointM

Description

The function initiates a multi-axis motion to the specified point.

Syntax

int acsc_ToPointM(HANDLE Handle, int Flags, int* Axes, double* Point,
 ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Flags	Bit-mapped parameter that can include one or more of the following flags: ACSC_AMF_WAIT: plan the motion but don't start it until the function acsc_GoM is executed. ACSC_AMF_RELATIVE: the Point values are relative to the end point of the previous motion. If the flag is not specified, the Point specifies absolute coordinates.

	ACSC_AMF_MAXIMUM: not to use the motion parameters from the leading axis but to calculate the maximum allowed motion velocity, acceleration, deceleration, and jerk of the involved axes.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions.
Point	Array of the target coordinates. The number and order of values must correspond to the **Axes** array. The **Point** must specify a value for each element of **Axes** except the last –1 element.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function initiates a multi-axis point-to-point motion.

The motion is executed using the required motion velocity and finishes with zero end velocity. The required motion velocity is the velocity specified by the previous call of the acsc_SetVelocity function, or the default velocity if the function was not called.

To execute single-axis point-to-point motion, use acsc_ToPoint.To execute motion with other motion velocity or non-zero end velocity, use acsc_ExtToPoint or acsc_ExtToPointM.

The controller response indicates that the command was accepted and the motion was planned successfully. The function does not wait for the motion end. To wait for the motion end, use acsc_WaitMotionEnd function.

The motion builds the velocity profile using the required values of velocity, acceleration, deceleration, and jerk of the leading axis. The leading axis is the first axis in the **Axes** array.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_ToPointM
int Axes[] = {  ACSC_AXIS_0, ACSC_AXIS_1, ACSC_AXIS_2, ACSC_AXIS_3,
ACSC_AXIS_4, ACSC_AXIS_5, ACSC_AXIS_6, ACSC_AXIS_7, -1 };
double Points[] = {50000, 60000, 30000, -30000, -20000, -50000, -15000,
                        15000};
if (!acsc_ToPointM(Handle,              // communication handle
                        0,              // start up immediately the motion
                        Axes,           // of the axes 0, 1, 2, 3, 4, 5
                                        // 6 and 7
                        Points,         // to the absolutely target point
                        NULL            // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.15.3 acsc_ExtToPoint

Description

The function initiates a single-axis motion to the specified point using the specified velocity or end velocity.

Syntax

int acsc_ExtToPoint(HANDLE Handle, int Flags, int Axis, double Point, double Velocity,
double EndVelocity, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Flags	Bit-mapped parameter that can include one or more of the following flags: ACSC_AMF_WAIT: plan the motion but don't start it until the function acsc_Go is executed. ACSC_AMF_RELATIVE: the Point value is relative to the end point of the previous motion. If the flag is not specified, the Point specifies an absolute coordinate. ACSC_AMF_VELOCITY: the motion will use velocity specified by the Velocity argument instead of the default velocity. ACSC_AMF_ENDVELOCITY: the motion will come to the end point with the velocity specified by the **EndVelocity** argument.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.

Point	Coordinate of the target point.
Velocity	Motion velocity. The argument is used only if the ACSC_AMF_VELOCITY flag is specified.
EndVelocity	Velocity in the target point. The argument is used only if the ACSC_AMF_ENDVELOCITY flag is specified. Otherwise, the motion finishes with zero velocity.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function initiates a single-axis point-to-point motion.

If the ACSC_AMF_VELOCITY flag is specified, the motion is executed using the velocity specified by the **Velocity** argument. Otherwise, the required motion velocity is used. The required motion velocity is the velocity specified by the previous call of the acsc_SetVelocity function, or the default velocity if the function was not called.

If the ACSC_AMF_ENDVELOCITY flag is specified, the motion velocity at the final point is specified by the **EndVelocity** argument. Otherwise, the motion velocity at the final point is zero.

To execute a multi-axis point-to-point motion with the specified velocity or end velocity, use acsc_ExtToPointM. To execute motion with default motion velocity and zero end velocity, use **acsc_ExtToPoint** or acsc_ToPointM.

The controller response indicates that the command was accepted and the motion was planned successfully. The function does not wait for the motion end. To wait for the motion end, use acsc_WaitMotionEnd function.

The motion builds the velocity profile using the required values of acceleration, deceleration and jerk of the specified axis.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_ExtToPoint
if (!acsc_ExtToPoint(Handle,            // communication handle
                ACSC_AMF_VELOCITY |     // start up the motion with
                                        // specified velocity 5000
                ACSC_AMF_ENDVELOCITY,   // come to the end point with
                                        // specified velocity 1000
                ACSC_AXIS_0,            // axis 0
                10000,                  // target point
                5000,                   // motion velocity
                1000,                   // velocity in the target
                                        // point
                NULL                    // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.15.4 acsc_ExtToPointM

Description

The function initiates a multi-axis motion to the specified point using the specified velocity or end velocity.

Syntax

int acsc_ExtToPointM(HANDLE Handle, int Flags, int* Axes, double* Point,
double Velocity, double EndVelocity, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Flags	Bit-mapped parameter that can include one or more of the following flags: ACSC_AMF_WAIT: plan the motion but don't start it until the function acsc_Go is executed. ACSC_AMF_RELATIVE: the Point values are relative to the end of the previous motion. If the flag is not specified, the Point specifies absolute coordinates. ACSC_AMF_VELOCITY: the motion will use velocity specified by the Velocity argument instead of the default velocity. ACSC_AMF_ENDVELOCITY: the motion will come to the end with the velocity specified by the **EndVelocity** argument. ACSC_AMF_MAXIMUM: not to use the motion parameters from the leading axis but to calculate the maximum allowed motion velocity, acceleration, deceleration and jerk of the involved axes.

Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions.
Point	Array of the target coordinates. The number and order of values must correspond to the **Axes** array. The **Point** must specify a value for each element of **Axes** except the last –1 element.
Velocity	Motion vector velocity. The argument is used only if the ACSC_AMF_VELOCITY flag is specified.
EndVelocity	Vector velocity in the target point. The argument is used only if the ACSC_AMF_ENDVELOCITY is specified. Otherwise, the motion finishes with zero velocity.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function initiates a multi-axis point-to-point motion.

If the ACSC_AMF_VELOCITY flag is specified, the motion is executed using the velocity specified by the **Velocity** argument. Otherwise, the required motion velocity is used. The required motion velocity is the velocity specified by the previous call of the acsc_SetVelocity function, or the default velocity if the function was not called.

If the ACSC_AMF_ENDVELOCITY flag is specified, the motion velocity at the final point is specified by the **EndVelocity** argument. Otherwise, the motion velocity at the final point is zero.

To execute a single-axis point-to-point motion with the specified velocity or end velocity, use acsc_ExtToPoint. To execute a motion with default motion velocity and zero end velocity, use acsc_ToPoint or acsc_ToPointM.

The controller response indicates that the command was accepted and the motion was planned successfully. The function does not wait for the motion end. To wait for the motion end, use acsc_WaitMotionEnd function.

The motion builds the velocity profile using the required values of acceleration, deceleration and jerk of the leading axis. The leading axis is the first axis in the **Axes** array.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_ExtToPointM
int Axes[] = {  ACSC_AXIS_0, ACSC_AXIS_1, ACSC_AXIS_2, ACSC_AXIS_3,
                        ACSC_AXIS_4, ACSC_AXIS_5, ACSC_AXIS_6,
                        ACSC_AXIS_7, -1
                };
double Points[] =       { 50000, 60000, 30000, 20000, -20000, -50000,
                        -15000, 15000 };
if (!acsc_ExtToPointM(  Handle,         // communication handle
                ACSC_AMF_VELOCITY |     // start up the motion with
                                        // specified velocity 5000
                                        // and come to the end point
                ACSC_AMF_ENDVELOCITY,   // with specified velocity
                                        // 1000
                Axes,                   // axes 0, 1, 2, 3, 4, 5, 6 and 7
                Points,                 // target point
                5000,                   // motion velocity
                1000,                   // velocity in the target
                                        // point
                NULL                    // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.16 Augmented Point-to-Point Motion

From version 3.12 the C Library supports point-to-point motion with various augmentations.

Function	Description
acsc_BoostedPointToPointMotion	This function defines a motion profile using the ***MotionBoost*** Feature
acsc_SmoothPointToPointMotion	Defines a multi-axes Point-to-Point motion and provides the positioning to specific target and looks like acsc_ToPointM command. Unlike **acsc_ToPointM**, **acsc_SmoothPointToPointMotion** uses a 4th order motion profile.

4.16.1 acsc_BoostedPointToPointMotion

Description

This function defines a motion profile using the ***MotionBoost*** Feature.

Syntax

```
int acsc_BoostedPointToPointMotion(HANDLE Handle, int Flags, int* Axes
,double* point, double Velocity, double FinalVelocity, double Time,
double FixedTime ,double MotionDelay, ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Flags	Bit-mapped argument that can include one or more of the following flags: ACSC_AMF_WAIT(/w) plan the motion but do not start it until the function acsc_GoM is executed. ACSC_AMF_VELOCITY(/v) the motion will use velocity specified for each segment instead of the default velocity. ACSC_AMF_EXT_DELAY_MOTION(/q) Defines actual motor movement delay in microseconds. The delay resolution is 50 microseconds. The switch requires an additional parameter that specifies the motion delay. ACSC_AMF_TIME (/t) Minimum travel time in seconds, The calculated travel time will be at least the specified value. Incompatible with the ACSC_AMF_FIXED_TIME switch. ACSC_AMF_FIXED_TIME (/d) Travel Time – specifies the exact travel time for the motion in seconds. All other considerations are ignored, which could cause a safety fault during motion execution. Incompatible with the ACSC_AMF_TIME switch. ACSC_AMF_ENDVELOCITY The motion comes to the end point with the specified velocity User will enter final, nonzero velocity ACSC_AMF_RELATIVE

	Relative motion, The value of the point coordinate is relative to the end point coordinate of the previous motion. ACSC_AMF_LOCALCS Interpret entered coordinates according to the Local Coordinate System. With this switch, use 2 axes motion coordinate only (X,Y). Using 3 or more coordinates causes a runtime error. ACSC_AMF_MAXIMUM Use maximum velocity under axis limits. With this suffix, no required velocity should be specified. The required velocity is calculated for each segment individually on the base of segment geometry and axis velocities (VEL values) of the involved axes. ACSC_AMF_2 Use 20 kHz motion mode Limited to at most 2 axes in a single function call.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains – 1 which marks the end of the array. For the axis constants see Axis Definitions.
Velocity	If ACSC_AMF_VELOCITY flag was specified, this argument specifies a motion velocity for current segment. Set this argument to ACSC_NONE if not used.
FinalVelocity	If ACSC_AMF_ENDVELOCITY flag was specified, this argument defines FinalVelocity . FinalVelocity specified the motion velocity at the final point Set this argument to ACSC_NONE if not used.
Time	If ACSC_AMF_TIME flag was specified, this argument defines Time. Minimum travel time in seconds, The calculated travel time will be at least the specified value. Set this argument to ACSC_NONE if not used.
FixedTime	If ACSC_AMF_FIXED_TIME flag has been specified, this argument defines the FixedTime . specifies the exact travel time for the motion in seconds. Set this argument to ACSC_NONE if not used
MotionDelay	If ACSC_AMF_DELAY_MOTION flag has been specified, this argument

	defines the bit motion delay. Defines actual motor movement delay in microseconds. The delay resolution is 50 microseconds. The maximum delay is 100 controller cycles or 100ms for CTIME=1ms or 20ms for CTIME=0.2ms. Set this argument to ACSC_NONE if not used
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

Supported from V3.12.

Example

```
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, -1 };
double Point[2];
Point[0] = 1000; Point[1] = 1000;

if (!acsc_BoostedPointToPointMotion(Handle, ACSC_AMF_VELOCITY, Axes,
    Point, 1000, ACSC_NONE,  ACSC_NONE, ACSC_NONE, ACSC_NONE,
    ACSC_SYNCHRONOUS)) {
        printf("transaction error: %d\n", acsc_GetLastError());
    }

acsc_EndSequenceM( Handle, Axes, 0);
```

4.16.2 acsc_SmoothPointToPointMotion

Description

The **acsc_SmoothPointToPointMotion** command defines a multi-axes Point-to-Point motion and provides the positioning to specific target and looks like the **acsc_ToPointM command**. Unlike **acsc_ ToPointM**, **acsc_SmoothPointToPointMotion** uses a 4th order motion profile.

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.

The **acsc_SmoothPointToPointMotion** command can be either a single axis or an axis group.

acsc_SmoothPointToPointMotion responds to **KILL /HALT** commands in the same fashion as do other motion commands.

Syntax

```
int acsc_SmoothPointToPointMotion (HANDLE Handle, int Flags , int* Axes,
double* point, double Velocity,ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Flags	Bit-mapped argument that can include one or more of the following flags: ACSC_AMF_VELOCITY The motion will use velocity specified for each segment instead of the default velocity. ACSC_AMF_LOCALCS Interpret entered coordinates according to the Local Coordinate System. With this switch, use 2 axes motion coordinate only (X,Y). Using 3 or more coordinates causes a runtime error. ACSC_AMF_WAIT plan the motion but do not start it until the function acsc_GoM is executed. ACSC_AMF_RELATIVE Relative motion, The value of the point coordinate is relative to the end point coordinate of the previous motion. ACSC_AMF_ENVELOPE Wait for motion termination before executing next command. ACSC_AMF_DELAY_MOTION Defines actual motor movement delay in microseconds. The delay resolution is 50 microseconds. The switch requires an additional parameter that specifies the motion delay.
Axes	PEG axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. 2 or 3 Cartesian axes can participate in the motion. For the axis constants see Axis Definitions.

Point	Array of the final point coordinates. The number and order of values must correspond to the Axes array. The Point must specify a value for each element of Axes except the last –1 element.
Velocity	If ACSC_AMF_VELOCITY flag was specified, this argument specifies a motion velocity for current segment. Set this argument to ACSC_NONE if not used.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero

If the function fails, the return value is zero.

Extended error information can be obtained by calling acsc_GetLastError.

Example

```
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, -1 };
    double Point[2] ;
    Point[0] = 1000; Point[1] = 1000;

if (!acsc_SmoothPointToPointMotion(Handle, ACSC_AMF_ENVELOPE, Axes,
Point, ACSC_NONE, ACSC_SYNCHRONOUS)) {
    printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.17 Track Motion Control Functions

The Track Motion Control functions are:

Table 5-16. Track Motion Control Functions

Function	Description
acsc_Track	The function initiates a single-axis track motion.
acsc_SetTargetPosition	The function assigns a current value of target position.
acsc_GetTargetPosition	The function receives the current value of target position.

4.17.1 acsc_Track

Description

The function initiates a single-axis track motion.

Syntax

int acsc_Track(HANDLE Handle, int Flags, int Axis, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Flags	Bit-mapped parameter that can include the following flag: ACSC_AMF_WAIT: plan the motion but don't start it until the function acsc_Go is executed.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc. For the axis constants see Axis Definitions.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function initiates a single-axis track motion. After the motion is initialized, ptp motion will be generated with every change in TPOS value.

The controller response indicates that the command was accepted and the motion was planned successfully.

Example

```
// example of the waiting call of acsc_Track
if (!acsc_Track(Handle,         // communication handle
                0,              // start up immediately the motion
                ACSC_AXIS_0,    // of the axis 0
                NULL            // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.17.2 acsc_SetTargetPosition

Description

The function assigns a current value of track position.

Syntax

int acsc_SetTargetPosition(HANDLE Handle, int Axis, double TargetPosition,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc. For the axis constants see Axis Definitions.
TargetPosition	The value specifies the current value of track position.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function assigns a current value to the Track position. If the corresponding axis is initialized with track motion, the change of TPOS will cause generation of ptp motion to that new value.

For more information see the explanation of the track command in the *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SetFPosition
if (!acsc_SetTargetPosition(Handle,// communication handle
                ACSC_AXIS_0,    // axis 0
                0,              // required target position
                NULL            // waiting call
                ))
{
        printf("acsc_SetTargetPosition error: %d\n", acsc_GetLastError());
}
```

4.17.3 acsc_GetTargetPosition

Description

The function retrieves the instant value of track position.

Syntax

int acsc_GetTargetPosition(HANDLE Handle, int Axis, double *TargetPosition,
 ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc. For the axis constants see Axis Definitions.
TargetPosition	The pointer to variable that receives the instant value of the target position.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function reads a current value of the corresponding TPOS variable. If the corresponding axis is initialized with track motion, TPOS controls the motion of the axis.

For more information see the explanation of the track command in the *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SetFPosition
double TPOS;
if (!acsc_SetTargetPosition(    Handle, // communication handle
                        ACSC_AXIS_0,    // axis 0
                        &TPOS           // target position
                        NULL            // waiting call
                        ))
{
        printf("acsc_GetTargetPosition error: %d\n", acsc_GetLastError());
}
```

4.18 Jog Functions

The Jog functions are:

Table 5-17. Jog Functions

Function	Description
acsc_Jog	Initiates a single-axis jog motion.
acsc_JogM	Initiates a multi-axis jog motion.

4.18.1 acsc_Jog

Description

The function initiates a single-axis jog motion.

Syntax

int acsc_Jog(HANDLE Handle, int Flags, int Axis, double Velocity,
 ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Flags	Bit-mapped parameter that can include one or more of the following flags: ACSC_AMF_WAIT: plan the motion but don't start it until the function acsc_Go is executed. ACSC_AMF_VELOCITY: the motion will use the velocity specified by the Velocity argument instead of the default velocity.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc. For the axis constants see Axis Definitions.
Velocity	If the ACSC_AMF_VELOCITY flag is specified, the velocity profile is built using the value of **Velocity**. The sign of **Velocity** defines the direction of the motion. If the ACSC_AMF_VELOCITY flag is not specified, only the sign of **Velocity** is used in order to specify the direction of motion. In this case, the constants ACSC_POSITIVE_DIRECTION or ACSC_NEGATIVE_DIRECTION can be used.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function initiates a single-axis jog. To execute multi-axis jog, use acsc_JogM.

The jog motion is a motion with constant velocity and no defined ending point. The jog motion continues until the next motion is planned, or the motion is killed for any reason.

The motion builds the velocity profile using the default values of acceleration, deceleration and jerk of the specified axis. If the ACSC_AMF_VELOCITY flag is not specified, the default value of velocity is used as well. In this case, only the sign of **Velocity** is used in order to specify the direction of motion. The positive velocity defines a positive direction, the negative velocity – negative direction.

If the ACSC_AMF_VELOCITY flag is specified, the value of **Velocity** is used instead of the default velocity. The sign of **Velocity** defines the direction of the motion.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the motion was planned successfully. No waiting for the motion end is provided.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_Jog
if (!acsc_Jog(Handle,                    // communication handle
                0,                       // start up immediately the jog
                                         // motion
                                         // with default velocity
                ACSC_AXIS_0,             // axis 0
                ACSC_NEGATIVE_DIRECTION,// to the negative direction
                NULL                     // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.18.2 acsc_JogM

Description

The function initiates a multi-axis jog motion.

Syntax

int acsc_JogM(HANDLE Handle, int Flags, int* Axes, int* Direction, double Velocity, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Flags	Bit-mapped parameter that can include one or more of the following flags: ACSC_AMF_WAIT: plan the motion but don't start it until the function acsc_Go is executed. ACSC_AMF_VELOCITY: the motion will use the velocity specified by the Velocity argument instead of the default velocity.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions..

Direction	Array of directions. The number and order of values must correspond to the **Axes** array. The **Direction** array must specify direction for each element of **Axes** except the last –1 element. The constant ACSC_POSITIVE_DIRECTION in the **Direction** array specifies the correspondent axis to move in positive direction, the constant ACSC_NEGATIVE_DIRECTION specifies the correspondent axis to move in the negative direction.
Velocity	If the ACSC_AMF_VELOCITY flag is specified, the velocity profile is built using the value of **Velocity**. If the ACSC_AMF_VELOCITY flag is not specified, **Velocity** is not used.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCallfunction to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function initiates a multi-axis jog motion. To execute single-axis jog motion, useacsc_Jog.

The jog motion is a motion with constant velocity and no defined ending point. The jog motion continues until the next motion is planned, or the motion is killed for any reason.

The motion builds the vector velocity profile using the default values of velocity, acceleration, deceleration and jerk of the axis group. If the ACSC_AMF_VELOCITY flag is not specified, the default value of velocity is used as well. If the ACSC_AMF_VELOCITY flag is specified, the value of **Velocity** is used instead of the default velocity.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the motion was planned successfully. The function cannot wait or validate the end of the motion. To wait for the motion end, use the acsc_WaitMotionEnd function.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_JogM
int Axes[] = {                  ACSC_AXIS_0, ACSC_AXIS_1, ACSC_AXIS_2,
                                ACSC_AXIS_3,
                                ACSC_AXIS_4, ACSC_AXIS_5, ACSC_AXIS_6,
                                ACSC_AXIS_7, -1 };
int Directions[] = {    ACSC_POSITIVE_DIRECTION, ACSC_POSITIVE_DIRECTION,
                                ACSC_POSITIVE_DIRECTION,
                                ACSC_POSITIVE_DIRECTION,
                                ACSC_NEGATIVE_DIRECTION,
                                ACSC_NEGATIVE_DIRECTION,
                                ACSC_NEGATIVE_DIRECTION,
                                ACSC_NEGATIVE_DIRECTION
                };
if (!acsc_JogM( Handle,         // communication handle
                0,              // start up immediately the jog motion
                                // with the specified velocity 5000
                Axes,           // axes 0, 1, 2, 3, 4, 5, 6 and 7
                Directions,     // axes 0, 1, 2, and 3 in the positive
                                // direction, and axes 4, 5, 6 and 7
                                // in the negative direction
                5000,           // motion velocity
                NULL            // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.19 Slaved Motion Functions

The Slaved Motion functions are:

Table 5-18. Slaved Motion Functions

Function	Description
acsc_SetMaster	Initiates calculation of a master value for an axis.
acsc_Slave	Initiates a master-slave motion.
acsc_SlaveStalled	Initiates master-slave motion with limited follow-on area.

4.19.1 acsc_SetMaster

Description

The function initiates calculating of master value for an axis.

Syntax

int acsc_SetMaster(HANDLE Handle, int Axis, char* Formula,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc. For the axis constants see Axis Definitions.
Formula	ASCII string that specifies a rule for calculating master value.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function initiates calculating of master value for an axis.

The master value for each axis is presented in the controller as one element of the **MPOS** array. Once the **acsc_SetMaster** function is called, the controller is calculates the master value for the specified axis each controller cycle.

The **acsc_SetMaster** function can be called again for the same axis at any time. At that moment, the controller discards the previous formula and accepts the newly specified formula for the master calculation.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the controller starts calculating the master value according to the formula.

The **Formula** string can specify any valid ACSPL+ expression that uses any standard or user global variables as its operands.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SetMaster
char* szFormula = "2 * FPOS(1)";// master value is calculated as
                                // feedback position of axis 1
                                // with scale factor equal 2
if (!acsc_SetMaster(    Handle, // communication handle
                ACSC_AXIS_0,    // set master value for the axis 0
                szFormula,      // ASCII string that specifies a rule for
                                // calculating master value
                NULL            // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.19.2 acsc_Slave

Description

The function initiates a master-slave motion.

Syntax

int acsc_Slave(HANDLE Handle, int Flags, int Axis, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Flags	Bit-mapped parameter that can include one or more of the following flags: ACSC_AMF_WAIT: plan the motion but don't start it until the function acsc_Go is executed. ACSC_AMF_POSITIONLOCK: the motion will use position lock. If the flag is not specified, velocity lock is used.
Axis	Slaved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc. For the axis constants see Axis Definitions.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function initiates a single-axis master-slave motion with an unlimited area of following. If the area of following must be limited, use acsc_SlaveStalled.

The master-slave motion continues until the motion is killed or the motion fails for any reason.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the motion was planned successfully.

The master value for the specified axis must be defined before by the call to acsc_SetMaster function. The **acsc_SetMaster** function can be called again in order to change the formula of master calculation. If at this moment the master-slave motion is in progress, the slave can come out from synchronism. The controller then regains synchronism, probably with a different value of offset between the master and slave.

If the ACSC_AMF_POSITIONLOCK flag is not specified, the function activates a velocity-lock mode of slaved motion. When synchronized, the **APOS** axis reference follows the **MPOS** with a constant offset:

APOS = MPOS + C

The value of **C** is latched at the moment when the motion comes to synchronism, and then remains unchanged as long as the motion is synchronous. If at the moment of motion start the master velocity is zero, the motion starts synchronously and **C** is equal to the difference between initial values of **APOS** and **MPOS**.

If the ACSC_AMF_POSITIONLOCK flag is specified, the function activates a position-lock mode of slaved motion. When synchronized, the **APOS** axis reference strictly follows the **MPOS**:

APOS = MPOS

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_Slave
char* szFormula = "2 * FPOS(1)";// master value is calculated as feedback
                                // position of the axis 1 with scale
                                // factor equal 2
acsc_SetMaster(Handle, ACSC_AXIS_0, szFormula, NULL));
if (!acsc_Slave(        Handle,         // communication handle
                        0,              // velocity lock is used as default
                        ACSC_AXIS_0,    // axis 0
                        NULL            // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.19.3 acsc_SlaveStalled

Description

The function initiates master-slave motion within predefined limits.

Syntax

int acsc_SlaveStalled(HANDLE Handle, int Flags, int Axis, double Left, double Right, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Flags	Bit-mapped parameter that can include one or more of the following flags: ACSC_AMF_WAIT: plan the motion but don't start it until the acsc_Go function is executed. ACSC_AMF_POSITIONLOCK: the motion will use position lock. If the flag is not specified, velocity lock is used.
Axis	Slaved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc. For the axis constants see Axis Definitions.
Left	Left (negative) limit of the following area.
Right	Right (positive) limit of the following area.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result.

	If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function initiates single-axis master-slave motion within predefined limits. Use acsc_Slave to initiate unlimited motion. For sophisticated forms of master-slave motion, use slaved variants of segmented and spline motions.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the motion was planned successfully.

The master-slave motion continues until the **kill** command is executed, or the motion fails for any reason.

The master value for the specified axis must be defined before by the call to acsc_SetMaster function. The **acsc_SetMaster** function can be called again in order to change the formula of master calculation. If at this moment the master-slave motion is in progress, the slave can come out from synchronism. The controller then regains synchronism, probably with a different value of offset between the master and slave.

If the ACSC_AMF_POSITIONLOCK flag is not specified, the function activates a velocity-lock mode of slaved motion. When synchronized, the **APOS** axis reference follows the **MPOS** with a constant offset:

APOS = MPOS + C

The value of **C** is latched at the moment when the motion comes to synchronism, and then remains unchanged as long as the motion is synchronous. If at the moment of motion start the master velocity is zero, the motion starts synchronously and **C** is equal to the difference between initial values of **APOS** and **MPOS**.

If the ACSC_AMF_POSITIONLOCK flag is specified, the function activates a position-lock mode of slaved motion. When synchronized, the **APOS** axis reference strictly follows the **MPOS**:

APOS = MPOS

The **Left** and **Right** values define the allowed area of changing the **APOS** value. The **MPOS** value is not limited and can exceed the limits. In this case, the motion comes out from synchronism, and the **APOS** value remains (stalls) in one of the limits until the change of **MPOS** allows following again.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SlaveStalled
char* szFormula = "2 * FPOS(1)";// master  value is  calculated as
                                // feedback position of the axis 1 with
                                // scale factor equal 2
acsc_SetMaster(Handle, ACSC_AXIS_0, szFormula, NULL));
if (!acsc_SlaveStalled( Handle, // communication handle
                 0,             // velocity lock is used as default
                ACSC_AXIS_0,    // axis 0
                -100000,        // left (negative) limit of the
                                // following area
                100000,         // right (positive) limit of the
                                // following area
                NULL            // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.20 Multi-Point Motion Functions

The Multi-Point Motion functions are:

Table 5-19. Multi-Point Motion Functions

Function	Description
acsc_MultiPoint	Initiates a single-axis multi-point motion.
acsc_MultiPointM	Initiates a multi-axis multi-point motion.

4.20.1 acsc_MultiPoint

Description

The function initiates a single-axis multi-point motion.

Syntax

int acsc_MultiPoint(HANDLE Handle, int Flags, int Axis, double Dwell,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Flags	Bit-mapped parameter that can include one or more of the following flags: ACSC_AMF_WAIT: plan the motion but don't start it until the function acsc_Go is executed.

	ACSC_AMF_RELATIVE: the coordinates of each point are relative. The first point is relative to the instant position when the motion starts; the second point is relative to the first, etc. If the flag is not specified, the coordinates of each point are absolute. ACSC_AMF_VELOCITY: the motion uses the velocity specified with each point instead of the default velocity. ACSC_AMF_CYCLIC: the motion uses the point sequence as a cyclic array. After positioning to the last point it does positioning to the first point and continues.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc. For the axis constants see Axis Definitions.
Dwell	Dwell in each point in milliseconds.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function initiates a single-axis multi-point motion. To execute multi-axis multi-point motion, use acsc_MultiPointM.

The motion executes sequential positioning to each of the specified points, optionally with dwell in each point.

The function itself does not specify any point, so that the created motion starts only after the first point is specified. The points of motion are specified by using the acsc_AddPoint or acsc_ExtAddPoint functions that follow this function.

The motion finishes when the acsc_EndSequence function is executed. If the call of **acsc_EndSequence** 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 motion queue will occur until the function **acsc_EndSequence** executes.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the motion was planned successfully. The function cannot wait or validate the end of the motion. To wait for the motion end, use the acsc_WaitMotionEnd function.

During positioning to each point, a velocity profile is built using the default values of acceleration, deceleration, and jerk of the specified axis. If the ACSC_AMF_VELOCITY flag is not specified, the default value of velocity is used as well. If the ACSC_AMF_VELOCITY flag is specified, the value of velocity specified in subsequent acsc_ExtAddPoint functions is used.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_MultiPoint
if (!acsc_MultiPoint(   Handle,         // communication handle
                        0,              // create the multi-point motion
                                        // with default velocity
                        ACSC_AXIS_0,    // axis 0
                        1,              // with dwell 1 ms
                        NULL            // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
                                        // add some points
acsc_AddPoint(Handle, ACSC_AXIS_0, 1000, NULL); // from the point 1000
acsc_AddPoint(Handle, ACSC_AXIS_0, 2000, NULL); // to the point 2000
                                        // finish the motion
acsc_EndSequence(Handle, ACSC_AXIS_0, NULL);// end of multi-point motion
```

4.20.2 acsc_MultiPointM

Description

The function initiates a multi-axis multi-point motion.

Syntax

int acsc_MultiPointM(HANDLE Handle, int Flags, int* Axes, double Dwell,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Flags	Bit-mapped parameter that can include one or more of the following flags: ACSC_AMF_WAIT: plan the motion but don't start it until the function acsc_GoMis executed.

	ACSC_AMF_RELATIVE: the coordinates of each point are relative. The first point is relative to the instant position when the motion starts; the second point is relative to the first, etc. If the flag is not specified, the coordinates of each point are absolute. ACSC_AMF_VELOCITY: the motion will use the velocity specified with each point instead of the default velocity. ACSC_AMF_CYCLIC: the motion uses the point sequence as a cyclic array: after positioning to the last point does positioning to the first point and continues.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions..
Dwell	Dwell in each point in milliseconds.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function initiates a multi-axis multi-point motion. To execute single-axis multi-point motion, use acsc_MultiPoint.

The motion executes sequential positioning to each of the specified points, optionally with dwell in each point.

The function 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 using acsc_AddPointM or acsc_ExtAddPointM, functions that follow this function.

The motion finishes when the acsc_EndSequenceM function is executed. If the call of **acsc_EndSequenceM** 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 motion queue will occur until the function **acsc_EndSequenceM** executes.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the motion was planned successfully. The function cannot wait or validate the end of the motion. To wait for the motion end, use acsc_WaitMotionEnd function.

During positioning to each point, a vector velocity profile is built using the default values of velocity, acceleration, deceleration, and jerk of the axis group. If the AFM_VELOCITY flag is not specified, the default value of velocity is used as well. If the AFM_VELOCITY flag is specified, the value of velocity specified in subsequent acsc_ExtAddPointM functions is used.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_MultiPointM
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, -1 };
int Points[2];
if (!acsc_MultiPointM(  Handle, // communication handle
                0,              // create the multi-point motion with
                                // default velocity
                Axes,           // of the axes 0 and 1
                0,              // without dwell in the points
                NULL            // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
                                // add some points
Points[0] = 1000; Points[1] = 1000;
acsc_AddPointM(Handle, Axes, Points, NULL);// from the point 1000, 1000
Points[0] = 2000; Points[1] = 2000;
acsc_AddPointM(Handle, Axes, Points, NULL);// to the point 2000, 2000
                                        // finish the motion
acsc_EndSequenceM(Handle, Axes, NULL);  // the end of the multi-point
motion
```

4.21 Arbitrary Path Motion Functions

The Arbitrary Path Motion functions are:

Table 5-20. Arbitrary Path Motion Functions

Function	Description
acsc_Spline	Initiates a single-axis spline motion. The motion follows an arbitrary path defined by a set of points.
acsc_ SplineM	Initiates a multi-axis spline motion. The motion follows an arbitrary path defined by a set of points.

4.21.1 acsc_Spline

Description

The function initiates a single-axis spline motion. The motion follows an arbitrary path defined by a set of points.

Syntax

int acsc_Spline(HANDLE Handle, int Flags, int Axis, double Period,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Flags	Bit-mapped parameter that can include one or more of the following flags: ACSC_AMF_WAIT: plan the motion but don't start it until the acsc_Go function is executed. ACSC_AMF_RELATIVE: use the coordinates of each point as relative. The first point is relative to the instant position when the motion starts; the second point is relative to the first, etc. If the flag is not specified, the coordinates of each point are absolute. ACSC_AMF_VARTIME: the time interval between adjacent points is non-uniform and is specified along with each added point. If the flag is not specified, the interval is uniform and is specified in the Period argument. ACSC_AMF_CYCLIC: use the point sequence as a cyclic array: after the last point come to the first point and continue. ACSC_AMF_CUBIC: use a cubic interpolation between the specified points (third-order spline). If the flag is not specified, linear interpolation is used (first-order spline). If the flag is specified and the ACSC_AMF_VARTIME is not specified, the controller builds PV spline motion. If the flag is specified and the ACSC_AMF_VARTIME is specified, the controller builds PVT spline motion.

Axis	Axis constant: **ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc**. For the axis constants see Axis Definitions.
Period	Time interval between adjacent points. The parameter is used only if the ACSC_AMF_VARTIME flag is not specified.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function initiates a single-axis spline motion. To execute multi-axis spline motion, use acsc_SplineM.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the motion was planned successfully. The function cannot wait or validate the end of the motion. To wait for the motion end, use the acsc_WaitMotionEnd function.

The motion does not use the default values of velocity, acceleration, deceleration, and jerk. The points and the time intervals between the points completely define the motion profile.

Points for arbitrary path motion are defined by the consequent calls of acsc_AddPoint or acsc_ExtAddPoint functions. The acsc_EndSequence function terminates the point sequence. After execution of the **acsc_EndSequence** function, no **acsc_AddPoint** or **acsc_ExtAddPoint** functions 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 ACSC_AMF_VARTIME flag.

This motion does not use a motion profile generation. The time intervals between the points are typically short, so that the array of the points implicitly specifies the desired velocity in each point.

If the time interval does not coincide with the controller cycle, the controller provides interpolation of the points according to the ACSC_AMF_CUBIC flag.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_Spline
int i;
if (!acsc_Spline(       Handle,          // communication handle
                        0,               // create the arbitrary path motion
                                         // with uniform interval 10 ms
                        ACSC_AXIS_0 ,
                        10,              // uniform interval 10 ms
                NULL // waiting call
                ))
{
printf("transaction error: %d\n", acsc_GetLastError());
}
// add some points
for (i = 0; i <100; i++)
{
do
{
if(!acsc_AddPoint (Handle, ACSC_AXIS_0, i*100, NULL))
        ErrNum=acsc_GetLastError();
else
        ErrNum=0;
}while(ErrNum==3065);
}
// finish the motion
acsc_EndSequence(Handle, ACSC_AXIS_0, NULL); // the end of arbitrary path
```

4.21.2 acsc_SplineM

Description

The function initiates a multi-axis spline motion. The motion follows an arbitrary path defined by a set of points.

Syntax

int acsc_SplineM(HANDLE Handle, int Flags, int* Axes, double Period,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Flags	Bit-mapped parameter that can include one or more of the following flags: ACSC_AMF_WAIT: plan the motion but don't start it until the acsc_GoM function is executed.

	ACSC_AMF_RELATIVE: the coordinates of each point are relative. The first point is relative to the instant position when the motion starts; the second point is relative to the first, etc. If the flag is not specified, the coordinates of each point are absolute. ACSC_AMF_VARTIME: the time interval between adjacent points is non-uniform and is specified along with each added point. If the flag is not specified, the interval is uniform and is specified in the Period argument. ACSC_AMF_CYCLIC: the motion uses the point sequence as a cyclic array: after the last point the motion comes to the first point and continues. ACSC_AMF_CUBIC: use a cubic interpolation between the specified points (third-order spline). If the flag is not specified, linear interpolation is used (first-order spline).
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions..
Period	Time interval between adjacent points. The parameter is used only if the ACSC_AMF_VARTIME flag is not specified.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function initiates a multi-axis spline motion. To execute a single-axis spline motion, use acsc_Spline.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the motion was planned successfully. The function cannot wait or validate the end of the motion. To wait for the motion end, use the acsc_WaitMotionEnd function.

The motion does not use the default values of velocity, acceleration, deceleration, and jerk. The points and the time intervals between the points define the motion profile completely.

Points for arbitrary path motion are defined by the consequent calls of the acsc_AddPointM or acsc_ExtAddPointM functions. The acsc_EndSequenceM function terminates the point sequence. After execution of the **acsc_EndSequenceM** function, no **acsc_AddPointM** or **acsc_ExtAddPointM** functions 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 ACSC_AMF_VARTIME flag.

This motion does not use motion profile generation. Typically, the time intervals between the points are short, so that the array of the points implicitly specifies the desired velocity in each point.

If the time interval does not coincide with the controller cycle, the controller provides interpolation of the points according to the ACSC_AMF_CUBIC flag.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
/ example of the waiting call of acsc_SplineM
int i;
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, -1 };
int Points[2];
if (!acsc_SplineM( Handle,      // communication handle
               0,               // create the arbitrary path motion
                                // with uniform interval 10 ms
                Axes,           // axes XY
                10,             // uniform interval 10 ms
                NULL            // waiting call
                      ))
{
printf("transaction error: %d\n", acsc_GetLastError());
}
// add some points
for (i = 0; i <100; i++)
{
Points[0] = i * 100; Points[1] = i * 50;
do
{
if(!acsc_AddPointM(Handle, Axes, Points, NULL))
        ErrNum=acsc_GetLastError();
else
        ErrNum=0;
}while(ErrNum==3065);
}
// finish the motion
acsc_EndSequenceM(Handle, Axes, NULL); // the end of arbitrary path
motion
```

4.22 PVT Functions

The PVT functions are:

Table 5-21. PVT Functions

Function	Description
acsc_AddPVPoint	Adds a point to a single-axis multi-point or spline motion.
acsc_AddPVPointM	Adds a point to a multi-axis multi-point or spline motion.
acsc_AddPVTPoint	Adds a point to a single-axis multi-point or spline motion.
acsc_AddPVTPointM	Adds a point to a multi-axis multi-point or spline motion.

4.22.1 acsc_AddPVPoint

Description

The function adds a point to a single-axis PV spline motion and specifies velocity.

Syntax

int acsc_AddPVPoint(HANDLE Handle, int Axis, double Point, double Velocity,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axes	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc. For the axis constants see Axis Definitions.
Point	Coordinate of the added point.
Velocity	Desired velocity at the point
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

Before this function can be used, PV spline motion must be initiated by calling acsc_Spline with the appropriate flags.

The function adds a point to a single-axis PV spline motion with a uniform time and specified velocity at that point

To add a point to a multi-axis PV motion, use acsc_AddPVPointM. To add a point to a PVT motion with non-uniform time interval, use the acsc_AddPVTPoint and acsc_AddPVTPointM functions. The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the point is added to the motion buffer. The point can be rejected if the motion buffer is full. In this case, you can call this function periodically until the function returns non-zero value.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
int i;
if (!acsc_Spline(Handle,                    // communication handle
                 ACSC_AMF_CUBIC,            //PV motion uniform time inteval
                 ACSC_AXIS_0,               // axis 0
                 10,                        // uniform interval 10 ms
                 NULL                       // waiting call
                 ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
// add some points
for (i = 0; i <100; i++)
acsc_AddPVPoint(Handle,ACSC_AXIS_0,i*100,i*100,NULL);
        //position,velocity and time interval for each point
// the end of the arbitrary path motion
acsc_EndSequence(Handle, ACSC_AXIS_0, NULL);
```

4.22.2 acsc_AddPVPointM

Description

The function adds a point to a multiple-axis PV spline motion and specifies velocity.

Syntax

int acsc_AddPVPointM(HANDLE Handle, int *Axis, double *Point, double *Velocity,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions..
Point	Array of the coordinates of added point. The number and order of values must correspond to the **Axes** array. The **Point** must specify a value for each element of **Axes** except the last –1 element.
Velocity	Array of the velocities of added point. The number and order of values must correspond to the **Axes** array. The **Velocity** must specify a value for each element of **Axes** except the last –1 element.

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

Before this function can be used, PVT spline motion must be initiated by calling acsc_SplineM with the appropriate flags.

The function adds a point to a multiple-axis PV spline motion with a uniform time and specified velocity at that point.

To add a point to a single-axis PV motion, use acsc_AddPVPoint. To add a point to a PVT motion with non-uniform time interval, use the acsc_AddPVTPoint andacsc_AddPVTPointM functions.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the point is added to the motion buffer. The point can be rejected if the motion buffer is full. In this case, you can call this function periodically until the function returns non-zero value.

All axes specified in the **Axes** array must be specified before the call of the acsc_MultiPointM or acsc_SplineMfunction. The number and order of the axes in the **Axes** array must correspond exactly to the number and order of the axes of acsc_MultiPointM or acsc_SplineMfunctions.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
int i;
int Axis[]={ACSC_AXIS_0, ACSC_AXIS_1, ACSC_AXIS_4,-1};
double Point[3];
double Velocity[3];
if (!acsc_SplineM(      Handle,                 // communication handle
                        ACSC_AMF_CUBIC,         // PV motion
                        Axis,                   // axis 0
```

```
                        10,                     // uniform interval 10 ms
                        NULL                    // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
                                                // add some points
for (i = 0; i <100; i++)
{
        Point[0]=i*50;    Point[1]=i*100;     Point[2]=i*150;
        Velocity[0]=i*50; Velocity [1]=i*100; Velocity [2]=i*150;
        acsc_AddPVPointM(Handle,Axis,Point,Velocity,NULL);
                //position,velocity and time interval for each point
}
                // the end of the arbitrary path motion
acsc_EndSequence(Handle, ACSC_AXIS_0, NULL);
```

4.22.3 acsc_AddPVTPoint

Description

The function adds a point to a single-axis PVT spline motion and specifies velocity and motion time.

Syntax

int acsc_AddPVTPoint(HANDLE Handle, int Axis, double Point, double Velocity,
double TimeInterval, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axes	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc. For the axis constants see Axis Definitions.
Point	Coordinate of the added point.
Velocity	Desired velocity at the point
TimeInterval	If the motion was activated by the acsc_Spline function with the ACSC_AMF_VARTIME flag, this parameter defines the time interval between the previous point and the present one.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result.

	If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

Before this function can be used, PV spline motion must be initiated by calling acsc_Spline with the appropriate flags.

The function adds a point to a single-axis PVT spline motion with a non-uniform time and specified velocity at that point.

To add a point to a multi-axis PVT motion, use acsc_AddPVTPointM. To add a point to a PV motion with uniform time interval, use the acsc_AddPVPoint and acsc_AddPVPointM functions.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the point is added to the motion buffer. The point can be rejected if the motion buffer is full. In this case, you can call this function periodically until the function returns non-zero value.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
int i;
if (!acsc_Spline(Handle,        // communication handle
                ACSC_AMF_CUBIC|ACSC_AMF_VARTIME,//PVT motion
                ACSC_AXIS_0,    // axis 0
                0,              // uniform interval is not used
                NULL            // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
// add some points
for (i = 0; i <100; i++)
        acsc_AddPVTPoint(Handle,ACSC_AXIS_0,i*100,i*100,100+i,NULL);
        //position,velocity and time interval for each point
// the end of the arbitrary path motion
acsc_EndSequence(Handle, ACSC_AXIS_0, NULL);
```

4.22.4 acsc_AddPVTPointM

Description

The function adds a point to a multiple-axis PVT spline motion and specifies velocity and motion time.

Syntax

int acsc_AddPVTPointM(HANDLE Handle, int *Axis, double *Point,
double *Velocity,double TimeInterval, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions..
Point	Array of the coordinates of added point. The number and order of values must correspond to the **Axes** array. The **Point** must specify a value for each element of **Axes** except the last –1 element.
Velocity	Array of the velocities of added point. The number and order of values must correspond to the **Axes** array. The **Velocity** must specify a value for each element of **Axes** except the last –1 element.
TimeInterval	If the motion was activated by the acsc_SplineM function with the ACSC_AMF_VARTIME flag, this parameter defines the time interval between the previous point and the present one.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

Before this function can be used, PVT spline motion must be initiated by calling acsc_SplineM with the appropriate flags.

The function adds a point to a multiple-axis PVT spline motion with a non-uniform time and specified velocity at that point.

To add a point to a single-axis PVT motion, use acsc_AddPVTPoint. To add a point to a PV motion with uniform time interval, use the acsc_AddPVPoint and acsc_AddPointM functions.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the point is added to the motion buffer. The point can be rejected if the motion buffer is full. In this case, you can call this function periodically until the function returns non-zero value.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
int i;
int Axis[]={ACSC_AXIS_0, ACSC_AXIS_1, ACSC_AXIS_2,-1};
double Point[3];
double Velocity[3];
if (!acsc_SplineM(Handle, // communication handle
                        ACSC_AMF_CUBIC|ACSC_AMF_VARTIME,//PVT motion
                        Axis,   // axis 0
                        0,              // uniform interval is not used
                        NULL            // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
// add some points
for (i = 0; i <100; i++)
{
        Point[0]=i*50;    Point[1]=i*100;     Point[2]=i*150;
        Velocity[0]=i*50; Velocity [1]=i*100; Velocity [2]=i*150;
        acsc_AddPVTPointM(Handle,Axis,Point,Velocity,100+i,NULL);
        //position,velocity and time interval for each point
}
// the end of the arbitrary path motion
acsc_EndSequence(Handle, ACSC_AXIS_0, NULL);
```

4.23 Segmented Motion Functions

The Segmented Motion functions are:

Table 5-22. Segmented Motion Functions

Function	Description
acsc_ ExtendedSegmentedMotionV2	Initiates a multi-axis extended segmented motion.
acsc_SegmentLineV2	Adds a linear segment to a segmented motion.
acsc_ExtendedSegmentArc1V2	Adds an arc segment to a segmented motion and specifies the coordinates of center point, coordinates of the final point, and the direction of rotation.
acsc_SegmentArc2V2	Adds an arc segment to a segmented motion and specifies the coordinates of center point and rotation angle.
acsc_Stopper	Provides a smooth transition between two segments of segmented motion.
acsc_Projection	Sets a projection matrix for a segmented motion.

4.23.1 acsc_ExtendedSegmentedMotionV2

This function replaces acsc_ExtendedSegmentMotionExt, which is now obsolete.

Description

The function initiates a multi-axis extended segmented motion.

Syntax

int acsc_ExtendedSegmentedMotionExtV2(HANDLE Handle, int Flags, int* Axes, double* Point, double Velocity, double EndVelocity, double JunctionVelocity, double Angle, double CurveVelocity, double Deviation, double Radius, double MaxLength, double StarvationMargin, char* Segments, int ExtLoopType, double MinSegmentLength, double MaxAllowedDeviation, int OutputIndex, int BitNumber, int Polarity, double MotionDelay, ACSC_WAITBLOCK* Wait);

Arguments

Handle	Communication handle.
Flags	Bit-mapped argument that can include one or more of the following flags: ACSC_AMF_WAIT: plan the motion but do not start it until the function acsc_GoM is executed. ACSC_AMF_VELOCITY: the motion will use velocity specified for each segment instead of the default velocity. ACSC_AMF_ENDVELOCITY: This flag requires additional parameter that specifies end velocity.

The controller decelerates to the specified velocity in the end of segment.

The specified value should be less than the required velocity; otherwise the parameter is ignored.

This flag affects only one segment.

This flag also disables corner detection and processing at the end of segment.

If this flag is not specified, deceleration is not required. However, in special cases the deceleration might occur due to corner processing or other velocity control conditions.

ACSC_AMF_MAXIMUM: use maximum velocity under axis limits.

With this suffix, no required velocity should be specified.

The required velocity is calculated for each segment individually on the base of segment geometry and axis velocities (VEL values) of the involved axes.

ACSC_AMF_JUNCTIONVELOCITY: Decelerate to corner.

This flag requires additional parameter that specifies corner velocity. The controller detects corner on the path and decelerates to the specified velocity before the corner. The specified value should be less than the required velocity; otherwise the parameter is ignored.

If ACSC_AMF_JUNCTIONVELOCITY flag is not specified while ACSC_AMF_ANGLE flag is specified, zero value of corner velocity is assumed.

If none of the ACSC_AMF_JUNCTIONVELOCITY and ACSC_AMF_ANGLE flags are specified, the controller provides automatic calculation as described in Automatic corner processing.

ACSC_AMF_ANGLE: Do not treat junction as a corner, if junction angle is less than or equal to the specified value in radians.

This flag requires additional parameter that specifies negligible angle in radians.

If ACSC_AMF_ANGLE flag is not specified while ACSC_AMF_JUNCTIONVELOCITY flag is specified, the controller accepts default value of 0.01 radians that is about 0.57 degrees.

If none of the ACSC_AMF_JUNCTIONVELOCITY and ACSC_AMF_ANGLE flags are specified, the controller provides automatic calculation as described in Automatic corner processing.

ACSC_AMF_AXISLIMIT

Enable velocity limitations under axis limits.

With this flag set, setting the ACSC_AMF_VELOCITY flag will result in the requested velocity being restrained by the velocity limits of all involved axes.

ACSC_AMF_CURVEVELOCITY

Decelerate to curvature discontinuity point.

This flag requires an additional parameter that specifies velocity at curvature discontinuity points.

Curvature discontinuity occurs in linear-to-arc or arc-to-arc smooth junctions.

If the flag is not set, the controller does not decelerate to smooth junction disregarding curvature discontinuity in the junction.

If the flag is set, the controller detects curvature discontinuity points on the path and provides deceleration to the specified velocity.

The specified value should be less than the required velocity; otherwise the parameter is ignored.

The flag can be set together with flags ACSC_AMF_JUNCTIONVELOCITY and/or ACS_AMF_ANGLE.

If neither of ACSC_AMF_JUNCTIONVELOCITY, ACS_AMF_ANGLE or ACSC_AMF_CURVEVELOCITY is set, the controller provides automatic calculation of the corner processing.

ACSC_AMF_CURVEAUTO

If the Flag is specified the controller provides automatic calculations as described in Enhanced automatic corner and curvature discontinuity points processing.

ACSC_AMF_CORNERDEVIATION

Use a corner rounding option with the specified permitted deviation. This flag requires an additional parameter that specifies maximal allowed deviation of motion trajectory from the corner point. This flag cannot be set together with flags ACSC_AMF_CORNERRADIUS and ACSC_AMF_CORNERLENGTH.

ACSC_AMF_CORNERRADIUS

Use a corner rounding option with the specified permitted curvature. This flag requires an additional parameter that specifies maximal allowed rounding radius of the additional segment. This flag cannot be

specified together with flags ACSC_AMF_CORNERLENGTH or ACSC_AMF_CORNERDEVIATION.

ACSC_AMF_CORNERLENGTH

Use automatic corner rounding option.

This flag requires an additional parameter that specifies the maximum length of the segment for automatic corner rounding. If a length of one of the segments that built a corner exceeds the specified maximal length, the corner will not be rounded. The automatic corner rounding is only applied to pair of linear segments. If one of the segments in a pair is an arc, the rounding is not applied for this corner.

This flag cannot be set together with flags ACSC_AMF_CORNERDEVIATION or ACSC_AMF_CORNERRADIUS.

ACSC_AMF_EXT_LOOP

Use external loops at corners.

The switch requires additional parameters

that specify the external loop type, the minimum segment length, and

the maximum allowed deviation from profile.

ACSC_AMF_EXT_LOOP_SYNC

Defines output bit to support external loop synchronization..

The switch requires additional parameters

that specify the output index, the bit number, and

the polarity.

ACSC_AMF_DELAY_MOTION

Defines actual motor movement delay in microseconds. The delay resolution is 50 microseconds.

The switch requires additional parameter

that specify the motion delay.

ACSC_AMF_LOCALCS

Interpret entered coordinates according to the Local Coordinate System.

With this switch, use 2 axes motion coordinate only (X,Y). Using 3 or more coordinates causes a runtime error.

Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions.
Point	Array of the starting point coordinates. The number and order of values must correspond to the Axes array. **Point** must specify a value for each element of **Axes** except the last –1 element.
Velocity	If ACSC_AMF_VELOCITY flag was specified, this argument specifies a motion velocity for current segment. Set this argument to ACSC_NONE if not used.
EndVelocity	If ACSC_AMF_ENDVELOCITY flag was specified, this argument defines required velocity at the end of the current segment. Set this argument to ACSC_NONE if not used.
JunctionVelocity	If ACSC_AMF_JUNCTIONVELOCITY flag was specified, this argument defines the required velocity at the junction. Set this argument to ACSC_NONE if not used.
Angle	If ACSC_AMF_ANGLE flag was specified, this argument specifies the threshold above which a junction angle will be treated as a corner. Set this argument to ACSC_NONE if not used.
CurveVelocity	If ACSC_AMF_CURVEVELOCITY flag has been specified, this argument defines the required velocity at curvature discontinuity points. Set this argument to ACSC_NONE if not used.
Deviation	If ACSC_AMF_CORNERDEVIATION flag has been specified, this argument defines the maximal allowed trajectory deviation from the corner point. Set this argument to ACSC_NONE if not used.
Radius	If ACSC_AMF_CORNERRADIUS flag has been specified, this argument defines the maximal allowed rounding radius of the additional segment. Set this argument to ACSC_NONE if not used.

MaxLength	If ACSC_AMF_CORNERLENGTH flag has been specified, this argument defines the maximum length of the segment for processing automatic corner rounding. If a segment's corner attempts to exceed the maximum length, the corner will not be rounded. Set this argument to ACSC_NONE if not used.
StarvationMargin	Starvation margin in milliseconds. The controller sets the **AST.#NEWSEGM** bit to the **StarvationMargin** millisecond before the starvation condition occurs. Set this argument to ACSC_NONE if not used. By default, if this argument is not specified, the starvation margin is 500 milliseconds.

If the Segments parameter is used, then the starvation margin must also be defined.

Segments	Pointer to the null-terminated character string that contains the name of a one-dimensional user-defined array used to store added segments. Set this argument to NULL if not used. By default, if this argument is not specified, the controller allocates internal buffer for storing 50 segments only. The argument allows the user application to reallocate the buffer for storing a larger number of segments. The larger number of segments may be required if the application needs to add many very small segments in advanced. For most applications, the internal buffer size is enough and should not be enlarged. The buffer is for the controller internal use only and should not be used by the user application. The buffer size calculation rule: each segment requires about 600 bytes, so if it is necessary to allocate the buffer for 200 segments, it should be at least 600 * 200 = 120,000 bytes. The following declaration defines a 120,000 bytes buffer: real buf (15000) See **XARRSIZE** explanation in the *ACSPL+ Command and Variable Reference Guide* for details on how to declare a buffer with more than 100,000 elements.
ExtLoopType	If ACSC_AMF_EXT_LOOP flag has been specified, this argument defines the external loop type.

	0 - Cancel external loop 1 – Smooth External loop (line-arc-line) 2 – Triangle External loop (line-line-line) Set this argument to ACSC_NONE if not used
MinSegmentLength	If ACSC_AMF_EXT_LOOP flag has been specified, this argument defines the Minimum Segment Length. If the lengths of both segments are more than this value, the skywriting algorithm will be applied. Set this argument to ACSC_NONE if not used
MaxAllowedDeviation	If ACSC_AMF_EXT_LOOP flag has been specified, this argument defines the maximum allowed deviation. The parameter limits the external loop deviation from the defined profile. If the value is negative there is no limitation. Set this argument to ACSC_NONE if not used
OutputIndex	If ACSC_AMF_EXT_LOOP_SYNC flag has been specified, this argument defines the output index, the index of the digital output port to assigned to synchronization (read by OUT) Set this argument to ACSC_NONE if not used
BitNumber	If ACSC_AMF_EXT_LOOP_SYNC flag has been specified, this argument defines the bit number. Bit number (BIT_NUMBER_MIN(0) - BIT_NUMBER_MAX(16)) assigned to synchronization output. Set this argument to ACSC_NONE if not used.
Polarity	If ACSC_AMF_EXT_LOOP_SYNC flag has been specified, this argument defines the polarity. POLARITY_ON(0) or POLARITY_OFF(1) - which value is considered the initial state Set this argument to ACSC_NONE if not used
MotionDelay	If ACSC_AMF_DELAY_MOTION flag has been specified, this argument defines the actual motor movement delay in microseconds. The delay resolution is 50 microseconds. The maximum delay is 100 controller cycles or 100ms for CTIME=1ms or 20ms for CTIME=0.2ms. Set this argument to ACSC_NONE if not used.

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling acsc_GetLastError.

Comments

The function itself does not specify any segment, so the created motion starts only after the first segment is specified.

The segments of motion are specified by using the **acsc_SegmentLineV2**, **acsc_SegmentArc1V2**, and **acsc_SegmentArc2V2** functions.

The motion finishes when the **acsc_EndSequenceM** function is executed. If the call to **acsc_EndSequenceM** is omitted, the motion will stop at the last segment of the sequence and wait for the next segment. No transition to the next motion in the motion queue will occur until the function **acsc_EndSequenceM** is executed.

During positioning to each point, a vector velocity profile is built using the default values of velocity, acceleration, deceleration, and jerk of the axis group. If the ACSC_AFM_VELOCITY flag is not specified, the default value of velocity is used as well. If the ACSC_AFM_VELOCITY flag is specified, the value of velocity specified in subsequent acsc_SegmentLineExt, acsc_ExtendedSegmentArc1, or acsc_ExtendedSegmentArc2 functions is used.

The function can wait for the controller response or can return immediately as specified by the Wait argument.

The controller response indicates that the command was accepted and the motion was planned successfully. The function does not wait and does not validate the end of the motion. To wait for the motion end, use the acsc_WaitMotionEnd function.

If Wait points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the Wait item until a call to the acsc_WaitForAsyncCall function is made.

Supported from V3.12.

Example

```
// Example of the waiting call of acsc_ExtendedSegmentedMotionV2 Int Axes
[] = {ACSC_AXIS_0, ACSC_AXIS_1, -1};
double Point[2], Center[2]; Point[0] = 1000; Point[1] = 1000;
```

```
if (!acsc_ExtendedSegmentedMotionV2(Handle,
ACSC_AMF_VELOCITY | ACSC_AMF_CORNERRADIUS,
// Velocity and corner radius flags are set, parameters now required.
Axes, //Axes 0,1 active Point, // Starting point
5000, // Velocity is set to 5000
ACSC_NONE, //EndVelocity is the default value
ACSC_NONE, //JunctionVelocity is the default value
 ACSC_NONE, // Angle is the default value
ACSC_NONE, // CurveVElocity is the default value ACSC_NONE, // Deviation
10, // Radius is set
ACSC_NONE, // Maximum corner length is default
ACSC_NONE, // Starvation margin is the default value
ACSC_SYNCHRONOUS // Waiting call
))}
```

4.23.2 acsc_SegmentLineV2

This function replaces the **acsc_SegmentLineExt** which is now obsolete.

Description

The function adds a linear segment that starts at the current point and ends at the destination point of segmented motion.

Syntax

Int acsc_SegmentLineExtV2(HANDLE handle, int Flags, int* Axes, Double* Point, double Velocity, double EndVelocity, int Time, char* Values, char* Variables, int Index, char* Masks, int ExtLoopType, double MinSegmentLength, double MaxAllowedDeviation, int LciState, ACSC_WAITBLOCK* Wait);

Arguments

Handle	Communication handle.
Flags	Bit-mapped argument that can include one or more of the following flags: ACSC_AMF_VELOCITY: the motion will use velocity specified for each segment instead of the default velocity. ACSC_AMF_ENDVELOCITY: this flag requires additional parameter that specifies end velocity. The controller decelerates to the specified velocity in the end of segment. The specified value should be less than the required velocity; otherwise the parameter is ignored. This flag affects only one segment.

	ACSC_AMF_VARTIME: this flag requires an addition parameter that specifies the segment processing time in milliseconds. The segment processing time defines velocity at the current segment only and has no effect on subsequent segments. This flag also disables corner detection and processing at the end of segment. If this flag is not specified, deceleration is not required. However, in special cases the deceleration might occur due to corner processing or other velocity control conditions. ACSC_AMF_USERVARIABLES: synchronize user variables with segment execution. This flag requires additional parameters that specify values, user variable and mask. See details in **Values**, **Variables**, and **Masks** below. ACSC_AMF_EXT_LOOP: Use external loops at corners. The switch requires additional parameters that specify the external loop type, the minimum segment length, and the maximum allowed deviation from profile. ACSC_AMF_LCI_STATE: The switch requires additional parameter that specify LCI state.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions..
Point	Array of the final point coordinates. The number and order of values must correspond to the **Axes** array. The **Point** must specify a value for each element of **Axes** except the last –1 element.
Velocity	If ACSC_AMF_VELOCITY flag has been specified, this argument specifies a motion velocity for current segment. Set this argument to ACSC_NONE if not used.
EndVelocity	If ACSC_AMF_ENDVELOCITY flag has been specified, this argument defines the required velocity at the end of the current segment. Set this argument to ACSC_NONE if not used.

Time	If ACSC_AMF_VARTIME flag has been specified, this argument defines the segment processing time in milliseconds, for the current segment only and has no effect on subsequent segments. Set this argument to ACSC_NONE if not used.
Values	Pointer to the null-terminated character string that contains the name of a one-dimensional user-defined array of integer or real type with a size of 10 elements maximum. If ACSC_AMF_USERVARIABLES flag has been specified, this argument defines the values to be written to the **Variables** array at the beginning of the current segment execution. Set this argument to NULL if not used.
Variables	Pointer to the null-terminated character string that contains the name of a one-dimensional user-defined array of the same type and size as **Values** array. If ACSC_AMF_USERVARIABLES flag has been specified, this argument defines the user-defined array, which will be written with **Values** data at the beginning of the current segment execution. Set this argument to NULL if not used.
Index	If ACSC_AMF_USERVARIABLES has not been specified, this argument defines the first element (starting from zero) of the **Variables** array, to which **Values** data will be written to. Set this argument to ACSC_NONE if not used.
Masks	Pointer to the null-terminated character string that contains the name of a one-dimensional user defined array of integer type and same size as the **Values** array. If ACSC_AMF_USERVARIABLES flag has been specified, this argument defines the masks that are applied to **Values** before the **Values** are written to variables array at the beginning of the current segment execution. The masks are only applied for integer values: *variables(n) = values(n) AND mask(n)* If **Values** is a real array, the masks argument should be NULL. Set this argument to ACSC_NONE if not used.
ExtLoopType	If ACSC_AMF_EXT_LOOP flag has been specified, this argument defines the external loop type. 0 - Cancel external loop

	1 – Smooth External loop (line-arc-line) 2 – Triangle External loop (line-line-line) Set this argument to ACSC_NONE if not used.
MinSegmentLength	If ACSC_AMF_EXT_LOOP flag has been specified, this argument defines the Minimum Segment Length. If the lengths of both segments are more than this value, the skywriting algorithm will be applied. Set this argument to ACSC_NONE if not used.
MaxAllowedDeviation	If ACSC_AMF_EXT_LOOP flag has been specified, this argument defines the Maximum Allowed Deviation. The parameter limits the external loop deviation from the defined profile. If the value is negative there is no limitation. Set this argument to ACSC_NONE if not used
LciState	If ACSC_AMF_LCI_STATE flag has been specified, this argument defines the LCI state. LCI_STATE_ON(0) or LCI_STATE_OFF(1) determines the value to be considered the initial state. Set this argument to ACSC_NONE if not used.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function adds a linear segment that starts at the current point and ends at the destination point to segmented motion.

All axes specified in the **Axes** array must be specified before the call of the acsc_ExtendedSegmentedMotionV2 function. The number and order of the axes in the **Axes** array must correspond exactly to the number and order of the axes of the acsc_ExtendedSegmentedMotionV2 function.

The **Point** argument specifies the coordinates of the final point. The coordinates are absolute in the plane.

ACSC_AMF_VELOCITY and ACSC_AMF_VARTIME are mutually exclusive, meaning they cannot be used together.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the segment is added to the motion buffer. The segment can be rejected if the motion buffer is full. In that case, you can call this function periodically until the function returns a non-zero value

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Supported from V3.12.

Example

```
// Example of the waiting call of acsc_SegmentLineV2
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, -1 };
double Point[2];
// create segmented motion, coordinates of the initial point are
Point[0] = 1000; Point[1] = 1000;
if (!acsc_ExtendedSegmentedMotionV2(Handle,// Communication handle ACSC_
AMF_VELOCITY, // Create the segmented motion with specified
Axes, // Axes 0 and 1 Point, // Initial point
1000, // Vector velocity is specified
ACSC_NONE, // End velocity is not specified ACSC_NONE, // Junction
velocity is not specified
ACSC_NONE, // Angle is not specified
ACSC_NONE, // Curve velocity is not specified
 ACSC_NONE, // Deviation is not specified
ACSC_NONE, // curve radius is not specified
ACSC_NONE, // maximal curve length is not specified
ACSC_NONE, // Default starvation margin will be used
 NULL, // Internal buffer will be used,
ACSC_NONE, // external_loop_type
ACSC_NONE, // minimal_segment_length
ACSC_NONE, // maximum_allowed_deviation
ACSC_NONE, // output_index
ACSC_NONE, // bit_number
ACSC_NONE, // polarity
ACSC_NONE, // MotionDelay
NULL // Waiting call
))
```

```
{
printf("transaction error: %d\n", acsc_GetLastError());
}
// add line segment with final point (1000, -1000), vector velocity 25000
Point[0] = 1000; Point[1] = -1000;
if (!acsc_SegmentLineV2(Handle, // Communication handle ACSC_AMF_
VELOCITY, // Velocity is specified
Axes, // Axes 0 and 1 Point, // Final point
25000, // Vector velocity
ACSC_NONE, // End velocity is not specified
 ACSC_NONE, // Time is not specified
NULL, // Values array is not specified
 NULL, // Variables array is not specified
ACSC_NONE, // Index is not specified
 NULL, // Masks array is not specified
ACSC_NONE, // external_loop_type
ACSC_NONE, // minimal_segment_length
ACSC_NONE, // maximum_allowed_deviation
ACSC_NONE, // Lci State
NULL // Waiting call
))
{
printf("transaction error: %d\n", acsc_GetLastError());
}
// add line segment with final point (1000, 1000), vector velocity 5000
Point[0] = 1000; Point[1] = 1000;

if (!acsc_SegmentLineV2(Handle, ACSC_AMF_VELOCITY, // Velocity is
specified Axes, // Axes 0 and 1
Point, // Final point
5000, // Vector velocity
ACSC_NONE, // End velocity is not specified
ACSC_NONE, // Time is not specified
NULL,  // Values array is not specified
NULL, // Variables array is not specified
 ACSC_NONE, // Index is not specified
 NULL, // Masks array is not
ACSC_NONE, // external_loop_type
ACSC_NONE, // minimal_segment_length
ACSC_NONE, // maximum_allowed_deviation
ACSC_NONE, // Lci State
specified ACSC_SYNCHRONOUS // Waiting call
))
{
printf("transaction error: %d\n", acsc_GetLastError());
}
// finish the motion
acsc_EndSequenceM(Handle, Axes, NULL);
```

4.23.3 acsc_ExtendedSegmentArc1V2

This function replaces the **acsc_SegmentArc1Ext** which is now obsolete.

Description

The function defines an arc segment that starts at the current point and ends at the destination point with the specified center point. The segment is added to the motion path .

Syntax

```
int  acsc_SegmentArc1V2(HANDLE Handle, int Flags, int* Axes, double*
Center, double* FinalPoint, int Rotation, double Velocity, double
EndVelocity, double Time, char* Values, char* Variables, int Index, char*
Masks, int ExtLoopType, double MimSegmentLength, double
MaxAllowedDeviation, int LciState, ACSC_WAITBLOCK* Wait)
```

Arguments

Handle	Communication handle.
Flags	Bit-mapped argument that can include one or more of the following flags: ACSC_AMF_VELOCITY: the motion will use velocity specified for each segment instead of the default velocity. ACSC_AMF_ENDVELOCITY: this flag requires an additional parameter that specifies end velocity. The controller decelerates to the specified velocity in the end of segment. The specified value should be less than the required velocity; otherwise the argument is ignored. This flag affects only one segment. ACSC_AMF_VARTIME: this flag requires an addition parameter that specifies the segment processing time in milliseconds. The segment processing time defines velocity at the current segment only and has no effect on subsequent segments. This flag also disables corner detection and processing at the end of segment. If this flag is not specified, deceleration is not required. However, in special cases the deceleration might occur due to corner processing or other velocity control conditions.

	ACSC_AMF_USERVARIABLES: synchronize user variables with segment execution. This flag requires additional parameters that specify values, user variable and mask. See details in the **Values**, **Variables**, and **Masks** arguments below. ACSC_AMF_EXT_LOOP: Use external loops at corners. The switch requires additional parameters that specify the external loop type, the minimum segment length, and the maximum allowed deviation from profile. ACSC_AMF_LCI_STATE: The switch requires additional parameter that specify LCI state.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to 0, ACSC_AXIS_1 to 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions.
Center	Array of the center coordinates. The number and order of values must correspond to the **Axes** array. The **Center** must specify a value for each element of the **Axes** except the last–1 element.
FinalPoint	Array of the final point coordinates. The number and order of values must correspond to the **Axes** array. The **FinalPoint** must specify a value for each element of **Axes** except the last –1 element.
Rotation	This argument defines the direction of rotation. If **Rotation** is set to ACSC_COUNTERCLOCKWISE, then the rotation is counterclockwise. If **Rotation** is set to ACSC_CLOCKWISE, then rotation is clockwise.
Velocity	If ACSC_AMF_VELOCITY flag has been specified, this argument specifies a motion velocity for current segment. Set this argument to ACSC_NONE if not used.
EndVelocity	If ACSC_AMF_ENDVELOCITY flag has been specified, this argument defines required velocity at the end of the current segment. Set this argument to ACSC_NONE if not used.
Time	If ACSC_AMF_VARTIME flag has been specified, this argument defines the segment processing time in milliseconds, for the current segment only and has no effect on subsequent segments. Set this argument to ACSC_NONE if not used.

Values	Pointer to the null-terminated character string that contains the name of a one-dimensional user-defined array of integer or real type with a size of 10 elements maximum. If ACSC_AMF_USERVARIABLES flag has been specified, this argument defines the values to be written to the **Variables** array at the beginning of the current segment execution. Set this argument to NULL if not used.
Variables	Pointer to the null-terminated character string that contains the name of a one-dimensional user-defined array of the same type and size as **Values** array.. If ACSC_AMF_USERVARIABLES flag has been specified, this argument defines the user-defined array, which will be written with **Values** data at the beginning of the current segment execution. Set this argument to NULL if not used.
Index	If ACSC_AMF_USERVARIABLES has not been specified, this argument defines the first element (starting from zero) of the **Variables** array, to which **Values** data will be written to. Set this argument to ACSC_NONE if not used.
Masks	Pointer to the null-terminated character string that contains the name of a one-dimensional user defined array of integer type and same size as the **Values** array. If ACSC_AMF_USERVARIABLES flag has been specified, this argument defines the masks that are applied to **Values** before the **Values** are written to variables array at the beginning of the current segment execution. The masks are only applied for integer values: *variables(n) = values(n) AND mask(n)* If **Values** is a real array, the masks argument should be NULL. Set this argument to ACSC_NONE if not used.
ExtLoopType	If ACSC_AMF_EXT_LOOP flag has been specified, this argument defines the external loop type. 0 - Cancel external loop 1 – Smooth External loop (line-arc-line) 2 – Triangle External loop (line-line-line) Set this argument to ACSC_NONE if not used

MinSegmentLength	If ACSC_AMF_EXT_LOOP flag has been specified, this argument defines the Minimum Segment Length. If the lengths of both segments are more than this value, the skywriting algorithm will be applied. Set this argument to ACSC_NONE if not used.
MaxAllowedDeviation	If ACSC_AMF_EXT_LOOP flag has been specified, this argument defines the Maximum Allowed Deviation. The parameter limits the external loop deviation from the defined profile. If the value is negative there is no limitation. Set this argument to ACSC_NONE if not used.
LciState	If ACSC_AMF_LCI_STATE flag has been specified, this argument defines the LCI state. LCI_STATE_ON(0) or LCI_STATE_OFF(1) determines the value to be considered the initial state. Set this argument to ACSC_NONE if not used.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

All axes specified in the **Axes** array must be specified before the call of the acsc_ExtendedSegmentedMotionV2 function. The number and order of the axes in the **Axes** array must correspond exactly to the number and order of the axes of the acsc_ExtendedSegmentedMotionV2 function.

The **Point** argument specifies the coordinates of the final point. The coordinates are absolute in the plane.

ACSC_AMF_VELOCITY and ACSC_AMF_VARTIME are mutually exclusive, meaning they cannot be used together.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the segment is added to the motion buffer. The segment can be rejected if the motion buffer is full. In that case, call this function periodically until the function returns a non-zero value.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the "acsc_WaitForAsyncCall" on page 53 function.

Supported from V3.12.

Example

```
// Example of the waiting call of acsc_SegmentArc1V2
 int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, -1 };
double Point[2],CntrPnt[2];
// create segmented motion, coordinates of the initial point are
// (1000, 1000)
Point[0] = 1000; Point[1] = 1000;
if (!acsc_ExtendedSegmentedMotionV2(Handle,// Communication handle ACSC_
AMF_VELOCITY, // Create the segmented motion with specified
// velocity
Axes, // Axes 0 and 1 Point, // Initial point
1000, // Vector velocity is specified
ACSC_NONE, // End velocity is not specified
ACSC_NONE, // Junction velocity is not specified
ACSC_NONE, // Angle is not specified
ACSC_NONE, // Curve velocity is not specified ACSC_NONE, // Deviation is
not specified ACSC_NONE, // curve radius is not specified
ACSC_NONE, // maximal curve length is not specified
ACSC_NONE, // Default starvation margin will be used
NULL, // Internal buffer will be used
ACSC_NONE, // external_loop_type
ACSC_NONE, // minimal_segment_length
ACSC_NONE, // maximum_allowed_deviation
ACSC_NONE, // output_index
ACSC_NONE, // bit_number
ACSC_NONE, // polarity
ACSC_NONE, // MotionDelay
ACSC_SYNCHRONOUS // Waiting call
))
{
printf("transaction error: %d\n", acsc_GetLastError());
}
// add arc segment with final point (1000, -1000), vector velocity 7500
Point[0] = 1000; Point[1] = -1000;
CntrPnt[0] = 0; CntrPnt[1] = 0;
```

```
if (!acsc_SegmentArc1V2(Handle, // Communication handle ACSC_AMF_
VELOCITY, // Velocity is specified
Axes, //Axes 0 and 1 CntrPnt, //Center point Point, // Final point
ACSC_COUNTERCLOCKWISE, //Positive rotation
7500, // Vector velocity
ACSC_NONE, // End velocity is not specified
 ACSC_NONE, // Time is not specified
NULL, // Values array is not specified
NULL, // Variables array is not specified
ACSC_NONE, // Index is not specified NULL, // Masks array is not
specified
ACSC_NONE, //ExtLoopType
ACSC_NONE, //MinSegemntlegth
ACSC_NONE, // maximum_allowed_deviation
ACSC_NONE, // LciState
ACSC_SYNCHRONOUS // Waiting call
))
{
printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.23.4 acsc_SegmentArc2V2

This function replaces the **acsc_SegmentArc2Ext** which is now obsolete.

Description

The function adds an arc segment to a segmented motion and specifies the coordinates of the center point and the rotation angle.

Syntax

```
int acsc_SegmentArc2V2(HANDLE Handle, int Flags, int* Axes, double*
Center, double Angle, double* FinalPoint, double Velocity, double
EndVelocity, double Time, char* Values, char* Variables, int Index, char*
Masks, int ExtLoopType, double MimSegmentLength, double
MaxAllowedDeviation, int LciState, ACSC_WAITBLOCK* Wait)
```

Arguments

Handle	Communication handle.
Flags	Bit-mapped argument that can include one or more of the following flags: ACSC_AMF_VELOCITY: the motion will use velocity specified for each segment instead of the default velocity.

	ACSC_AMF_ENDVELOCITY: this flag requires an additional parameter that specifies end velocity. The controller decelerates to the specified velocity in the end of segment. The specified value should be less than the required velocity; otherwise the argument is ignored. This flag affects only one segment. ACSC_AMF_VARTIME: this flag requires an addition parameter that specifies the segment processing time in milliseconds. The segment processing time defines velocity at the current segment only and has no effect on subsequent segments. This flag also disables corner detection and processing at the end of segment. If this flag is not specified, deceleration is not required. However, in special cases the deceleration might occur due to corner processing or other velocity control conditions. ACSC_AMF_USERVARIABLES: synchronize user variables with segment execution. This flag requires additional parameters that specify values, user variable and mask. See details in the **Values**, **Variables**, and **Masks** arguments below. ACSC_AMF_EXT_LOOP: Use external loops at corners. The switch requires additional parameters that specify the external loop type, the minimum segment length, and the maximum allowed deviation from profile. ACSC_AMF_LCI_STATE: The switch requires an additional parameter that specifies LCI state.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to 0, ACSC_AXIS_1 to 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions.
Center	Array of the center coordinates. The number and order of values must correspond to the **Axes** array. The **Center** must specify a value for each element of the **Axes** except the last–1 element.
Angle	Rotation angle in radians. Positive angle for counterclockwise rotation, negative for clockwise rotation.
FinalPoints	Array indicating the final points of the secondary axes, array size must be number of secondary axes (size of **Axes** – 2). Set this argument to **NULL** if not used.

Velocity	If ACSC_AMF_VELOCITY flag has been specified, this argument specifies a motion velocity for current segment. Set this argument to ACSC_NONE if not used.
EndVelocity	If ACSC_AMF_ENDVELOCITY flag has been specified, this argument defines required velocity at the end of the current segment. Set this argument to ACSC_NONE if not used.
Time	If ACSC_AMF_VARTIME flag has been specified, this argument defines the segment processing time in milliseconds, for the current segment only and has no effect on subsequent segments. Set this argument to ACSC_NONE if not used.
Values	Pointer to the null-terminated character string that contains the name of a one-dimensional user-defined array of integer or real type with a size of 10 elements maximum. If ACSC_AMF_USERVARIABLES flag has been specified, this argument defines the values to be written to the **Variables** array at the beginning of the current segment execution. Set this argument to NULL if not used.
Variables	Pointer to the null-terminated character string that contains the name of a one-dimensional user-defined array of the same type and size as **Values** array. If ACSC_AMF_USERVARIABLES flag has been specified, this argument defines the user-defined array, which will be written with **Values** data at the beginning of the current segment execution. Set this argument to NULL if not used.
Index	If ACSC_AMF_USERVARIABLES has not been specified, this argument defines the first element (starting from zero) of the **Variables** array, to which **Values** data will be written to. Set this argument to ACSC_NONE if not used.

Masks	Pointer to the null-terminated character string that contains the name of a one-dimensional user defined array of integer type and same size as the **Values** array. If ACSC_AMF_USERVARIABLES flag has been specified, this argument defines the masks that are applied to **Values** before the **Values** are written to variables array at the beginning of the current segment execution. The masks are only applied for integer values: *variables(n) = values(n) AND mask(n)* If **Values** is a real array, the masks argument should be NULL. Set this argument to ACSC_NONE if not used.
ExtLoopType	If ACSC_AMF_EXT_LOOP flag has been specified, this argument defines the external loop type. 0 - Cancel external loop 1 – Smooth External loop (line-arc-line) 2 – Triangle External loop (line-line-line) Set this argument to ACSC_NONE if not used
MinSegmentLength	If ACSC_AMF_EXT_LOOP flag has been specified, this argument defines the Minimum Segment Length.If the lengths of both segments are more than this value, the skywriting algorithm will be applied. Set this argument to ACSC_NONE if not used.
MaxAllowedDeviation	If ACSC_AMF_EXT_LOOP flag has been specified, this argument defines the Maximum Allowed Deviation. The parameter limits the external loop deviation from the defined profile. If the value is negative there is no limitation. Set this argument to ACSC_NONE if not used.
LciState	If ACSC_AMF_LCI_STATE flag has been specified, this argument defines the LCI state. LCI_STATE_ON(0) or LCI_STATE_OFF(1) determines the value to be considered the initial state. Set this argument to ACSC_NONE if not used.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result.

	If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

All axes specified in the **Axes** array must be specified before calling the function. The number and order of the axes in the **Axes** array must correspond exactly to the number and order of the axes of the **acsc_SegmentedMotion** or **acsc_ExtendedSegmentedMotionV2** function.

The **Point** argument specifies the coordinates of the final point. The coordinates are absolute in the plane.

ACSC_AMF_VELOCITY and ACSC_AMF_VARTIME are mutually exclusive, meaning they cannot be used together.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the segment is added to the motion buffer. The segment can be rejected if the motion buffer is full. In that case, you can call this function periodically until the function returns a non-zero value.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the "acsc_WaitForAsyncCall" on page 53 function.

Supported from V3.12.

Example

```
int axes[] = { 0,1,2,3,-1 };
acsc_EnableM(Handle, axes, NULL);
acsc_SetRPosition(Handle,      0,      1000, NULL);
acsc_SetRPosition(Handle,      1,      1000, NULL);
acsc_SetRPosition(Handle,      2,      500, NULL);
acsc_SetRPosition(Handle,      3,      500, NULL);

double Point[4], center[2], FinalPoint[4], FinalPointArc2[2]; Point[0] =
1000; Point[1] = 1000; Point[2] = 500; Point[3] = 500;

acsc_ExtendedSegmentedMotionV2(Handle,ACSC_AMF_VELOCITY, axes, Point,
5000, ACSC_NONE, ACSC_NONE, ACSC_NONE, ACSC_NONE, ACSC_NONE, ACSC_NONE,
ACSC_NONE, ACSC_NONE, NULL, ACSC_NONE, ACSC_NONE, ACSC_NONE, ACSC_NONE,
ACSC_NONE, ACSC_NONE, ACSC_NONE,  ACSC_SYNCHRONOUS );
```

```
center[0] = -1000; center[1] = 0;
FinalPointArc2[0] = -500; FinalPointArc2[1] = 500;

if (!acsc_SegmentArc2V2(Handle,
ACSC_AMF_VELOCITY | ACSC_AMF_ENDVELOCITY,// Flags
axes,// Axes center,//Center
-3.141529,// Angle FinalPointArc2,// FinalPoints 5000,// Velocity
10000,// EndVelocity ACSC_NONE,// Time NULL,// Values NULL,// Variables
ACSC_NONE,// Index NULL,// Masks
ACSC_NONE, //ExtLoopType
ACSC_NONE, //MinSegemntlegth
ACSC_NONE, // maximum_allowed_deviation
ACSC_NONE, // LciState
NULL))// Wait
{
printf("transaction error: %d\n", acsc_GetLastError());
}

acsc_EndSequenceM(Handle, axes, NULL);
```

4.23.5 acsc_Stopper

Description

The function provides a smooth transition between two segments of segmented motion.

Syntax

int acsc_Stopper(HANDLE Handle, int* Axes, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The controller builds the motion so that the vector velocity follows the smooth velocity diagram. The segments define the projection of the vector velocity to axis velocities. If all segments are connected smoothly, axis velocity is also smooth. However, if the user defined a path with an inflection point, axis velocity has a jump in this point. The jump can cause a motion failure due to the acceleration limit.

The function is used to avoid velocity jump in the inflection points. If the function is specified between two segments, the controller provides smooth deceleration to zero in the end of first segment and smooth acceleration to specified velocity in the beginning of second segment.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// Example of the waiting call of acsc_Stopper
// the example provides a rectangular path without velocity jumps
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, -1 };
double Point[2];
// create segmented motion, coordinates of the initial point are
// (1000,1000)
Point[0] = 1000; Point[1] = 1000;
acsc_SegmentedMotion(Handle, 0, Axes, Point, NULL);
// add line segment with final point (1000, -1000)
Point[0] = 1000; Point[1] = -1000;
acsc_Line(Handle, Axes, Point, NULL);
acsc_Stopper(Handle, Axes, NULL);
// add line segment with final point (-1000, -1000)
Point[0] = -1000; Point[1] = -1000;
acsc_Line(Handle, Axes, Point, NULL);
acsc_Stopper(Handle, Axes, NULL);
// add line segment with final point (-1000, 1000)
Point[0] = -1000; Point[1] = 1000;
acsc_Line(Handle, Axes, Point, NULL);
acsc_Stopper(Handle, Axes, NULL);
// add line segment with final point (1000, 1000)
Point[0] = 1000; Point[1] = 1000;
acsc_Line(Handle, Axes, Point, NULL);
// finish the motion
acsc_EndSequenceM(Handle, Axes, NULL);
```

4.23.6 acsc_Projection

Description

The function sets a projection matrix for segmented motion.

Syntax

int acsc_Projection(HANDLE Handle, int* Axes, char* Matrix,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions.
Matrix	Pointer to the null-terminated string containing the name of the matrix that provides the specified projection.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function sets a projection matrix for segmented motion.

The projection matrix connects the plane coordinates and the axis values in the axis group. The projection can provide any transformation as rotation or scaling. The number of the matrix rows must be equal to the number of the specified axes. The number of the matrix columns must equal two.

The matrix must be declared before as a global variable by an ACSPL+ program or by the acsc_DeclareVariable function and must be initialized by an ACSPL+ program or by the acsc_WriteReal function.

For more information about projection, see the *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_Projection
int Axes[4];
double Point[2], Center[2];
// prepare the projection matrix
double Matrix[3][2] = { { 1, 0 },
                        { 0, 1.41421 },
                        { 0, 1.41421 }  };
// declare the matrix that will contain the projection
acsc_DeclareVariable(Handle, ACSC_REAL_TYPE, "ProjectionMatrix(3)(2)",
                        NULL);
// initialize the projection matrix
acsc_WriteReal(Handle, ACSC_NONE, "ProjectionMatrix", 0, 2, 0, 1, Matrix,
                        NULL);
Axes[0]= ACSC_AXIS_0; Axes[1]=ACSC_AXIS_1;
Axes[2]= ACSC_AXIS_2; Axes[3] = -1;     // create a group of the involved
                                        // axes
acsc_Group(Handle, Axes, NULL);         // create segmented motion,
                                        // coordinates of the initial point
                                        // are (1000,1000)
Axes[0] = ACSC_AXIS_0; Axes[1] = ACSC_AXIS_1; Axes[2] = -1;
Point[0] = 1000; Point[1] = 1000;
acsc_SegmentedMotion(Handle, 0, Axes, Point, NULL);
                                        // incline the working plane XY by
                                        // 45°
Axes[0] = 0; Axes[1] = 1; Axes[2] = 2; Axes[3] = -1;
acsc_Projection(Handle, Axes, "ProjectionMatrix", NULL);
// describe circle with center (1000, 0), clockwise rotation
// although the circle was defined, really on the plane XY we will get
the
// ellipse stretched along the Y axis
Axes[0] = 0; Axes[1] = 1; Axes[2] = -1;
Center[0] = 1000; Center[1] = 0;
acsc_Arc2(Handle, Axes, Center, -2 * 3.141529, NULL);
// finish the motion
acsc_EndSequenceM(Handle, Axes, NULL);
```

4.23.7 acsc_SmoothTransitionPointToPointMotion

Description

The **acsc_SmoothTransitionPointToPointMotion** command creates motion to a specified target. **acsc_SmoothTransitionPointToPointMotion** commands work in sequence and the next **acsc_SmoothTransitionPointToPointMotion** command changes the previous target and provide a smooth transition from one motion direction to another, based on acceleration, deceleration and jerk values. The motion profile is optimized to pass on a rounded path near the breaking point, minimizing changes in speed and direction that would cause unwanted vibrations in the system.

Syntax

```
int acsc_SmoothTransitionPointToPointMotion(HANDLE Handle, int Flags ,
int* Axes, double* point, double Velocity, double MotionDelay ,ACSC_
WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Flags	Bit-mapped argument that can include one or more of the following flags: ACSC_AMF_VELOCITY(/v) the motion will use velocity specified for each segment instead of the default velocity. ACSC_AMF_LOCALCS(/z) Interpret entered coordinates according to the Local Coordinate System. With this switch, use 2 axes motion coordinate only (X,Y). Using 3 or more coordinates causes a runtime error. ACSC_AMF_EXT_DELAY_MOTION(/q) Defines actual motor movement delay in microseconds. The delay resolution is 50 microseconds. The switch requires additional parameter that specify the motion delay.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to 0, ACSC_AXIS_1 to 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. 2 or 3 Cartesian axes can participate in the motion. For the axis constants see Axis Definitions.
Point	Array of the final point coordinates. The number and order of values must correspond to the Axes array. The Point must specify a value for each element of Axes except the last –1 element.

Velocity	If ACSC_AMF_VELOCITY flag was specified, this argument specifies a motion velocity for current segment. Set this argument to ACSC_NONE if not used.
MotionDelay	If the ACSC_AMF_DELAY_MOTION flag has been specified, this argument defines the bit motion delay. Defines actual motor movement delay in microseconds. The delay resolution is 50 microseconds. The maximum delay is 100 controller cycles or 100ms for CTIME=1ms or 20ms for CTIME=0.2ms. Set this argument to ACSC_NONE if not used.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

Supported from V3.12.

Example

```
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, -1 };
double Point[2] ;
Point[0] = 1000; Point[1] = 1000;

if (!acsc_SmoothTransitionPointToPointMotion(Handle, ACSC_AMF_VELOCITY,
Axes, Point, 1000, ACSC_NONE, ACSC_SYNCHRONOUS)) {
    printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.24 Blended Segmented Motion Functions

The Blended Segmented Motion Functions are:

Function	Description
acsc_ BlendedSegmentMotion	The function initiates a multi-axis blended segmented motion.
acsc_BlendedLine	The function adds a linear segment that starts at the current point and ends at the destination point of segmented motion.
acsc_BlendedArc1	The function adds to the motion path an arc segment that starts at the current point and ends at the destination point with the specified center point.
acsc_BlendedArc2	The function adds an arc segment to a segmented motion and specifies the coordinates of the center point and the rotation angle.

4.24.1 acsc_BlendedSegmentMotion

Description

The function initiates a multi-axis blended segmented motion. Extended segmented motion provides new features:

Syntax

int acsc_BlendedSegmentMotion(HANDLE handle, int Flags, int* Axes, double* Position, double SegmentTime, double AccelerationTime, double JerkTime, double DwellTime, ACSC_WAITBLOCK *Wait)

Arguments

Handle	Communication handle
Flags	Bit-mapped argument that can include one or more of the following flags: ACSC_AMF_WAIT: plan the motion but do not start it until the function acsc_GoM is executed. ACSC_AMF_DWELLTIME: Dwell time between segments. This flag requires an additional parameter that specifies the dwell time, in milliseconds, at the final point of the segment. If this argument is specified, no blending will be done for all segments of the motion. That means that the motion will be

	stopped **at the end of each segment** for the specified ***DwellTime*** milliseconds.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions.
Position	Array of the starting coordinates. The number and order of values must correspond to the **Axes** array. The **Center** must specify a value for each element of the **Axes** except the last –1 element.
Segment Time	This parameter will set the default initial segment time in milliseconds.
AccelerationTime	This parameter will set the default Acceleration time in milliseconds.
JerkTime	This parameter will set the default Jerk time in milliseconds.
DwellTime	If ACSC_AMF_DWELLTIME is set, this parameter will set the initial dwell time between segments in milliseconds. If this argument is specified, no blending will be done for all segments of the motion. That means that the motion will be stopped **at the end of each segment** for the specified ***DwellTime*** milliseconds.

Wait	Wait – Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received If Wait points to a valid ACSC_ WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, a non-zero is returned.

If the function fails, return value is zero.

Extended error information can be obtained by calling acsc_GetLastError.

Comments

Blended segmented motion is a type of segmented motion that doesn't provide look-ahead capabilities, unlike Extended segmented motion. Both type of motions are intended for processing a complex multi-axis trajectory and smoothing corners between segments, but do it in different ways. The Extended segmented motion (XSEG) allows achieving highest throughput within the defined axis limitations and the defined accuracy. The Blended segmented motion (BSEG) allows passing along the trajectory with the defined timing constrains.

The function itself does not specify any movement, so the created motion starts only after the first segment is specified.

The segments of motion are specified by using acsc_BlendedLine, acsc_BlendedArc1, acsc_ BlendedArc2 functions that follow this function.

The motion finishes when the acsc_EndSequenceM function is executed. If the call of **acsc_ EndSequenceM** is omitted, the motion will stop at the last segment of the sequence and wait for the next segment. No transition to the next motion in the motion queue will occur until the function **acsc_EndSequenceM** is executed.

The function can wait for the controller response or can return immediately as specified by the Wait argument.

The controller response indicates that the command was accepted and the motion was planned successfully. The function does not wait and does not validate the end of the motion. To wait for the motion end, use the acsc_WaitMotionEnd function.

If Wait points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the Wait item until a call to the acsc_WaitForAsyncCall function.

Example

```
// Example of the waiting call of acsc_BlendedSegmentedMotion
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, -1 };
double Point[2], Center[2];
// create segmented motion, coordinates of the initial point are (1000,
1000)
Point[0] = 1000; Point[1] = 1000;
If(!acsc_BlendedSegmentMotion(Handle, // Communication Handle
0, //No flags are set
Axes, // Axes 0 and 1
Point, // Starting point of motion
200, // Segment time
30, // Segment Acceleration time
5, // Segment jerk time
ACSC_NONE, // Segment Dwell time is default
ACSC_SYNCHRONOUS)){
printf("transaction error: %d\n", acsc_GetLastError());
}
// add line segment with final point (-1000, -1000)
Point[0] = -1000; Point[1] = -1000;
if (!acsc_BlendedLine(Handle, // Communication handle
0, // Flags are not specified, default parameters
// will be used
Axes, // Axes 0 and 1
Point, // Final point
ACSC_NONE, // Segment time is not specified
ACSC_NONE, // Acceleration time is not specified
ACSC_NONE, // Jerk time is not specified
ACSC_NONE, // Dwell time is not specified
ACSC_SYNCHRONOUS // Waiting call
)){
printf("transaction error: %d\n", acsc_GetLastError());
}
... // Other segments
```

4.24.2 acsc_BlendedLine

Description

The function adds a linear segment that starts at the current point and ends at the destination point of segmented motion.

Syntax

Int acsc_BlendedLine (HANDLE handle, int Flags, int* Axes, double* Point, double SegmentTime, double AccelerationTime, double JerkTime, double DwellTime, ACSC_WAITBLOCK* Wait);

Arguments

Handle	Communication handle
Flags	Bit-mapped argument that can include one or more of the following flags: ACSC_AMF_BSEGTIME: This flag requires an additional parameter that defines the required segment time in milliseconds. ACSC_AMF_BSEGACC: This flag requires an additional parameter that defines the required segment acceleration time in milliseconds. ACSC_AMF_BSEGJERK: This flag requires an additional parameter that defines the required jerk time in milliseconds. ACSC_AMF_DWELLTIME: This flag requires an additional parameter that specifies the dwell time, in milliseconds, at the final point of the segment.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions.
Point	Array of the final point coordinates. The number and order of values must correspond to the **Axes** array. The **Point** must specify a value for each element of **Axes** except the last –1 element.
SegmentTime	If ACSC_AMF_BSEGTIME is set, this parameter will set the segment time, in milliseconds, for the current and all following segments – until the parameter is redefined.
AccelerationTime	If ACSC_AMF_BSEGACC is set, this parameter will set the Acceleration time, in milliseconds, for the current and all following segments – until the parameter is redefined.
JerkTime	If ACSC_AMF_BSEGJERK is set, this parameter will set the default Jerk time, in milliseconds, for the current and all following segments – until the parameter is redefined.
DwellTime	If ACSC_AMF_DWELLTIME is set, this parameter will set the dwell time between segments in milliseconds. If this argument is specified, no blending will be done for all segments of the motion. That means that the motion will be stopped **at the end of each segment** for the specified ***DwellTime*** milliseconds.

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, a non-zero is returned.

If the function fails, return value is zero.

Extended error information can be obtained by calling acsc_GetLastError.

Comments

The function adds a linear segment that starts at the current point and ends at the destination point to segmented motion.

All axes specified in the **Axes** array must be specified in a previous call to the acsc_BlendedSegmentMotion function. The number and order of the axes in the **Axes** array must correspond exactly to the number and order of the axes of the call to the acsc_BlendedSegmentMotion function.

The **Point** argument specifies the coordinates of the final point. The coordinates are absolute in the plane.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the segment is added to the motion buffer. The segment can be rejected if the motion buffer is full. In that case, you can call this function periodically until the function returns a non-zero value.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the**Wait** item until a call to the "acsc_WaitForAsyncCall" on page 53 function.

Example

```
//Example of the waiting call of acsc_BlendedLine
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, -1 };
double Point[2], Center[2];
// create segmented motion, coordinates of the initial point are (1000,
1000)
Point[0] = 1000; Point[1] = 1000;
```

```
If(!acsc_BlendedSegmentMotion(Handle, // Communication Handle
0, //No flags are set
Axes, // Axes 0 and 1
Point, // Starting point of motion
200, // Segment time
30, // Segment Acceleration time
5, // Segment jerk time
ACSC_NONE, // Segment Dwell time is default
ACSC_SYNCHRONOUS)){
printf("transaction error: %d\n", acsc_GetLastError());
}
// add line segment with final point (-1000, -1000)
Point[0] = -1000; Point[1] = -1000;
if (!acsc_BlendedLine(Handle, // Communication handle
ACSC_AMF_BSEGJERK, // BSEGJERK flag is set, JerkTime parameter is
required.
Axes, // Axes 0 and 1
Point, // Final point
ACSC_NONE, // Segment time is not specified
ACSC_NONE, // Acceleration time is not specified
6, // JerkTime is specified, will act as the new default.
ACSC_NONE, // DwellTime is not specified
ACSC_SYNCHRONOUS // Waiting call
)){
printf("transaction error: %d\n", acsc_GetLastError());
}
// finish the motion
acsc_EndSequenceM(Handle, Axes, NULL);
```

4.24.3 acsc_BlendedArc1

Description

The function adds to the motion path an arc segment that starts at the current point and ends at the destination point with the specified center point.

Syntax

Int acsc_BlendedArc1 (HANDLE handle, int Flags, int* Axes, double* Center, double* FinalPoint, int Rotation, double SegmentTime, double AccelerationTime, double JerkTime, double DwellTime, ACSC_WAITBLOCK* Wait);

Arguments

Handle	Communication Table
Flags	Bit-mapped argument that can include one or more of the following flags: ACSC_AMF_BSEGTIME: This flag requires an additional parameter that defines the required segment time in milliseconds. ACSC_AMF_BSEGACC: This flag requires an additional parameter

	that defines the required segment acceleration time in milliseconds. ACSC_AMF_BSEGJERK: This flag requires an additional parameter that defines the required jerk time in milliseconds. ACSC_AMF_DWELLTIME: This flag requires an additional parameter that specifies the dwell time, in milliseconds, at the final point of the segment.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions.
Center	Array of the center coordinates. The number and order of values must correspond to the **Axes** array. The **Center** must specify a value for each element of the **Axes** except the last –1 element.
FinalPoint	Array of the final point coordinates. The number and order of values must correspond to the **Axes** array. The **Point** must specify a value for each element of **Axes** except the last –1 element.
Rotation	This argument defines the direction of rotation. If **Rotation** is set to ACSC_COUNTERCLOCKWISE, then the rotation is counterclockwise. If **Rotation** is set to ACSC_CLOCKWISE, then rotation is clockwise.
SegmentTime	If ACSC_AMF_BSEGTIME is set, this parameter will set the segment time, in milliseconds, for the current and all following segments – until the parameter is redefined.
AccelerationTime	If ACSC_AMF_BSEGACC is set, this parameter will set the Acceleration time, in milliseconds, for the current and all following segments – until the parameter is redefined.
JerkTime	If ACSC_AMF_BSEGJERK is set, this parameter will set the default Jerk time, in milliseconds, for the current and all following segments – until the parameter is redefined.
DwellTime	If ACSC_AMF_DWELLTIME is set, this parameter will set the dwell time between segments in milliseconds. If this argument is specified, no blending will be done for all segments of the motion. That means that the motion will be stopped **at the end of each segment** for the specified ***DwellTime*** milliseconds.

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the "acsc_WaitForAsyncCall" on page 53 function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, a non-zero is returned.

If the function fails, return value is zero.

Extended error information can be obtained by calling acsc_GetLastError.

Comments

All axes specified in the **Axes** array must be specified in a previous call to the acsc_BlendedSegmentMotion function. The number and order of the axes in the **Axes** array must correspond exactly to the number and order of the axes of the call to the acsc_BlendedSegmentMotion function.

The **FinalPoint** argument specifies the coordinates of the final point. The coordinates are absolute in the plane.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the segment is added to the motion buffer. The segment can be rejected if the motion buffer is full. In that case, you can call this function periodically until the function returns a non-zero value.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
//Example of the waiting call of acsc_BlendedArc1
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, -1 };
double Point[2], Center[2];
// create segmented motion, coordinates of the initial point are (1000,
1000)
Point[0] = 1000; Point[1] = 1000;
CntrPnt[0] = 0; CntrPnt[1] = 0;
If(!acsc_BlendedSegmentMotion(Handle, // Communication Handle
0, //No flags are set
```

```
Axes, // Axes 0 and 1
Point, // Starting point of motion
200, // Segment time
30, // Segment Acceleration time
5, // Segment jerk time
ACSC_NONE, // Segment Dwell time is default
ACSC_SYNCHRONOUS)){
printf("transaction error: %d\n", acsc_GetLastError());
}
// add line segment with final point (-1000, -1000)
Point[0] = -1000; Point[1] = -1000;
if (!acsc_BlendedArc1(Handle, // Communication handle
ACSC_AMF_DWELLTIME, // DwellTime parameter is now required.
Axes, // Axes 0 and 1
CntrPnt, // Center point
Point, // Final point
ACSC_COUNTERCLOCKWISE, // Positive rotation
ACSC_NONE, // Segment time is not specified
ACSC_NONE, // Acceleration time is not specified
ACSC_NONE, // Jerk time is not specified.
10, // Dwell time at the end of segment is set.
ACSC_SYNCHRONOUS // Waiting call
)){
printf("transaction error: %d\n", acsc_GetLastError());
}
// finish the motion
acsc_EndSequenceM(Handle, Axes, NULL);
```

4.24.4 acsc_BlendedArc2

Description

The function adds an arc segment to a segmented motion and specifies the coordinates of the center point and the rotation angle.

Syntax

Int acsc_BlendedArc2 (HANDLE handle, int Flags, int* Axes, double* Center, double Angle, double SegmentTime, double AccelerationTime, double JerkTime, double DwellTime, ACSC_WAITBLOCK* Wait);

Arguments

Handle	communication handle
Flags	Bit-mapped argument that can include one or more of the following flags: ACSC_AMF_BSEGTIME: This flag requires an additional parameter that defines the required segment time in milliseconds.

	ACSC_AMF_BSEGACC: This flag requires an additional parameter that defines the required segment acceleration time in milliseconds. ACSC_AMF_BSEGJERK: This flag requires an additional parameter that defines the required jerk time in milliseconds. ACSC_AMF_DWELLTIME: This flag requires an additional parameter that specifies the dwell time, in milliseconds, at the final point of the segment.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions.
Center	Array of the center coordinates. The number and order of values must correspond to the**Axes** array. The **Center** must specify a value for each element of the **Axes** except the last –1 element.
Angle	Rotation angle in radians. Positive angle for counterclockwise rotation, negative for clockwise rotation.
SegmentTime	If ACSC_AMF_BSEGTIME is set, this parameter will set the segment time, in milliseconds, for the current and all following segments – until the parameter is redefined.
AccelerationTime	If ACSC_AMF_BSEGACC is set, this parameter will set the Acceleration time, in milliseconds, for the current and all following segments – until the parameter is redefined.
JerkTime	If ACSC_AMF_BSEGJERK is set, this parameter will set the default Jerk time, in milliseconds, for the current and all following segments – until the parameter is redefined.
DwellTime	If ACSC_AMF_DWELLTIME is set, this parameter will set the dwell time between segments in milliseconds. If this argument is specified, no blending will be done for all segments of the motion. That means that the motion will be stopped at the end of each segment for the specified DwellTime milliseconds.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result.

	If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, a non-zero is returned.

If the function fails, return value is zero.

Extended error information can be obtained by calling acsc_GetLastError.

Comments

All axes specified in the **Axes** array must be specified in a previous call to the acsc_BlendedSegmentMotion function. The number and order of the axes in the **Axes** array must correspond exactly to the number and order of the axes of the call to the acsc_BlendedSegmentMotion function.

The **Center** argument specifies the coordinates of the Center point. The coordinates are absolute in the plane.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the segment is added to the motion buffer. The segment can be rejected if the motion buffer is full. In that case, you can call this function periodically until the function returns a non-zero value.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the "acsc_WaitForAsyncCall" on page 53 function.

Example

```
//Example of the waiting call of acsc_BlendedArc2
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, -1 };
double Point[2], Center[2];
// create segmented motion, coordinates of the initial point are (1000,
1000)
Point[0] = 1000; Point[1] = 1000;
CntrPnt[0] = 1000; CntrPnt[1] = 0;
If(!acsc_BlendedSegmentMotion(Handle, // Communication Handle
0, //No flags are set
Axes, // Axes 0 and 1
Point, // Starting point of motion
200, // Segment time
30, // Segment Acceleration time
5, // Segment jerk time
ACSC_NONE, // Segment Dwell time is default
ACSC_SYNCHRONOUS)){
```

```
printf("transaction error: %d\n", acsc_GetLastError());
}
// add an arc2 full circular movement.
if (!acsc_BlendedArc2(Handle, // Communication handle
ACSC_AMF_DWELLTIME, // DwellTime parameter is now required.
Axes, // Axes 0 and 1
CntrPnt, // Center point
-3.14, // Rotation angle, counter clockwise
ACSC_NONE, // Segment time is not specified
ACSC_NONE, // Acceleration time is not specified
ACSC_NONE, // Jerk time is not specified.
10, // Dwell time at the end of segment is set.
ACSC_SYNCHRONOUS // Waiting call
)){
printf("transaction error: %d\n", acsc_GetLastError());
}
// finish the motion
acsc_EndSequenceM(Handle, Axes, NULL);
```

4.25 NURBS and Smooth Path Motion

Function	Description
acsc_NurbsMotion	Creates NURBS motion
acsc_NurbsPoint	Adds a new control point to the NURBS motion generator.
acsc_SmoothPathMotion	Smooth path is a motion type that accepts line segments and generates a spline motion between the specified points.
acsc_ SmoothPathSegment	Adds a new segment to the Smooth Path Motion generator.

4.25.1 acsc_NurbsMotion

Description

The function creates NURBS motion. NURBS is a motion generator based on the NURBS spline specification. The algorithm accepts as parameters points, weights, and knots, and generates a spline trajectory accordingly.

Syntax

```
int acsc_NurbsMotion(HANDLE Handle, int Flags, int* Axes , double
Velocity, double ExceptionAngle, double ExceptionLength, double
MotionDelay, char* Segments, ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Flags	Bit-mapped argument that can include one or more of the following flags: ACSC_AMF_WAIT plan the motion but do not start it until the function acsc_GoM is executed. ACSC_AMF_VELOCITY the motion will use the velocity specified for each segment instead of the default velocity. ACSC_AMF_EXT_DELAY_MOTION Defines actual motor movement delay in microseconds. The delay resolution is 50 microseconds. The switch requires additional parameter that specifies the motion delay. ACSC_AMF_NURBS_CONSIDER_ACC Acceleration consideration. Allow the MG to deviate from specified axes acceleration parameter during velocity profile generation. ACSC_AMF_NURBS_EXCEPTION_ANGLE The suffix requires additional parameter that specifies maximum angle in a control point. The value defines exceptions from spline interpolation. If for an internal control point, directions to the previous and the next control points require direction change more than the specified angle (by modulo), the control point is processed as a corner. Actually, defining such a point divides the spline into two independent splines. ACSC_AMF_NURBS_EXCEPTION_LENGTH Specify exception length. The suffix requires an additional parameter that specifies maximum segment length. The value defines exceptions from spline interpolation. If a distance between two control points appears longer than the specified length, the trajectory between the points is considered straight. Actually, two independent splines are built before and after the segment.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions.

Velocity	If ACSC_AMF_VELOCITY flag was specified, this argument specifies a motion velocity for current segment. Set this argument to ACSC_NONE if not used.
ExceptionAngle	If ACSC_AMF_NURBS_EXCEPTION_ANGLE flag was specified, this argument defines ExceptionAngle. Set this argument to ACSC_NONE if not used.
ExceptionLength	If ACSC_AMF_ NURBS_EXCEPTION_LENGTH flag was specified, this argument defines ExceptionLength. Set this argument to ACSC_NONE if not used.
MotionDelay	If ACSC_AMF_DELAY_MOTION flag has been specified, this argument defines the bit motion delay. Defines actual motor movement delay in microseconds. The delay resolution is 50 microseconds. The maximum delay is 100 controller cycles or 100ms for CTIME=1ms or 20ms for CTIME=0.2ms. Set this argument to ACSC_NONE if not used
Segments	Pointer to the null-terminated character string that contains the name of a one-dimensional user-defined array used to store added segments. Set this argument to NULL if not used. By default, if this argument is not specified, the controller allocates internal buffer for storing 50 segments only. The argument allows the user application to reallocate the buffer for storing a larger number of segments. The larger number of segments may be required if the application needs to add many very small segments in advanced. For most applications, the internal buffer size is enough and should not be enlarged. The buffer is for the controller internal use only and should not be used by the user application. The buffer size calculation rule: each segment requires about 600 bytes, so if it is necessary to allocate the buffer for 200 segments, it should be at least 600 * 200 = 120,000 bytes. The following declaration defines a 120,000 bytes buffer: real buf(15000) See XARRSIZE explanation in the ACSPL+ Command and Variable Reference Guide for details on how to declare a buffer with more than 100,000 elements.

Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

Supported from V3.12.

Example

```
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, -1 };
double Point[2[={ 0.0,0.0 };

if (!acsc_NurbsMotion(controller, ACSC_AMF_VELOCITY, Axes, 2000, ACSC_
NONE, ACSC_NONE, ACSC_NONE, NULL, ACSC_SYNCHRONOUS)) {
        printf("transaction error: %d\n", acsc_GetLastError());
    }
acsc_EndSequenceM(controller, Axes, 0);
```

4.25.2 acsc_NurbsPoint

Description

The function adds a new control point to the NURBS motion generator.

Syntax

```
int acsc_NurbsPoint(HANDLE Handle, int Flags , int* Axes, double* Point,
double Velocity, double Required_vel, double Knot, double Weight, ACSC_
WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle
	Bit-mapped argument that may include one or more of the following flags: ACSC_AMF_VELOCITY The motion will use velocity specified for each segment instead of the

Flags	default velocity. ACSC_AMF_REQUIRED_VELOCITY Specify required velocity. The suffix is not compatible with ACSC_AMF_VELOCITY. The flag requires a non-zero value in the parameter that specifies required velocity. The value is considered required velocity at the current point, but does not change required velocity for subsequent points. ACSC_AMF_CORNER Mark the current point as a corner. The control point is processed as a corner. Actually, such point divides the spline into two independent splines. ACSC_AMF_DUMMY Mark the point specification as dummy. Dummy point specifications can either precede the first control point specification, or follow the last control point specification. Dummy points specification is required in rare cases where default calculation of starting/trailing knots is not suitable (see Dummy point specification) ACSC_AMF_WEIGHT Specify control point weight. The suffix requires additional parameter that specifies the weight of the control point. ACSC_AMF_KNOT Specify knot delta. The suffix requires additional parameter that specifies knot delta from the previous knot. The new knot is calculated as the previous knot plus delta.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be included that contains –1, marking the end of the array. For the axis constants see Axis Definitions.
Point	Array of the final point coordinates. The number and order of values must correspond to the Axes array. The Point must specify a value for each element of Axes except the last –1 element.
Velocity	If ACSC_AMF_VELOCITY flag was specified, this argument specifies a motion velocity for current segment. Set this argument to ACSC_NONE if not used.
Required_vel	If ACSC_AMF_REQUIRED_VELOCITY flag was specified, this argument specifies a Required velocity in the current point. Set this argument to ACSC_NONE if not used.

Knot	If ACSC_AMF_ KNOT flag was specified, this argument specifies a knot delta from the previous knot. Set this argument to ACSC_NONE if not used.
Weight	If ACSC_AMF_ WEIGHT flag has been specified, this argument defines the weight of the control point. Set this argument to ACSC_NONE if not used
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling acsc_GetLastError.

Comments

Supported from V3.12.

Example

```
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, -1 };
    double point[2[={ 0.0,0.0 };

if (!acsc_NurbsMotion(controller, ACSC_AMF_VELOCITY, Axes, 2000, ACSC_
NONE, ACSC_NONE, ACSC_NONE, NULL, ACSC_SYNCHRONOUS)) {
        printf("transaction error: %d\n", acsc_GetLastError());
    }

    double point_Square[9][2] = { { 0, 0 }, { 50, 0 }, { 100, 0 }, { 100,
50 },
    { 100, 100 }, { 50, 100 }, { 0, 100 }, { 0, 50 }, { 0, 0 }, };


    for (int i = 0; i < 9; i++) {
        point[0] = point_Square[i][0];
        point[1] = point_Square[i][1];
        unsigned int flags = 0;
        double vel = ACSC_NONE;
```

```
        if (i == 2) {
            flags = ACSC_AMF_VELOCITY;
            vel = 1000;
    }



if (!acsc_NurbsPoints(controller, flags, Axes, point, vel, ACSC_NONE,
ACSC_NONE, ACSC_NONE, ACSC_SYNCHRONOUS)) {
            printf("transaction error: %d\n", acsc_GetLastError());
     }

}
        acsc_EndSequenceM(controller, Axes, 0);
```

4.25.3 acsc_SmoothPathMotion

Description

Smooth path is a motion type that accepts line segments and generates a spline motion between the specified points.

The motion deviates from exact coordinates after interpolation.

To control the deviation, the user can generate more points along the trajectory or use the Corner specification.

Syntax

```
int acsc_SmoothPathMotion(HANDLE Handle, int Flags, int* Axes , double*
Point, double Velocity, double ExceptionAngle, double ExceptionLength,
double MotionDelay, char* Segments, ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Flags	Bit-mapped argument that can include one or more of the following flags: ACSC_AMF_WAIT Plan the motion but do not start it until the function acsc_GoM is executed. ACSC_AMF_VELOCITY The motion will use velocity specified for each segment instead of the default velocity. ACSC_AMF_DELAY_MOTION Defines actual motor movement delay in microseconds. The delay resolution is 50 microseconds.

	The switch requires additional parameter that specifies the motion delay. ACSC_AMF_NURBS_CONSIDER_ACCELERATION Acceleration consideration. Allow the motion generator to deviate from specified axes acceleration parameter during velocity profile generation. ACSC_AMF_NURBS_EXCEPTION_ANGLE The suffix requires additional parameter that specifies maximum angle in a control point. The value defines exceptions from spline interpolation. If for an internal control point, directions to the previous and the next control points require direction change more than the specified angle (by modulo), the control point is processed as a corner. Actually, such point divides the spline into two independent splines. ACSC_AMF_NURBS_EXCEPTION_LENGTH(/l) Specify exception length. The suffix requires additional parameter that specifies maximum segment length. The value defines exceptions from spline interpolation. If a distance between two control points appears longer than the specified length, the trajectory between the points is considered straight. Actually, two independent splines are built before and after the segment.
Point	Array of the final point coordinates. The number and order of values must correspond to the Axes array. The Point must specify a value for each element of Axes except the last –1 element.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions.
Velocity	If ACSC_AMF_VELOCITY flag was specified, this argument specifies a motion velocity for current segment. Set this argument to ACSC_NONE if not used.
ExceptionAngle	If ACSC_AMF_NURBS_EXCEPTION_ANGLE flag was specified, this argument defines ExceptionAngle. Set this argument to ACSC_NONE if not used.
ExceptionLength	If ACSC_AMF_ NURBS_EXCEPTION_LENGTH flag was specified, this argument defines ExceptionLength. Set this argument to ACSC_NONE if not used.

MotionDelay	If ACSC_AMF_DELAY_MOTION flag has been specified, this argument defines the bit motion delay. Defines actual motor movement delay in microseconds. The delay resolution is 50 microseconds. The maximum delay is 100 controller cycles or 100ms for CTIME=1ms or 20ms for CTIME=0.2ms. Set this argument to ACSC_NONE if not used
Segments	Pointer to the null-terminated character string that contains the name of a one-dimensional user-defined array used to store added segments. Set this argument to NULL if not used. By default, if this argument is not specified, the controller allocates internal buffer for storing 50 segments only. The argument allows the user application to reallocate the buffer for storing a larger number of segments. The larger number of segments may be required if the application needs to add many very small segments in advanced. For most applications, the internal buffer size is enough and should not be enlarged. The buffer is for the controller internal use only and should not be used by the user application. The buffer size calculation rule: each segment requires about 600 bytes, so if it is necessary to allocate the buffer for 200 segments, it should be at least 600 * 200 = 120,000 bytes. The following declaration defines a 120,000 bytes buffer: real buf(15000) See XARRSIZE explanation in the ACSPL+ Command and Variable Reference Guide for details on how to declare a buffer with more than 100,000 elements.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero

If the function fails, the return value is zero.

Extended error information can be obtained by calling acsc_GetLastError.

Comments

Supported from V3.12.

Example

```
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, -1 };
double Point[2[={ 0.0,0.0 };

if (!acsc_SmoothPathMotion(controller, ACSC_AMF_VELOCITY, Axes, point,
   1000, ACSC_NONE, ACSC_NONE, ACSC_NONE, NULL, ACSC_SYNCHRONOUS)) {
        printf("transaction error: %d\n", acsc_GetLastError());
    }
acsc_EndSequenceM(controller, Axes, 0);
```

4.25.4 acsc_SmoothPathSegment

Description

Smooth path is a motion type that accepts line segments and generates a spline motion between the specified points.

The motion deviates from exact coordinates after interpolation.

To control the deviation, the user can generate more points along the trajectory or use the Corner specification.

Syntax

```
int acsc_SmoothPathMotion(HANDLE Handle, int Flags, int* Axes , double*
Point, double Velocity, double ExceptionAngle, double ExceptionLength,
double MotionDelay, char* Segments, ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Flags	Bit-mapped argument that can include one or more of the following flags: ACSC_AMF_WAIT: plan the motion but do not start it until the function acsc_GoM is executed. ACSC_AMF_VELOCITY: the motion will use velocity specified for each segment instead of the default velocity. ACSC_AMF_ENDVELOCITY: This flag requires additional parameter that specifies end velocity. The controller decelerates to the specified velocity in the end of segment. The specified value should be less than the required velocity; otherwise the parameter is ignored. This flag affects only one segment. This flag also disables corner detection and processing at the end of segment.

If this flag is not specified, deceleration is not required. However, in special cases the deceleration might occur due to corner processing or other velocity control conditions.

ACSC_AMF_MAXIMUM: use maximum velocity under axis limits. With this suffix, no required velocity should be specified.

The required velocity is calculated for each segment individually on the base of segment geometry and axis velocities (VEL values) of the involved axes.

ACSC_AMF_JUNCTIONVELOCITY: Decelerate to corner.

This flag requires additional parameter that specifies corner velocity. The controller detects corner on the path and decelerates to the specified velocity before the corner. The specified value should be less than the required velocity; otherwise the parameter is ignored.

If ACSC_AMF_JUNCTIONVELOCITY flag is not specified while ACSC_AMF_ ANGLE flag is specified, zero value of corner velocity is assumed.

If neither the ACSC_AMF_JUNCTIONVELOCITY nor the ACSC_ AMF_ANGLE flags are specified, the controller provides automatic calculation as described in Automatic corner processing.

ACSC_AMF_ANGLE: Do not treat junction as a corner, if junction angle is less than or equal to the specified value in radians. This flag requires additional parameter that specifies a negligible angle in radians.

If ACSC_AMF_ANGLE flag is not specified while ACSC_AMF_ JUNCTIONVELOCITY flag is specified, the controller accepts default value of 0.01 radians, about 0.57 degrees.

If neither the ACSC_AMF_JUNCTIONVELOCITY nor the ACSC_ AMF_ANGLE flags are specified, the controller provides automatic calculation as described in Automatic corner processing.

ACSC_AMF_AXISLIMIT

Enable velocity limitations under axis limits.

With this flag set, setting the ACSC_AMF_VELOCITY flag will result in the requested velocity being restrained by the velocity limits of all involved axes.

ACSC_AMF_CURVEVELOCITY

Decelerate to curvature discontinuity point.

This flag requires an additional parameter that specifies velocity at curvature discontinuity points.

Curvature discontinuity occurs in linear-to-arc or arc-to-arc smooth junctions.

If the flag is not set, the controller does not decelerate to smooth junction disregarding curvature discontinuity in the junction.

	If the flag is set, the controller detects curvature discontinuity points on the path and provides deceleration to the specified velocity. The specified value should be less than the required velocity; otherwise the parameter is ignored. The flag can be set together with flags ACSC_AMF_ JUNCTIONVELOCITY and/or ACS_AMF_ANGLE. If neither of ACSC_AMF_JUNCTIONVELOCITY, ACS_AMF_ ANGLE or ACSC_ AMF_CURVEVELOCITY is set, the controller provides automatic calculation of the corner processing. ACSC_AMF_CURVEAUTO If the Flag is specified the controller provides automatic calculations as described in Enhanced automatic corner and curvature discontinuity points processing. ACSC_AMF_CORNERDEVIATION Use a corner rounding option with the specified permitted deviation. This flag requires an additional parameter that specifies maximal allowed deviation of motion trajectory
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_ AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains – 1 which marks the end of the array. For the axis constants see Axis Definitions.
Point	Array of the final point coordinates. The number and order of values must correspond to the Axes array. The Point must specify a value for each element of Axes except the last –1 element.
Velocity	If ACSC_AMF_VELOCITY flag was specified, this argument specifies a motion velocity for current segment. Set this argument to ACSC_NONE if not used.
Required_vel	If ACSC_AMF_REQUIRED_VELOCITY flag was specified, this argument specifies a Required velocity in the current point. Set this argument to ACSC_NONE if not used.

Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

Supported from V3.12.

Example

```
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, -1 };
double Point[2]={0.0,0.0}

if (!acsc_SmoothPathMotion(controller, ACSC_AMF_VELOCITY, Axes, point,
1000, ACSC_NONE, ACSC_NONE, ACSC_NONE, NULL, ACSC_SYNCHRONOUS)) {

        printf("transaction error: %d\n", acsc_GetLastError());
}

double point_Square[8][2] = { { 50, 0 }, { 100, 0 }, { 100, 50 }, { 100,
100 },
{ 50, 100 }, { 0, 100 }, { 0, 50 }, { 0, 0 }, };

for (int i = 0; i < 8; i++) {
if (!acsc_SmoothPathSegment(controller, 0, Axes, point_Square[i], ACSC_
NONE, ACSC_NONE, ACSC_SYNCHRONOUS)) {
        printf("transaction error: %d\n", acsc_GetLastError());
    }
}

acsc_EndSequenceM(controller, Axes, 0);
```

4.26 Points and Segments Manipulation Functions

The Points and Segments Manipulation functions are:

Table 5-23. Points and Segments Manipulation Functions

Function	Description
acsc_AddPoint	Adds a point to a single-axis multi-point or spline motion.
acsc_AddPointM	Adds a point to a multi-axis multi-point or spline motion.
acsc_ExtAddPoint	Adds a point to a single-axis multi-point or spline motion and specifies a specific velocity or motion time.
acsc_ExtAddPointM	Adds a point to a multi-axis multi-point or spline motion and specifies a specific velocity or motion time.
acsc_EndSequence	Informs the controller that no more points will be specified for the current single-axis motion.
acsc_ EndSequenceM	Informs the controller that no more points or segments will be specified for the current multi-axis motion.

4.26.1 acsc_AddPoint

Description

The function adds a point to a single-axis multi-point or spline motion.

Syntax

int acsc_AddPoint(HANDLE Handle, int Axis, double Point,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
Point	Coordinate of the added point.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function adds a point to a single-axis multi-point or spline motion. To add a point to a multi-axis motion, use acsc_AddPVPointM. To add a point with a specified non-default velocity or time interval use acsc_AddPVPoint or acsc_AddPVPointM.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the point is added to the motion buffer. The point can be rejected if the motion buffer is full. In this case, you can call this function periodically until the function returns non-zero value.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_AddPoint
int i;
acsc_MultiPoint(Handle, 0, 0, 1, NULL));        // create multi-point motion
// add some points
for (i = 0; i < 5; i++)
{
if (!acsc_AddPoint(Handle, // communication handle
                                ACSC_AXIS_0,    // axis 0
                                1000 * i,       // points 1000, 2000, 3000, …
                                NULL            // waiting call
                                ))
{
                printf("transaction error: %d\n",acsc_GetLastError());
                break;
        }
}
// finish the motion
acsc_EndSequence(Handle, 0, NULL);              // end of the multi-point motion
```

4.26.2 acsc_AddPointM

Description

The function adds a point to a multi-axis multi-point or spline motion.

Syntax

int acsc_AddPointM(HANDLE Handle, int* Axes, double* Point,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions.
Point	Array of the coordinates of added point. The number and order of values must correspond to the **Axes** array. The **Point** must specify a value for each element of **Axes** except the last –1 element.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function adds a point to a multi-axis multi-point or spline motion. To add a point to a single-axis motion, use acsc_AddPoint. To add a point with a specified non-default velocity or time interval use acsc_ExtAddPoint or acsc_ExtAddPointM.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the point is added to the motion buffer. The point can be rejected if the motion buffer is full. In this case, you can call this function periodically until the function returns non-zero value.

All axes specified in the **Axes** array must be specified before the call of the acsc_MultiPointM or acsc_SplineM function. The number and order of the axes in the **Axes** array must correspond exactly to the number and order of the axes of acsc_MultiPointM or acsc_SplineM functions.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_AddPointM
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, -1 };
int Points[2];
int i;
acsc_MultiPointM(Handle, 0, Axes, 0, NULL));    // create multi-point
motion
// add some points
for (i = 0; i < 5; i++)
{
        Points[0] = 1000 * i; Points[1] = 1000 * i;
        // points (1000, 1000), (2000, 2000)…
        if (!acsc_AddPointM(Handle, Axes, Points, NULL))
        {
                printf("transaction error: %d\n", acsc_GetLastError());
                break;
        }
}
// finish the motion
acsc_EndSequenceM(Handle, Axes, NULL);  // the end of the multi-point
motion
```

4.26.3 acsc_ExtAddPoint

Description

The function adds a point to a single-axis multi-point or spline motion and specifies a specific velocity or motion time.

Syntax

int acsc_ExtAddPoint(HANDLE Handle, int Axis, double Point, double Rate,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
Point	Coordinate of the added point.
Rate	If the motion was activated by the acsc_MultiPoint function with the ACSC_AMF_VELOCITY flag, this parameter defines the motion velocity. If the motion was activated by the acsc_Spline function with the ACSC_AMF_VARTIME flag, this parameter defines the time interval between the previous point and the present one.

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function adds a point to a single-axis multi-point motion with specific velocity or to single-axis spline motion with a non-uniform time.

To add a point to a multi-axis motion, use acsc_ExtAddPointM. To add a point to a motion with default velocity or uniform time interval, the acsc_AddPoint and acsc_AddPointM functions are more convenient.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the point is added to the motion buffer. The point can be rejected if the motion buffer is full. In this case, you can call this function periodically until the function returns a non-zero value.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_ExtAddPoint
int i;
// create multi-point motion with the specific velocity
acsc_MultiPoint(Handle, ACSC_AMF_VELOCITY, ACSC_AXIS_0, 1, NULL);
// add some points
for (i = 0; i < 5; i++)
{
if (!acsc_ExtAddPoint(                  Handle,                 // communication handl
                        ACSC_AXIS_0,    // axis 0
                        1000 * i,               // points 1000, 2000, 3000, …
                        5000,           // in this case Rate defines
                                        // a motion velocity
```

```
                        NULL            // waiting call
                        ))
        {
                printf("transaction error: %d\n", acsc_GetLastError());
                break;
        }
}
// finish the motion
acsc_EndSequence(Handle, ACSC_AXIS_0, NULL);    // end of multi-point motion
```

4.26.4 acsc_ExtAddPointM

Description

The function adds a point to a multi-axis multi-point or spline motion and specifies a specific velocity or motion time.

Syntax

int acsc_ExtAddPointM(HANDLE Handle, int* Axes, double* Point, double Rate,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions.
Point	Array of the coordinates of added point. The number and order of values must correspond to the **Axes** array. The **Point** must specify a value for each element of **Axes** except the last –1 element.
Rate	If the motion was activated by the acsc_MultiPoint function with the ACSC_AMF_VELOCITY flag, this parameter defines as motion velocity. If the motion was activated by the acsc_Spline function with the ACSC_AMF_VARTIME flag, this parameter defines as time interval between the previous point and the present one.

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function adds a point to a multi-axis multi-point or spline motion. To add a point to a single-axis motion, use acsc_ExtAddPoint. To add a point to a motion with a default velocity or a uniform time interval, the **acsc_ExtAddPointM** function is more convenient.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

The controller response indicates that the command was accepted and the point is added to the motion buffer. The point can be rejected if the motion buffer is full. In this case, you can call this function periodically until the function returns non-zero value.

All axes specified in the **Axes** array must be specified before the call of the acsc_MultiPointM or acsc_SplineM function. The number and order of the axes in the **Axes** array must correspond exactly to the number and order of the axes of **acsc_MultiPointM** or **acsc_SplineM** functions.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_ExtAddPointM
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, -1 };
int Points[2];
int i;
// create multi-point motion with specific velocity
acsc_MultiPointM(Handle, ACSC_AMF_VELOCITY, Axes, 0, NULL));
// add some points
for (i = 0; i < 5; i++)
{
Points[0] = 1000 * i; Points[1] = 1000 * i;
        if (!acsc_ExtAddPointM(Handle, // communication handle
```

```
                        Axes,           // axes 0 and 1
                        Points,         // points (1000,1000),
                                        // (2000,2000), …
                        5000,           // in this case Rate defines
                                        // a motion velocity
                        NULL
                        ))
        {
                printf("transaction error: %d\n", acsc_GetLastError());
                break;
        }
}
// finish the motion
acsc_EndSequenceM(Handle, Axes, NULL);  // the end of the multi-point
motion
```

4.26.5 acsc_EndSequence

Description

The function informs the controller, that no more points will be specified for the current single-axis motion.

Syntax

int acsc_EndSequence(HANDLE Handle, int Axis, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The motion finishes when the **acsc_EndSequence** function is executed. If the call of **acsc_EndSequence** 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 motion queue will occur until the **acsc_EndSequence** function executes.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

This function applies to the single-axis multi-point or spline (arbitrary path) motions.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_EndSequence
int i;
// create multi-point motion
acsc_MultiPoint(Handle, 0, ACSC_AXIS_0, 1, NULL);
// add some points
for (i = 0; i < 5; i++)
{
acsc_AddPoint(Handle, ACSC_AXIS_0, 1000 * i, NULL);
}
// end of the multi-point motion
if (!acsc_EndSequence(  Handle,          // communication handle
                        ACSC_AXIS_0,     // axis 0
                        NULL             // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.26.6 acsc_EndSequenceM

Description

The function informs the controller, that no more points or segments will be specified for the current multi-axis motion.

Syntax

int acsc_EndSequence(HANDLE Handle, int* Axes, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The motion finishes when the **acsc_EndSequenceM** function is executed. If the call of **acsc_EndSequenceM** is omitted, the motion will stop at the last point or segment of the sequence and wait for the next point. No transition to the next motion in the motion queue will occur until the **acsc_EndSequenceM** function executes.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

This function applies to the multi-axis multi-point, spline (arbitrary path) and segmented motions.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_EndSequenceM
int Axes[] = { ACSC_AXIS_0, ACSC_AXIS_1, -1 };
int Points[2];
int i;
// create multi-point motion
acsc_MultiPointM(Handle, 0, Axes, 0, NULL);
```

```
// add some points
for (i = 0; i < 5; i++)
{
Points[0] = 1000 * i; Points[1] = 1000 * i;
// points (1000, 1000), (2000, 2000), …
acsc_AddPointM(Handle, Axes, Points, NULL);
}
// the end of the multi-point motion
if (!acsc_EndSequenceM( Handle,          // communication handle
                        Axes,            // axes 0 and 1
                        NULL             // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
```

4.27 Data Collection Functions

The Data Collection functions are:

Table 5-24. Data Collection Functions

Function	Description
acsc_DataCollectionExt	Initiates data collection.
acsc_StopCollect	Terminates data collection.
acsc_WaitCollectEndExt	Wait for the end of data collection.

4.27.1 acsc_DataCollectionExt

Description

The function initiates data collection.

This function replaces **csc_DataCollection** and which is now obsolete.

Syntax

int acsc_DataCollectionExt(HANDLE Handle, int Flags, int Axis, char*
Array, int NSample,double Period, char* Vars, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Flags	Bit-mapped parameter that can include one or more of the following flags: ACSC_DCF_SYNC: Start data collection synchronously to a motion.

	ACSC_DCF_WAIT: Create the synchronous data collection, but do not start until the acsc_Go function is called. This flag can only be used with the ACSC_DCF_SYNC flag. ACSC_DCF_TEMPORAL: Temporal data collection, the sampling period is calculated automatically according to the collection time. ACSC_DCF_CYCLIC: Cyclic data collection uses the collection array as a cyclic buffer and continues indefinitely. When the array is full, each new sample overwrites the oldest sample in the array.
Axis	Axis constant of the axis to which the data collection must be synchronized. The parameter is required only for axis data collection (ACSC_DCF_SYNC flag). For the axis constants see Axis Definitions
Array	Pointer to the null-terminated string contained the name of the array that stores the collected samples. The array must be declared as a global variable by an ACSPL+ program or by the acsc_DeclareVariable function.
NSample	Number of samples to be collected.
Period	Sampling period in milliseconds. If the ACSC_DCF_TEMPORAL flag is specified, this argument defines a minimal period.
Vars	Variable list - Pointer to null terminated string. The string contains chained names of the variables, separated by '\r'(13) character. The values of these variables will be collected in the Array. If variable name specifies an array, the name must be supplemented with indexes in order to specify one element of the array.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

Data collection started by this function without the ACSC_DCF_SYNC flag is called system data collection. Data collection started with the ACSC_DCF_SYNC flag is called axis data collection. Data collection started with the ACSC_DCF_CYCLIC flag 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 calling acsc_StopCollect function.

The array that stores the samples can be one or two-dimensional. A one-dimensional array is allowed only if the variable list contains one variable name.

The number of the array rows must be equal to or more than the number of variables in the variable list. The number of the array columns must be equal to or more than the number of samples specified by the **NSample** argument.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the Wait item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_DataCollection
// matrix consisting of two rows with 1000 columns each
char* ArrayName = "DCA(2)(1000)";
// positions of axes X and Y will be collected
char Vars[] ="FPOS(0)\rFPOS(1)";
acsc_DeclareVariable(Handle, ACSC_REAL_TYPE, ArrayName, NULL);
if (!acsc_DataCollection (Handle,// communication handle
        ACSC_DCF_SYNC,// system data collection
        ArrayName,// name of data collection array
        1000,// number of samples to be collected
        1,// sampling period 1 millisecond
        Vars,// variables to be collected
        NULL// waiting call ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.27.2 acsc_StopCollect

Description

The function terminates data collection.

Syntax

int acsc_StopCollect(HANDLE Handle, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The usual system data collection finishes when the required number of samples is collected or the **acsc_StopCollect** function is executed. The application can wait for data collection end with the acsc_WaitCollectEndExt function.

The temporal data collection runs until the **acsc_StopCollect** function is executed.

The function terminates the data collection prematurely. The application can determine the number of actually collected samples from the **S_DCN** variable and the actual sampling period from the **S_DCP** variable.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_StopCollect
// matrix consisting of rows with 1000 columns each
char* ArrayName = "DCA(2)(1000)";
// positions of axes X and Y will be collected
char* Vars[] = { "FPOS(0)", "FPOS(1)" };
acsc_DeclareVariable(Handle, ACSC_REAL_TYPE, ArrayName, NULL);
acsc_Collect(Handle, ACSC_DCF_TEMPORAL, ArrayName, 1000, 1, Vars, NULL);
// waiting for some time
Sleep(2000);
if (!acsc_StopCollect(Handle, NULL))
{
```

```
      printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.27.3 acsc_WaitCollectEndExt

Description

The function waits for the end of data collection.

Syntax

Int acsc_WaitCollectEndExt(HANDLE handle, int Timeout, int Axis)

Arguments

Handle	Communication handle
Timeout	Maximum wait time in milliseconds, If INFINITE - timeout interval never elapses
Axis	If using axis data collection – the axis number, otherwise ACSC_NONE. ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1 etc.

Return Value

If the function succeeds, a non-zero value is returned.

If the function fails, the return value is zero.

Comments

If using Axis data collection (data collection is synced to axis) set **Axis** to be the axis number you are collecting data from, otherwise set **Axis** to be ACSC_NONE.

The function does not return while data collection is in progress and the correct parameters have been passed.

Verifies AST<Axis#>.#DC flag for Synced data collection, otherwise verifies S_ST.#DC system flag.

Using the function with the wrong parameters (i.e. calling the function with an axis number while performing a system data collection) will result in undefined behavior.

Example

```
if(!acsc_DataCollectionExt(Handle, ACSC_DCF_SYNC,ACSC_AXIS_0, ArrayName,
1000, 10, Vars, ACSC_SYNCHRONOUS))
{
   printf("transaction error: %d\n",acsc_GetLastError());
}
if(!acsc_WaitCollectEndExt(Handle, // communication handle
        2000, // timeout
        ACSC_AXIS_0 // Axis number
        ))
{
   printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.28 Status Report Functions

The Status Report functions are:

Table 5-25. Status Report Functions

Function	Description
acsc_GetMotorState	Retrieves the current motor state.
acsc_GetAxisState	Retrieves the current axis state.
acsc_GetIndexState	Retrieves the current state of the index and mark variables.
acsc_ResetIndexState	Resets the specified bit of the index/mark state.
acsc_GetProgramState	Retrieves the current state of the program buffer.

4.28.1 acsc_GetMotorState

Description

The function retrieves the current motor state.

Syntax

int acsc_GetMotorState(HANDLE Handle, int Axis, int* State,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
State	Pointer to a variable that receives the current motor state. The parameter can include one or more of the following flags: ACSC_MST_ENABLE — a motor is enabled ACSC_MST_INPOS — a motor has reached a target position ACSC_MST_MOVE — a motor is moving ACSC_MST_ACC — a motor is accelerating
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result.

	If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the current motor state.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **State** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetMotorState
int State;
if (!acsc_GetMotorState(Handle,          // communication handle
                        ACSC_AXIS_0,     // axis 0
                        &State,          // received value
                        NULL             // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.28.2 acsc_GetAxisState

Description

The function retrieves the current axis state.

Syntax

int acsc_GetAxisState(HANDLE Handle, int Axis, int* State,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.

State	Pointer to a variable that receives the current axis state. The parameter can include one or more of the following flags: ACSC_AST_LEAD – an axis is leading in a group ACSC_AST_DC – an axis data collection is in progress ACSC_AST_PEG – a PEG for the specified axis is in progress ACSC_AST_MOVE – an axis is moving ACSC_AST_ACC – an axis is accelerating ACSC_AST_SEGMENT – a construction of segmented motion for the specified axis is in progress ACSC_AST_VELLOCK – a slave motion for the specified axis is synchronized to master in velocity lock mode ACSC_AST_POSLOCK - a slave motion for the specified axis is synchronized to master in position lock mode ACSC_AST_DECOMPON - dynamic error compensation is active
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the current axis state.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **State** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetAxisState
int State;
if (!acsc_GetAxisState( Handle,                 // communication handle
                        ACSC_AXIS_0,            // axis 0
                        &State,                 // received value
                        NULL                    // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.28.3 acsc_GetIndexState

Description

The function retrieves the current set of bits that indicate the index and mark state.

Syntax

int acsc_GetIndexState(HANDLE Handle, int Axis, int* State,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
State	Pointer to a variable that receives the current set of bits that indicate the index and mark state. The parameter can include one or more of the following flags: ACSC_IST_IND – a primary encoder index of the specified axis is latched ACSC_IST_IND2 – a secondary encoder index of the specified axis is latched ACSC_IST_MARK – a MARK1 signal has been generated and position of the specified axis was latched ACSC_IST_MARK2 – a MARK2 signal has been generated and position of the specified axis was latched
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the current set of bits that indicate the index and mark state.

The controller processes index/mark signals as follows:

When an index/mark signal is encountered for the first time, the controller latches feedback positions and raises the corresponding bit. As long as a bit is raised, the controller does not latch feedback position even if the signal occurs again. To resume latching logic, the application must call the acsc_ResetIndexState function to explicitly reset the corresponding bit.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **State** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetIndexState
int State;
if (!acsc_GetIndexState(        Handle,         // communication handle
                                ACSC_AXIS_0,    // axis 0
                                &State,         // received value
                                NULL            // waiting call
                                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.28.4 acsc_ResetIndexState

Description

The function resets the specified bit of the index/mark state.

Syntax

int acsc_ResetIndexState(HANDLE Handle, int Axis, int Mask,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.

Mask	The parameter contains bit to be cleared. Only one of the following flags can be specified: ACSC_IST_IND – a primary encoder index of the specified axis is latched ACSC_IST_IND2 – a secondary encoder index of the specified axis is latched
Mask	ACSC_IST_MARK – a MARK1 signal has been generated and position of the specified axis was latched ACSC_IST_MARK2 – a MARK2 signal has been generated and position of the specified axis was latched
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function resets the specified bit of the index/mark state. The parameter **Mask** contains a bit, which must be cleared, i.e. the function resets only that bit of the index/mark state, which corresponds to non-zero bit of the parameter **Mask**. To get the current index/mark state, use acsc_GetIndexState function.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_ResetIndexState
if (!acsc_ResetIndexState(Handle,       // communication handle
                        ACSC_AXIS_0,    // axis 0
                        ACSC_IST_IND,   // mentioned bit will be cleared
                        NULL            // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
```

4.28.5 acsc_GetProgramState

Description

The function retrieves the current state of the program buffer.

Syntax

int acsc_GetProgramState(HANDLE Handle, int Buffer, int* State,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Buffer	Number of the buffer.
State	Pointer to a variable that receives the current state of the program buffer. The parameter can include one or more of the following flags: ACSC_PST_COMPILED – a program in the specified buffer is compiled ACSC_PST_RUN – a program in the specified buffer is running ACSC_PST_AUTO – an auto routine in the specified buffer is running ACSC_PST_DEBUG – a program in the specified buffer is executed in debug mode, i.e. breakpoints are active ACSC_PST_SUSPEND – a program in the specified buffer is suspended after the step execution or due to breakpoint in debug mode
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the current state of the program buffer.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **State** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetProgramState
int State;
if (!acsc_GetProgramState(Handle, // communication handle
        0,                        // buffer 0
        &State,                   // received value
        NULL                      // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.29 Input/Output Access Functions

The Input/Output Access functions are:

Table 5-26. Input/Output Access Functions

Function	Description
acsc_GetInput	Retrieves the current state of the specified digital input.
acsc_GetInputPort	Retrieves the current state of the specified digital input port.
acsc_GetInputPort	Retrieves the current state of the specified digital input port.
acsc_GetOutput	Retrieves the current state of the specified digital output.
acsc_GetOutputPort	Retrieves the current state of the specified digital output port.
acsc_SetOutput	Sets the specified digital output to the specified value.
acsc_SetOutputPort	Sets the specified digital output port to the specified value.
acsc_GetAnalogInputNT	Retrieves the current value of the specified analog input signal from an external source such as a sensor or a potentiometer.
acsc_ GetAnalogOutputNT	Retrieves the current value of the specified analog output signal that is sent to an external device such as a sensor or a potentiometer.
acsc_ SetAnalogOutputNT	Writes the specified value to the specified analog output signal that is sent to an external device such as a sensor or a potentiometer.
acsc_GetExtInput	Retrieves the current state of the specified extended input.
acsc_GetExtInputPort	Retrieves the current state of the specified extended input port.

Function	Description
acsc_GetExtOutput	Retrieves the current state of the specified extended output.
acsc_GetExtOutputPort	Retrieves the current state of the specified extended output port.
acsc_SetExtOutput	Sets the specified extended output to the specified value.
acsc_SetExtOutputPort	Sets the specified extended output port to the specified value.

4.29.1 acsc_GetInput

Description

The function retrieves the current state of the specified digital input.

Syntax

int acsc_GetInput(HANDLE Handle, int Port, int Bit, int* Value,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Port	Number of the input port.
Bit	Number of the specific bit.
Value	Pointer to a variable that receives the current state of the specific input. The value will be populated by 0 or 1.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the current state of the specified digital input. To get values of all inputs of the specific port, use the acsc_GetInputPort function.

Digital inputs are represented in the controller variable **IN**. For more information about digital inputs, see the *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Value** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetInput
// the function reads input 0 of port 0 ( IN(0).0 )
int State;
if (!acsc_GetInput(Handle // communication handle
        0,              // port 0
        0,              // bit 0
        &State,         // received value
        NULL            // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.29.2 acsc_GetInputPort

Description

The function retrieves the current state of the specified digital input port.

Syntax

int acsc_GetInputPort(HANDLE Handle, int Port, int* Value,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Port	Number of the input port.
Value	Pointer to a variable that receives the current state of the specific input port.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result.

	If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the current state of the specified digital input port. To get the value of the specific input of the specific port, use the acsc_GetInput function.

Digital inputs are represented in the controller variable **IN**. For more information about digital inputs, see the *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Value** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetInputPort
// the function reads input port 0 ( IN(0) )
int State;
if (!acsc_GetInputPort(Handle, // communication handle
        0,              // port 0
        &State,         // received value
        NULL            // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.29.3 acsc_GetOutput

Description

The function retrieves the current state of the specified digital output.

Syntax

int acsc_GetOutput(HANDLE Handle, int Port, int Bit, int* Value,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Port	Number of the output port.

Bit	Number of the specific bit.
Value	Pointer to a variable that receives the current state of the specific output. The value will be populated by 0 or 1.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the current state of the specified digital output. To get values of all outputs of the specific port, use the acsc_GetOutputPort function.

Digital outputs are represented in the controller variable **OUT**. For more information about digital outputs, see the *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Value** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetOutput
// the function reads output 0 of port 0 ( OUT(0).0 )
int State;
if (!acsc_GetOutput(Handle //communication handle
        0,              // port 0
        0,              // bit 0
        &State,         // received value
        NULL            // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.29.4 acsc_GetOutputPort

Description

The function retrieves the current state of the specified digital output port.

Syntax

int acsc_GetOutputPort(HANDLE Handle, int Port, int* Value,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Port	Number of the output port.
Value	Pointer to a variable that receives the current state of the specific output. The value will be populated by 0 or 1.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the current state of the specified digital output port. To get the value of the specific output of the specific port, use the acsc_GetOutput function.

Digital outputs are represented in the controller variable **OUT**. For more information about digital outputs, see *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Value** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetOutputPort
// the function reads output port 0 ( OUT(0) )
int State;
if (!acsc_GetOutputPort(Handle //communication handle
                        0,              // port 0
                        &State,         // received value
                        NULL            // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.29.5 acsc_SetOutput

Description

The function sets the specified digital output to the specified value.

Syntax

int acsc_SetOutput(HANDLE Handle, int Port, int Bit, int Value,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Port	Number of the output port.
Bit	Number of the specific bit.
Value	The value to be written to the specified output. Any non-zero value is interpreted as 1.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function sets the specified digital output to the specified value. To set values of all outputs of a specific port, use the acsc_SetExtOutputPort function.

Digital outputs are represented in the controller variable **OUT**. For more information about digital outputs, see the *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SetOutput
// the function sets output 0 of port 0 to 1
// ( ACSPL+ equivalent: OUT(0).0 = 1 )
if (!acsc_SetOutput(Handle //communication handle
        0,              // port 0
        0,              // bit 0
        1,              // value to be set
        NULL            // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.29.6 acsc_SetOutputPort

Description

The function sets the specified digital output port to the specified value.

Syntax

int acsc_SetOutputPort(HANDLE Handle, int Port, int Value,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Port	Number of the output port.
Value	The value to be written to the specified output port.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result.

If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function sets the specified digital output port to the specified value. To set the value of the specific output of the specific port, use the acsc_SetOutput function.

Digital outputs are represented in the controller variable **OUT**. For more information about digital outputs, see the *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SetOutputPort
// the function sets first 4 outputs of port 0 to 1
// ( ACSPL+ equivalent: OUT(0) = 0x000F )
if (!acsc_SetOutputPort(Handle  //communication handle
        0,              // port 0
        0x000F,         // value to be set
        NULL            // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.29.7 acsc_GetAnalogInputNT

Description

The function retrieves the current value of the specified analog input signal from an external source such as a sensor or a potentiometer.

Syntax

int _ACSCLIB_ WINAPI acsc_GetAnalogInputNT(HANDLE Handle, int Port, double* Value, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Port	Index of the analog inputs port.
Value	Pointer to a variable that receives current value of the specified analog input as a percentage (range is -100% to 100%) of the maximum level.
Wait	Pointer to **ACSC_WAITBLOCK** structure. If **Wait** is **ACSC_SYNCHRONOUS**, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is **ACSC_IGNORE**, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Comments

The function retrieves the current value of the specified analog input signal from an external source such as a sensor or a potentiometer.

Analog inputs are represented in the controller variable **AIN**. For more information about analog inputs, see *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the calling thread must not use or delete the **Value** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// Example of the waiting call of acsc_GetAnalogInputNT
// The function reads analog input of port 0 ( AIN(0) )
double State;
if (!acsc_GetAnalogInputNT(
        Handle,                 // communication handle
        0,                      // port 0
        &State,                 // received value
        NULL                    // waiting call
        ))
{
        printf("acsc_GetAnalogInputNT(): Error Occurred - %d\n",
                acsc_GetLastError());
}
```

4.29.8 acsc_GetAnalogOutputNT

Description

The function retrieves the current value of the specified analog output signal that is sent to an external device such as a sensor or a potentiometer.

Syntax

int _ACSCLIB_ WINAPI acsc_GetAnalogOutputNT(HANDLE Handle, int Port, double* Value, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Port	Index of the output port.
Value	Pointer to a variable that receives current value of the specified analog output as a percentage (range is -100% to 100%) of the maximum level.
Wait	Pointer to **ACSC_WAITBLOCK** structure. If **Wait** is **ACSC_SYNCHRONOUS**, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is **ACSC_IGNORE**, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Comments

The function retrieves the current value of the specified analog output signal that is sent to an external device such as a sensor or a potentiometer. To write a value to the specific analog output, use the acsc_SetAnalogOutputNT function.

Analog outputs are represented in the controller variable **AOUT**. For more information about analog outputs, see *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the calling thread must not use or delete the **Value** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// Example of the waiting call of acsc_GetAnalogOutputNT
// The function reads analog output of port 0 ( AOUT(0) )
double State;
if (!acsc_GetAnalogOutputNT(
```

```
        Handle                          // communication handle
        0,                              // port 0
        &State,                         // received value
        NULL                            // waiting call
        ))
{
        printf("acsc_GetAnalogOutputNT(): Error Occurred - %d\n",
        acsc_GetLastError());
}
```

4.29.9 acsc_SetAnalogOutputNT

Description

The function writes the specified value to the specified analog output signal that is sent to an external device such as a sensor or a potentiometer.

Syntax

int _ACSCLIB_ WINAPI acsc_SetAnalogOutputNT(HANDLE Handle, int Port, double Value, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Port	Index of the output port.
Value	The value is written to the specified analog output as a percentage (range is -100% to 100%) of the maximum level
Wait	Pointer to **ACSC_WAITBLOCK** structure. If **Wait** is **ACSC_SYNCHRONOUS**, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is **ACSC_IGNORE**, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Comments

The function writes the specified value to the specified analog output signal that is sent to an external device such as a sensor or a potentiometer. To get a value of the specific analog output, use the acsc_GetAnalogOutputNT function.

Analog outputs are represented in the controller variable **AOUT**. For more information about analog outputs, see the *SPiiPlus ACSPL+ Programmer's Guide.*

If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the calling thread must not use or delete the Value and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// Example of the waiting call of acsc_SetAnalogOutputNT
// The function writes the value of 22.54 to the analog output of port 0
// ( ACSPL+ equivalent: AOUT(0) = 22.54 )
if (!acsc_SetAnalogOutputNT(
        Handle,                     // communication handle
        0,                          // port 0
        22.54,                      // received value
        NULL                        // waiting call
        ))
{
     printf("acsc_SetAnalogOutputNT(): Error Occurred - %d\n",
         acsc_GetLastError());
}
```

4.29.10 acsc_GetExtInput

Description

The function retrieves the current state of the specified extended input.

Syntax

int acsc_GetExtInput(HANDLE Handle, int Port, int Bit, int* Value,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Port	Number of the extended input port.
Bit	Number of the specific bit.
Value	Pointer to a variable that receives the current state of the specific input. The value will be populated by 0 or 1.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the current state of the specified extended input. To get values of all inputs of the specific extended port, use the acsc_GetExtInputPort function.

Extended inputs are represented in the controller variable **EXTIN**. For more information about extended inputs, see *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Value** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetExtInput
// the function reads extended input 0 of port 0 ( EXTIN(0).0 )
int State;
if (!acsc_GetExtInput(Handle    //communication handle
        0,                      // port 0
        0,                      // bit 0
        &State,                 // received value
        NULL                    // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.29.11 acsc_GetExtInputPort

Description

The function retrieves the current state of the specified extended input port.

Syntax

int acsc_GetExtInputPort(HANDLE Handle, int Port, int* Value,
 ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Port	Number of the extended input port.
Value	Pointer to a variable that receives the current state of the specific input port.
Wait	Pointer to ACSC_WAITBLOCK structure.

	If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCallfunction to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the current state of the specified extended input port. To get the value of the specific input of the specific extended port, use the acsc_GetExtInput function**.**

Extended inputs are represented in the controller variable **EXTIN**. For more information about extended inputs, see *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Value** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetExtInputPort
// the function reads extended input port 0 (EXTIN(0))
int State;
if (!acsc_GetExtInputPort(Handle        //communication handle
        0,              // port 0
        &State,         // received value
        NULL            // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.29.12 acsc_GetExtOutput

Description

The function retrieves the current state of the specified extended output.

Syntax

int acsc_GetExtOutput(HANDLE Handle, int Port, int Bit, int* Value,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Port	Number of the extended output port.
Bit	Number of the specific bit.
Value	Pointer to a variable that receives the current state of the specific output. The value will be populated by 0 or 1.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the current state of the specified extended output. To get values of all outputs of the specific extended port, use the acsc_GetExtOutputPort function.

Extended outputs are represented in the controller variable **EXTOUT**. For more information about extended outputs, see *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Value** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetExtOutput
// the function reads extended output 0 of port 0 ( EXTOUT(0).0 )
int State;
if (!acsc_GetExtOutput(Handle   //communication handle
        0,              // port 0
        0,              // bit 0
        &State,         // received value
        NULL            // waiting call
        ))
```

```
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.29.13 acsc_GetExtOutputPort

Description

The function retrieves the current state of the specified extended output port.

Syntax

int acsc_GetExtOutputPort(HANDLE Handle, int Port, int* Value,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Port	Number of the extended output port.
Value	Pointer to a variable that receives the current state of the specific output. The value will be populated by 0 or 1.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the current state of the specified extended output port. To get the value of the specific output of the specific extended port, use the **acsc_GetExtOutputPort** function.

Extended outputs are represented in the controller variable **EXTOUT**. For more information about extended outputs, see *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Value** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetExtOutputPort
// the function reads extended output port 0 ( EXTOUT(0) )
int State;
if (!acsc_GetExtOutputPort(Handle //communication handle
        0,              // port 0
        &State,         // received value
        NULL            // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.29.14 acsc_SetExtOutput

Description

The function sets the specified extended output to the specified value.

Syntax

int acsc_SetExtOutput(HANDLE Handle, int Port, int Bit, int Value,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Port	Number of the extended output port.
Bit	Number of the specific bit.
Value	The value to be written to the specified output. Any non-zero value is interpreted as 1.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function sets the specified extended output to the specified value. To set values of all outputs of the specific extended port, use the acsc_SetExtOutputPort function.

Extended outputs are represented in the controller **EXTOUT** variable. For more information about extended outputs, see the *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SetExtOutput
// the function sets output 0 of extended port 0 to 1
// ( ACSPL+ equivalent: EXTOUT(0).0 = 1 )
if (!acsc_SetExtOutput(Handle   //communication handle
        0,              // port 0
        0,              // bit 0
        1,              // value to be set
        NULL            // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.29.15 acsc_SetExtOutputPort

Description

The function sets the specified extended output port to the specified value.

Syntax

int acsc_SetExtOutputPort(HANDLE Handle, int Port, int Value,
 ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Port	Number of the extended output port.
Value	The value written to the specified output port.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received.

	If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function sets the specified extended output port to the specified value. To set the value of the specific output of the specific extended port, use the acsc_SetExtOutput function.

Extended outputs are represented in the controller variable **EXTOUT**. For more information about extended outputs, see the *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SetExtOutputPort
// the function sets first 4 outputs of extended port 0 to 1
// ( ACSPL+ equivalent: EXTOUT(0) = 0x000F )
if (!acsc_SetExtOutputPort(Handle//communication handle
        0,              // port 0
        0x000F,         // value to be set
        NULL            // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.30 Safety Control Functions

The Safety Control functions are:

Table 5-27. Safety Control Functions

Function	Description
acsc_GetFault	Retrieves the set of bits that indicate the motor or system faults.
acsc_SetFaultMask	Sets the mask, that enables/disables the examination and processing of the controller faults.
acsc_GetFaultMask	Retrieves the mask that defines which controller faults are examined and processed.
acsc_EnableFault	Enables the specified motor or system fault.
acsc_DisableFault	Disables the specified motor or system fault.
acsc_SetResponseMask	Sets the mask that defines for which motor or system faults the controller provides default response.
acsc_GetResponseMask	Retrieves the mask that defines for which motor or system faults the controller provides default response.
acsc_EnableResponse	Enables the default response to the specified motor or system fault.
acsc_DisableResponse	Disables the default response to the specified motor or system fault.
acsc_GetSafetyInput	Retrieves the current state of the specified safety input.
acsc_GetSafetyInputPort	Retrieves the current state of the specified safety input port.
acsc_ GetSafetyInputPortInv	Retrieves the set of bits that define inversion for the specified safety input port.
acsc_ SetSafetyInputPortInv	Sets the set of bits that define inversion for the specified safety input port.
acsc_FaultClear	The function clears the current faults and results of previous faults stored in the MERR variable.
acsc_FaultClearM	The function clears the current faults and results of previous faults stored in the MERR variable for multiple axis.

4.30.1 acsc_GetFault

Description

The function retrieves the set of bits that indicate the motor or system faults.

Syntax

int acsc_GetFault(HANDLE Handle, int Axis, int* Fault, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle
Axis	The axis constant specifying the axis (ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc.) to receive the motor faults or ACSC_NONE to receive the system faults. For the axis constants see Axis Definitions.
Fault	Pointer to the variable that receives the current set of fault bits.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the set of bits that indicate motor or system faults.

Motor faults are related to a specific motor, the power amplifier, and the Servo processor. For example: Position Error, Encoder Error, or Driver Alarm.

System faults are not related to any specific motor. For example: Emergency Stop or Memory Fault.

The parameter **Fault** receives the set of bits that indicates the controller faults. To recognize the specific fault, constants ACSC_SAFETY_*** can be used. See Safety Control Masks for a detailed description of these constants.

For more information about the controller faults, see the *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Fault** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetFault
int Fault;
if (!acsc_GetFault(Handle       //communication handle
```

```
        ACSC_AXIS_0,            // axis 0
        &Fault,                 // received set of faults
        NULL                    // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
if (Fault & ACSC_SAFETY_RL)     printf("Right Limit fault\n");
if (Fault & ACSC_SAFETY_LL)     printf("Left Limit fault\n");
```

4.30.2 acsc_SetFaultMask

Description

The function sets the mask that enables or disables the examination and processing of controller faults.

Syntax

int acsc_SetFaultMask(HANDLE Handle, int Axis, int Mask,
ACSC_WAITBLOCK* Wait)

Certain controller faults provide protection against potential serious bodily injury and damage to the equipment. Be aware of the implications before disabling any alarm, limit, or error.

Arguments

Handle	Communication handle
Axis	The axis constant specifying the axis (ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc.) to mask the motor faults, or ACSC_NONE to mask the system faults. For the axis constants see Axis Definitions.
Mask	The mask to be set: If a bit of the **Mask** is zero, the corresponding fault is disabled. To set/reset a specified bit, use ACSC_SAFETY_*** constants. See Safety Control Masks for a detailed description of these constants. If the **Mask** is ACSC_NONE, then all the faults for the specified axis are enabled. If the **Mask** is zero, then all the faults for the specified axis are disabled.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result.

If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function sets the mask that enables/disables the examination and processing of the controller faults. The two types of controller faults are motor faults and system faults.

The motor faults are related to a specific motor, the power amplifier or the Servo processor. For example: Position Error, Encoder Error or Driver Alarm.

The system faults are not related to any specific motor. For example: Emergency Stop or Memory Fault.

For more information about the controller faults, see the *SPiiPlus ACSPL+ Programmer's Guide.*

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SetFaultMask
if (!acsc_SetFaultMask(Handle,  //communication handle
        ACSC_AXIS_0,            // axis 0
        ACSC_NONE,              // enable all faults
        NULL            // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.30.3 acsc_GetFaultMask

Description

The function retrieves the mask that defines which controller faults are examined and processed.

Syntax

int acsc_GetFaultMask(HANDLE Handle, int Axis, int* Mask,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle
Axis	The axis constant specifying the axis (ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc.) to get the motor faults mask, or ACSC_NONE to get the system faults mask. For the axis constants see Axis Definitions.
Mask	The current faults mask. If a bit of **Mask** is zero, the corresponding fault is disabled. Use the ACSC_SAFETY_*** constants to examine the specified bit. See Safety Control Masks for a detailed description of these constants.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the mask, defining which controller faults are examined and processed. If a bit of the parameter **Mask** is zero, the corresponding fault is disabled.

The controller faults are of two types: motor faults and system faults.

The motor faults are related to a specific motor, the power amplifier or the Servo processor. For example: Position Error, Encoder Error or Driver Alarm.

The system faults are not related to any specific motor, for example: Emergency Stop or Memory Fault.

For more information about the controller faults, see the *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Mask** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetFaultMask
int Mask;
if (!acsc_GetFaultMask(  Handle,        // communication handle
                         ACSC_AXIS_0,   // axis 0
                         &Mask,         // current mask
                         NULL           // waiting call
                         ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.30.4 acsc_EnableFault

Description

The function enables the specified motor or system fault.

Syntax

int acsc_EnableFault(HANDLE Handle, int Axis, int Fault,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle
Axis	The axis constant specifying the axis (ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc.) to enable the motor fault or ACSC_NONE to enable the system fault. For the axis constants see Axis Definitions.
Fault	The fault to be enabled. Only one fault can be enabled at a time. To specify the fault, one of the constants ACSC_SAFETY_*** can be used. See Safety Control Masks for a detailed description of these constants.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the acsc_WaitForAsyncCall function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function enables the examination and processing of the specified motor or system fault by setting the specified bit of the fault mask to one.

The motor faults are related to a specific motor, the power amplifier, and the Servo processor. For example: Position Error, Encoder Error, and Driver Alarm.

The system faults are not related to any specific motor. For example: Emergency Stop, Memory Fault.

For more information about the controller faults, see *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_EnableFault
if (!acsc_EnableFault(Handle,    // communication handle
        ACSC_AXIS_0,             // axis 0
        ACSC_SAFETY_PE,          // enable fault Position Error
        NULL                     // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.30.5 acsc_DisableFault

Description

The function disables the specified motor or system fault.

Syntax

int acsc_DisableFault(HANDLE Handle, int Axis, int Fault,
ACSC_WAITBLOCK* Wait)

Certain controller faults provide protection against potential serious bodily injury and damage to the equipment. Be aware of the implications before disabling any alarm, limit, or error.

Arguments

Handle	Communication handle
Axis	The axis constant specifying the axis (ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc.) to disable the motor faults or ACSC_NONE to disable the system faults.

	For the axis constants see Axis Definitions.
Fault	The fault to be disabled. Only one fault can be disabled at a time. To specify the fault, one of the constants ACSC_SAFETY_*** can be used. See Safety Control Masks for a detailed description of these constants.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function disables the examination and processing of the specified motor or system fault by setting the specified bit of the fault mask to zero.

The motor faults are related to a specific motor, the power amplifier, and the Servo processor. For example: Position Error, Encoder Error, and Driver Alarm.

The system faults are not related to any specific motor, for example: Emergency Stop, Memory Fault.

For more information about the controller faults, see *SPiiPlus ACSPL+ Programmer's Guide.*

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_DisableFault
if (!acsc_DisableFault( Handle,                 // communication handle
                        ACSC_AXIS_0,            // axis 0
                        ACSC_SAFETY_VL,         // disable fault Velocity
                                                // Limit
                        NULL                    // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.30.6 acsc_SetResponseMask

Description

The function retrieves the mask that defines the motor or the system faults for which the controller provides the default response.

Syntax

int acsc_SetResponseMask(HANDLE Handle, int Axis, int Mask,
ACSC_WAITBLOCK* Wait)

Certain controller faults provide protection against potential serious bodily injury and damage to the equipment. Be aware of the implications before disabling any alarm, limit, or error.

Arguments

Handle	Communication handle
Axis	The axis constant specifying the axis (ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc.) to set the mask of responses to the motor faults, or ACSC_NONE to set the mask of responses to the system faults. For the axis constants see Axis Definitions.
Mask	The mask to be set. If a bit of **Mask** is zero, the corresponding default response is disabled. Use the ACSC_SAFETY_*** constants to set/reset a specified bit. See Safety Control Masksfor a detailed description of these constants. If **Mask** is ACSC_NONE, all default responses for the specified axis are enabled. If **Mask** is zero, all default responses for the specified axis are disabled.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the mask that defines the motor or the system faults for which the controller provides the default response.

The default response is a controller-predefined action for the corresponding fault. For more information about the controller faults and default responses, see the *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_SetResponseMask
if (!acsc_SetResponseMask(Handle,       // communication handle
                          ACSC_AXIS_0,  // axis 0
                          ACSC_NONE,    // enable all default responses
                          NULL          // waiting call
                          ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.30.7 acsc_GetResponseMask

Description

The function retrieves the mask that defines the motor or the system faults for which the controller provides the default response.

Syntax

int acsc_GetResponseMask(HANDLE Handle, int Axis, int* Mask,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle
Axis	The axis constant specifying the axis (ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc.) to get the mask of responses to the motor faults, or ACSC_NONE to get the mask of responses to the system faults. For the axis constants see Axis Definitions.
Mask	The current mask of default responses. If a bit of **Mask** is zero, the corresponding default response is disabled. Use the ACSC_SAFETY_*** constants to examine the specified bit. See Safety Control Masks for a detailed description of these constants.

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the mask that defines the motor or the system faults for which the controller provides the default response. If a bit of the parameter **Mask** is zero, the controller does not provide a default response to the corresponding fault.

The default response is a controller-predefined action for the corresponding fault. For more information about the controller faults and default, responses see the *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Mask** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetResponseMask
int Mask;
if (!acsc_GetResponseMask(Handle,       // communication handle
                ACSC_AXIS_0,            // axis 0
                &Mask,                  // current responses mask
                NULL                    // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.30.8 acsc_EnableResponse

Description

The function enables the response to the specified motor or system fault.

Syntax

int acsc_EnableResponse(HANDLE Handle, int Axis, int Response,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle
Axis	The axis constant specifying the axis (ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc.) to enable the response to the specified motor fault, or ACSC_NONE to enable the response to the specified system fault. For the axis constants see Axis Definitions.
Response	The default response to be enabled. Only one default response can be enabled at a time. To specify the default response, one of the constants ACSC_SAFETY_*** can be used. See Safety Control Masks for a detailed description of these constants.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function enables the default response to the specified motor or system fault by setting the specified bit of the response mask to one.

The default response is a controller-predefined action for the corresponding fault. For more information about the controller faults and default responses, see *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_EnableResponse
if (!acsc_EnableResponse(Handle,// communication handle
                ACSC_AXIS_0,    // axis 0
                ACSC_SAFETY_PE, // enable the default
                                // response to the Position
                                // Error fault
                NULL            // waiting call
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.30.9 acsc_DisableResponse

Description

The function disables the default response to the specified motor or system fault.

Syntax

int acsc_DisableResponse(HANDLE Handle, int Axis, int Response,
ACSC_WAITBLOCK* Wait)

Certain controller faults provide protection against potential serious bodily injury and damage to the equipment. Be aware of the implications before disabling any alarm, limit, or error.

Arguments

Handle	Communication handle
Axis	The axis constant specifying the axis (ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc.) to disable the default response to the specified motor fault, or ACSC_NONE to disable response to the specified system fault. For the axis constants see Axis Definitions.
Response	The response to be disabled. Only one default response can be disabled at a time. To specify the fault, one of the constants ACSC_SAFETY_*** can be used. See Safety Control Masks for a detailed description of these constants.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result.

	If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function disables the default response to the specified motor or system fault by setting the specified bit of the response mask to zero.

The default response is a controller-predefined action for the corresponding fault. For more information about the controller faults and default responses, see *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_DisableResponse
if (!acsc_DisableResponse(Handle,                // communication handle
                          ACSC_AXIS_0,           // axis 0
                          ACSC_SAFETY_VL,        // disable the default
                                                 // response to the
                                                 // Velocity Limit fault
                          NULL                   // waiting call
                          ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.30.10 acsc_GetSafetyInput

Description

The function retrieves the current state of the specified safety input.

Syntax

int acsc_GetSafetyInput(HANDLE Handle, int Axis, int Input, int* Value,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	The axis constant specifying the axis (ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc.) to get the specified safety motor input, or ACSC_NONE to get the specified system safety input. For the axis constants see Axis Definitions.
Input	The specific safety input. To specify a desired motor safety input OR combination of any of the following constants can be used: ACSC_SAFETY_RL ACSC_SAFETY_LL ACSC_SAFETY_RL2 ACSC_SAFETY_LL2 ACSC_SAFETY_HOT ACSC_SAFETY_DRIVE ACSC_SAFETY_ES See Safety Control Masks for a detailed description of these constants.
Value	Pointer to a variable that receives the current state of the specified inputs. The value will be populated by 0 or 1. The value will receive 1 if any of the specified safety inputs in the Input field is on. Otherwise, it receives 0.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the current state of the specified safety input. To get values of all safety inputs of the specific axis, use the **acsc_GetSafetyInput** function.

Safety inputs are represented in the controller variables **SAFIN** and **S_SAFIN**. For more information about safety inputs, see *SPiiPlus ACSPL+ Programmer's Guide.*

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Value** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Some safety inputs can be unavailable in a specific controller model. For example, SPiiPlus SA controller does not provide Motor Overheat, Preliminary Left Limit, and Preliminary Right Limit safety inputs.

See specific model documentation for details.

Example

```
// example of the waiting call of acsc_GetSafetyInput
// the function reads Emergency Stop system safety input
int State;
if (!acsc_GetSafetyInput(Handle,// communication handle
        ACSC_NONE,              // system safety input port
        ACSC_SAFETY_ES,         // Emergency Stop safety input
        &State,                 // received value
        NULL                    // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.30.11 acsc_GetSafetyInputPort

Description

The function retrieves the current state of the specified safety input port.

Syntax

int acsc_GetSafetyInputPort(HANDLE Handle, int Axis, int* Value,
 ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	The axis constant specifying the axis (ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc.) to get the specified safety motor inputs port, or ACSC_NONE to get the specified safety system inputs port. For the axis constants see Axis Definitions.

Value	Pointer to a variable that receives the current state of the specific safety input port. To recognize a specific motor safety input, only one of the following constants can be used: ACSC_SAFETY_RL ACSC_SAFETY_LL ACSC_SAFETY_RL2 ACSC_SAFETY_LL2 ACSC_SAFETY_HOT ACSC_SAFETY_DRIVE To recognize a specific system safety input, only ACSC_SAFETY_ES constant can be used. See Safety Control Masks for a detailed description of these constants.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the current state of the specified safety input port. To get the state of the specific safety input of a specific axis, use the acsc_GetSafetyInput function.

Safety inputs are represented in the controller variables **SAFIN** and **S_SAFIN**. For more information about safety inputs, see *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Value** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Some safety inputs can be unavailable in a specific controller model. For example, SPiiPlus SA controller does not provide Motor Overheat, Preliminary Left Limit, and Preliminary Right Limit safety inputs.

See specific model documentation for details.

Example

```
// example of the waiting call of acsc_GetSafetyInputPort
// the function reads safety input port of the axis 0
int State;
if (!acsc_GetSafetyInputPort(   Handle,         // communication handle
                                ACSC_AXIS_0,    // axis 0
                                &State,         // received value
                                NULL            // waiting call
                                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.30.12 acsc_GetSafetyInputPortInv

Description

The function retrieves the set of bits that define inversion for the specified safety input port.

Syntax

int acsc_GetSafetyInputPortInv(HANDLE Handle, int Axis, int* Value,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	The axis constant specifying the axis (ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc.) to get the inversion for the specified safety motor inputs port, or ACSC_NONE to get the inversion for the specified safety system inputs port. For the axis constants see Axis Definitions.

Value	Pointer to a variable that receives the set of bits that define inversion for the specific safety input port. To recognize a specific bit, use the following constants: ACSC_SAFETY_RL ACSC_SAFETY_LL ACSC_SAFETY_RL2 ACSC_SAFETY_LL2 ACSC_SAFETY_HOT ACSC_SAFETY_DRIVE Use the ACSC_SAFETY_ES constant to recognize an inversion for the specific system safety input port,. See Safety Control Masks for a detailed description of these constants.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the set of bits that define inversion for the specified safety input port. To set the specific inversion for the specific safety input port, use the **acsc_GetSafetyInputPortInv** function.

If a bit of the retrieved set is zero, the corresponding signal is not inverted and therefore high voltage is considered an active state. If a bit is raised, the signal is inverted and low voltage is considered an active state.

The inversions of safety inputs are represented in the controller variables **SAFINI** and **S_SAFINI**. For more information about safety inputs, see the *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Value** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Some safety inputs can be unavailable in a specific controller model. For example, SPiiPlus SA controller does not provide Motor Overheat, Preliminary Left Limit, and Preliminary Right Limit safety inputs.

See specific model documentation for details.

Example

```
// example of the waiting call of acsc_GetSafetyInputPortInv
// the function reads an inversion of safety input port of the axis 0
int State;
if (!acsc_GetSafetyInputPortInv(Handle, // communication handle
                        ACSC_AXIS_0,    // axis 0
                        &State,         // received value
                        NULL            // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.30.13 acsc_SetSafetyInputPortInv

Description

The function sets the set of bits that define inversion for the specified safety input port.

Syntax

int acsc_SetSafetyInputPortInv(HANDLE Handle, int Axis, int Value,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	The axis constant (ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc.) specifying the axis to set the specified safety motor inputs port, or ACSC_NONE to set the specified safety system inputs port. For the axis constants see Axis Definitions.

Value	The specific inversion. To set a specific bit, use the following constants: ACSC_SAFETY_RL, ACSC_SAFETY_LL, ACSC_SAFETY_RL2, ACSC_SAFETY_LL2, ACSC_SAFETY_HOT, ACSC_SAFETY_DRIVE. To set an inversion for the specific system safety input port, use only the ACSC_SAFETY_ES constant. See Safety Control Masks for a detailed description of these constants.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function sets the bits that define inversion for the specified safety input port. To retrieve an inversion for the specific safety input port, use the acsc_GetSafetyInputPortInv function.

If a bit of the set is zero, the corresponding signal will not be inverted and therefore high voltage is considered an active state. If a bit is raised, the signal will be inverted and low voltage is considered an active state.

The inversions of safety inputs are represented in the controller variables **SAFINI** and **S_SAFINI**. For more information about safety, inputs see *SPiiPlus ACSPL+ Programmer's Guide*.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Some safety inputs can be unavailable in a specific controller model. For example, SPiiPlus SA controller does not provide Motor Overheat, Preliminary Left Limit, and Preliminary Right Limit safety inputs.

See specific model documentation for details.

Example

```
// example of the waiting call of acsc_SetSafetyInputPortInv
// the function sets the inversion for safety input port of the axis 0
int State;
if (!acsc_SetSafetyInputPortInv(Handle, // communication handle
                        ACSC_AXIS_0,    // axis 0
                        State,          // inversion that will be set
                        NULL            // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.30.14 acsc_FaultClear

Description

The function clears the current faults and the result of the previous fault stored in the

MERR variable.

Syntax

int acsc_FaultClear(HANDLE Handle, int Axis, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function clears the current faults of the specified axis and the result of the previous fault stored in the **MERR** variable.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_Disable
if (!acsc_FaultClear (Handle,   // communication handle
         ACSC_AXIS_0,           // disable axis 0
         NULL                   // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError())
}
```

4.30.15 acsc_FaultClearM

Description

The function clears the current faults and results of previous faults stored in the MERR variable for multiple axes.

Syntax

int acsc_FaultClearM(HANDLE Handle, int *Axes, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axes	Array of axis constants. Each element specifies one involved axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. After the last axis, one additional element must be located that contains –1 which marks the end of the array. For the axis constants see Axis Definitions.

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

If the reason for the fault is still active, the controller will set the fault immediately after this command is performed. If cleared fault is Encoder Error, the feedback position is reset to zero.

Example

```
// example of the waiting call of acsc_FclearM
int Axes[]={ACSC_AXIS_0,ACSC_AXIS_1,-1};
if (!acsc_FaultClearM(  Handle,         //communication handle
                        Axes,           //Axes 0 and 1
                        NULL            // waiting call
                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.31 Wait-for-Condition Functions

The Wait-for-Condition functions are:

Table 5-28. Wait-for-Condition Functions

Function	Description
acsc_WaitMotionEnd	Waits for the end of a motion.
acsc_WaitLogicalMotionEnd	Waits for the logical end of a motion.

Function	Description
acsc_WaitForAsyncCall	Waits for the end of data collection.
acsc_WaitProgramEnd	Waits for the program termination in the specified buffer.
acsc_WaitMotorEnabled	Waits for the specified state of the specified motor.
acsc_WaitInput	Waits for the specified state of the specified digital input.
acsc_WaitUserCondition	Waits for user-defined condition.
acsc_WaitMotorCommutated	Waits for the specified motor to be commutated.

4.31.1 acsc_WaitMotionEnd

Description

The function waits for the end of a motion.

Syntax

int acsc_WaitMotionEnd (HANDLE Handle, int Axis, int Timeout)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
Timeout	Maximum waiting time in milliseconds. If **Timeout** is INFINITE, the function's time-out interval never elapses.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function does not return while the specified axis is involved in a motion, the motor has not settled in the final position and the specified time-out interval has not elapsed.

The function differs from the acsc_WaitLogicalMotionEnd function. Examining the same motion, the **acsc_WaitMotionEnd** function will return latter. The **acsc_WaitLogicalMotionEnd** function returns when the generation of the motion finishes. On the other hand, the **acsc_WaitMotionEnd** function returns when the generation of the motion finishes and the motor has settled in the final position.

Example

```
if (!acsc_WaitMotionEnd(Handle,          // communication handle
                ACSC_AXIS_0,             // wait for the end of motion of axis 0
                30000                    // during 30 sec
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.31.2 acsc_WaitLogicalMotionEnd

Description

The function waits for the logical end of a motion.

Syntax

int acsc_WaitLogicalMotionEnd (HANDLE Handle, int Axis, int Timeout)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions. The axis must be either a single axis not included in any group or a leading axis of a group.
Timeout	Maximum waiting time in milliseconds. If **Timeout** is INFINITE, the function's time-out interval never elapses.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function does not return while the specified axis is involved in a motion and the specified time-out interval has not elapsed.

The function differs from the acsc_WaitMotionEnd function. Examining the same motion, the **acsc_WaitMotionEnd** function will return later. The **acsc_WaitLogicalMotionEnd** function returns when the generation of the motion finishes. On the other hand, the **acsc_WaitMotionEnd** function returns when the generation of the motion finishes and the motor has settled in the final position.

Example

```
if (!acsc_WaitLogicalMotionEnd( Handle, // communication handle
                ACSC_AXIS_0,    // wait for the logical end of motion of axis 0
                30000           // during 30 sec
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.31.3 acsc_WaitForAsyncCall

Description

The function waits for completion of asynchronous call and retrieves a data.

Syntax

int acsc_WaitForAsyncCall(HANDLE Handle, void* Buf, int* Received,
ACSC_WAITBLOCK* Wait, int Timeout)

Arguments

Handle	Communication handle.
Buf	Pointer to the buffer that receives controller response. This parameter must be the same pointer that was specified for asynchronous call of SPiiPlus C function. If the SPiiPlus C function does not accept a buffer as a parameter, Buf has to be NULL pointer.
Received	Number of characters that were actually received.
Wait	Pointer to the same ACSC_WAITBLOCK structure that was specified for asynchronous call of SPiiPlus C function.
Timeout	Maximum waiting time in milliseconds. If **Timeout** is INFINITE, the function's time-out interval never elapses.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero. The **Ret** field of **Wait** contains the error code that the non-waiting call caused. If **Wait.Ret** is zero, the call succeeded: no errors occurred.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function waits for completion of asynchronous call, corresponds to the **Wait** parameter, and retrieves controller response to the buffer pointed by **Buf**. The **Wait** and **Buf** must be the same pointers passed to SPiiPlus C function when asynchronous call was initiated.

If the call of SPiiPlus C function was successful, the function retrieves controller response to the buffer **Buf**. The **Received** parameter will contain the number of actually received characters.

If the call of SPiiPlus C function does not return a response (for example: acsc_Enable, acsc_Jog, etc.) **Buf** has to be NULL.

If the call of SPiiPlus C function returned the error, the function retrieves this error code in the **Ret** member of the **Wait** parameter.

If the SPiiPlus C function has not been completed in Timeout milliseconds, the function aborts specified asynchronous call and returns ACSC_TIMEOUT error.

If the call of SPiiPlus C function has been aborted by the acsc_CancelOperation function, the function returns ACSC_OPERATIONABORTED error.

Example

```
char* cmd = "?VR\r";    // get firmware version
char buf[101];
int Received;
ACSC_WAITBLOCK wait;
if (!acsc_Transaction(Handle, cmd, strlen(cmd), buf, 100, &Received,
                      &wait))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
else
{
        // call is pending
        if (acsc_WaitForAsyncCall(Handle, // communication handle
                buf,    // pointer to the same buffer, that was specified
                        // for acsc_Transaction
                &Received, // received bytes
                &wait,  // pointer to the same structure, that was
                        // specified for acsc_Transaction
                500     // 500 ms
                        ))
        {
                buf[Received] = '\0';
                printf("Firmware version: %s\n", buf);
        }
        else
        {
                acsc_GetErrorString(Handle, wait.Ret, buf, 100, &Received);
                buf[Received] = '\0';
                printf("error: %s\n", buf);
        }
}
```

4.31.4 acsc_WaitProgramEnd

Description

The function waits for the program termination in the specified buffer.

Syntax

int acsc_WaitProgramEnd (HANDLE Handle, int Buffer, int Timeout)

Arguments

Handle	Communication handle.
Buffer	Buffer number, from 0 to 9.
Timeout	Maximum waiting time in milliseconds. If **Timeout** is INFINITE, the function's time-out interval never elapses.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function does not return while the ACSPL+ program in the specified buffer is in progress and the specified time-out interval has not elapsed.

Example

```
if (!acsc_WaitProgramEnd(Handle, // communication handle
                0,               // buffer 0
                30000            // during 30 sec
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.31.5 acsc_WaitMotorEnabled

Description

The function waits for the specified state of the specified motor.

Syntax

int acsc_WaitMotorEnabled (HANDLE Handle, int Axis, int State, int Timeout)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to 1, etc. For the axis constants see Axis Definitions.

State	1 – the function wait for the motor enabled, 0 – the function wait for the motor disabled.
Timeout	Maximum waiting time in milliseconds. If **Timeout** is INFINITE, the function's time-out interval never elapses.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function does not return while the specified motor is not in the desired state and the specified time-out interval has not elapsed. The function examines the **MST.#ENABLED** motor flag.

Example

```
If( !acsc_WaitMotorEnabled(Handle,      //communication handle
                        ACSC_AXIS_0,    //axis 0
                        1,              //wait untill the motor is enabled
                        30000           // during 30 sec
                        ))
{
        printf("WaitMotorEnabled error: %d\n", acsc_GetLastError() );
}
```

4.31.6 acsc_WaitInput

Description

The function waits for the specified state of digital input.

Syntax

int acsc_WaitInput (HANDLE Handle, int Port, int Bit, int State, int Timeout)

Arguments

Handle	Communication handle.
Port	Number of input port: 0 corresponds to **IN0**, 1 – to **IN1**, etc.
Bit	Selects one bit from the port, from 0 to 31.
State	Specifies a desired state of the input, 0 or 1.
Timeout	Maximum waiting time in milliseconds. If **Timeout** is INFINITE, the function's time-out interval never elapses.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The basic configuration of the SPiiPlus PCI model provides only 16 inputs. Therefore, the **Port** must be 0, and the **Bit** can be specified from 0 to 15.

Example

```
if (!acsc_WaitInput(Handle,     // communication handle
                0,              // IN0
                0,              // IN0.0
                1,              // wait for IN0.0 = 1
                30000           // during 30 sec
                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.31.7 acsc_WaitUserCondition

Description

The function waits for the user-defined condition.

Syntax

int acsc_WaitUserCondition(HANDLE Handle,
ACSC_USER_CONDITION_FUNC UserCondition, int Timeout)

Arguments

Handle	Communication handle.
UserCondition	A pointer to a user-defined function that accepts an argument of HANDLE type and returns the result of integer type. Its prototype is: `int WINAPI UserCondition (HANDLE Handle);`
Timeout	Maximum waiting time in milliseconds. If **Timeout** is INFINITE, the function's time-out interval never elapses.

Return Value

If the function succeeds, the return value is non-zero. If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function calls the **UserCondition** function in order to determine the termination moment.

While the **UserCondition** function returns zero and the specified time-out interval has not elapsed, the **acsc_WaitUserCondition** function calls **UserCondition** periodically. Once the **UserCondition** function returns non-zero results, the **acsc_WaitUserCondition** function also returns.

The **UserCondition** function accepts as argument the **Handle** parameter, passed to the function **acsc_WaitUserCondition**.

If the condition is satisfied, the **UserCondition** function returns 1, if it is not satisfied – 0. In case of an error the **UserCondition** function returns –1.

Example

```
// In the example the UserCondition function checks when the feedback
// position of the motor 0 will reach 100000.
int WINAPI UserCondition(HANDLE Handle)
{
    double FPos;
    if (acsc_ReadReal(Handle, ACSC_NONE, "FPOS", 0, 0, ACSC_NONE,
        ACSC_NONE, &FPos, NULL))
    {
        if (FPos > 100000)
            return 1;
    }
    else return -1;             // error is occurred
    return 0;                   // continue waiting
}
…..
// wait while FPOS arrives at point 100000
if (!acsc_WaitUserCondition(Handle,     // communication handle
                UserCondition,  // pointer to the user-defined function
                30000           // during 30 sec
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.31.8 acsc_WaitMotorCommutated

Description

The function waits for the specified motor to be commutated.

Syntax

int acsc_WaitMotorCommutated (HANDLE Handle, int Axis, int State, int Timeout)

Arguments

Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
State	1 - The function waits for the motor to be commutated.
Timeout	Maximum waiting time in milliseconds. If **Timeout** is INFINITE, the function's time-out interval never elapses.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function does not return while the specified motor is not in the desired state and the specified time-out interval has not elapsed. The function examines the **MFLAGS.#BRUSHOK** flag.

Example

```
if (!acsc_WaitMotorCommutated(Handle, // communication handle
        ACSC_AXIS_0,    // axis 0
        TRUE,   // wait until motor is commutated
        30000   // during 30 sec
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.32 Callback Registration Functions

The Callback Registration functions are:

Table 5-29. Callback Registration Functions

Function	Description
acsc_InstallCallback	Installs a user-defined callback function for the specified interrupt condition with user-defined parameter.
acsc_InstallCallbackExt	Installs a user-defined callback function for the specified interrupt condition with user-defined parameter, extending the range of axes up to 128.
acsc_SetCallbackMask	Sets the mask for the specified interrupt.

Function	Description
acsc_ SetCallbackMaskExt	Sets the mask for the specified interrupt, extended to 128 axes.
acsc_GetCallbackMask	Retrieves the mask for the specified interrupt.
acsc_GetCallbackMask	Retrieves the mask for the specified interrupt, extended to 128 axes.
acsc_SetCallbackPriority	Sets the priority for all callback threads.

4.32.1 acsc_InstallCallback

Description

The function installs a user-defined callback function for the specified interrupt condition with user-defined parameter.

Syntax

int acsc_InstallCallback(HANDLE Handle,
ACSC_USER_CALLBACK_FUNCTION Callback, void* UserParameter,
int Interrupt)

Arguments

Handle	Communication handle.
Callback	A pointer to a user-defined function that accepts an argument of integer type and returns the result of integer type. Its prototype is: `int WINAPI Callback(unsigned _int64 Param, void* UserParameter);` where UserParameter is the same as in this function. If **Callback** is NULL, the function resets previous installed callback for the corresponding **Interrupt**. If **Callback** should have no functionality and is set only for corresponding acsc_Waitxxx function, this parameter may be set as `acsc_dummy_callback_ext`.
UserParameter	Additional parameter that will be passed to the callback function.
Interrupt	See Callback Interrupts.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function installs a user-defined callback function **Callback** for the specified interrupts condition **Interrupt**.

The library calls the **Callback** function when the specified interrupt occurs. The bit-mapped parameter **Param** of the function **Callback** identifies for which axis/buffer/input the interrupt was generated. See Callback Interrupt Masks for a detailed description of the parameter **Param** for each interrupt.

One callback can be installed for each interrupt, for same communication channel. The library creates a separate thread for each interrupt. Therefore, each callback function is called in a separate thread so that the callbacks do not delay one another.

User on Callback registration defines the parameter **UserParameter**. It is especially useful when several SPiiPlus cards are used. That allows setting the same function as a Callback on certain interrupt, for all the cards. Inside that function user can see by the **UserParameter,** which card caused the interrupt.

To uninstall a specific callback, call the function **acsc_InstallCallback** with the parameter **Callback** equals to NULL and the parameter **Interrupt** equals the specified interrupt type. This action will uninstall the callback for specified communication channel.

Example

```
int WINAPI CallbackInput (unsigned __int64 Param,void* UserParameter);
// will be defined later
…
int CardIndex;//Some external variable, which contains card index
// set callback function to monitor digital inputs
if (!acsc_InstallCallback(Handle,       // communication handle
                CallbackInput,          // pointer to the user callback function
                &CardIndex,             // pointer to the index
                ))
{
        printf("callback registration error: %d\n", acsc_GetLastError());
}
…
// If callback was installed successfully, the CallbackInput function
will
// be called each time the any digital input has changed from 0 to 1.
// CallbackInput function checks the digital inputs 0 and 1
int WINAPI CallbackInput (unsigned __int64 Param,void* UserParameter)
{
if (Param & ACSC_MASK_INPUT_0 && (*(int *)UserParameter == 0)
//Treat input 0 only for card with index 0
        {
                // input 0 has changed from 0 to 1
                // doing something here
        }
if (Param & ACSC_MASK_INPUT_1 && *UserParameter == 1)
//Treat input 1 only for card with index 1
        {
```

```
                // input 1 has changed from 0 to 1
                // doing something here
        }
return 0;
}
```

4.32.2 acsc_InstallCallbackExt

Description

The function installs a user-defined callback function for the specified interrupt condition with user-defined parameter, with extension to 128 axes.

Syntax

int **acsc_InstallCallBackExt** (HANDLE Handle, ACSC_USER_CALLBACK_FUNCTION_EXT Callback, void* UserParameter, int Interrupt)

Arguments

Handle	Communication handle
Callback	A pointer to a user-defined function that accepts an argument of **AXMASK_EXT** type and returns the result of integer type. Its prototype is: int WINAPI Callback(**AXMASK_EXT** Param,void* UserParameter); where UserParameter is the same as in this function. If Callbackis NULL, the function resets previously installed callback for the corresponding interrupt. If Callback should have no functionality and is set only for corresponding acsc_Waitxxx function, this parameter may be set as **ACS_DUMMY_CALLBACK_FUNCTION_EXT**.
UserParameter	Additional parameter that will be passed to the callback function.
Interrupt	See callback Interrupts.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Comments

This function extends the functionality of acsc_InstallCallback work with up to 128 axes.

Extended error information can be obtained by calling acsc_GetLastError.

Example

```
int WINAPI Callback_MotionStart128(AXMASK_EXT Param, void*
UserParameter);
//will be defined later

if (!acsc_InstallCallbackExt(hComm,    //communication handle
 Callback_MotionStart128,      //pointer to the user callback function
NULL,
 ACSC_INTR_MOTION_START))
{
       printf("Callback Registration Error: %d\n", acsc_GetLastError());
}
//The callback definition
int WINAPI Callback_MotionStart128(AXMASK_EXT Param, void* UserParameter)
{
       int i = 0;
       for (i = 0; Param.AXMASK64; i++, Param.AXMASK64 >>= 1)
       {
              if (Param.AXMASK64 & 1)
                     printf("Callback_MotionStart128(): Callback
recieved. Axis Index = %d\n",i);
       }
       for (i = 0; Param.AXMASK128; i++, Param.AXMASK128 >>= 1)
       {
              if (Param.AXMASK128 & 1)
                     printf("Callback_MotionStart128(): Callback recieved. Axis Inde
%d\n", i+64);
       }
       return 0;
}
```

4.32.3 acsc_SetCallbackMask

Description

The function sets the mask for specified callback.

Syntax

int acsc_SetCallbackMask(HANDLE Handle, int Interrupt, unsigned __int64 Mask)

Arguments

Handle	Communication handle
Interrupt	See Callback Interrupts

Mask	The mask to be set. If some bit equals to 0 then interrupt for corresponding axis/buffer/input does not occur – interrupt is disabled. Use ACSC_MASK_*** constants to set/reset a specified bit. See Callback Interrupt Masksfor a detailed description of these constants. If **Mask** is ACSC_NONE, the interrupts for all axes/buffers/inputs are enabled. If **Mask** is 0, the interrupts for all axes/buffers/inputs are disabled. As default all bits for each interrupts are set to one.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function sets the bit mask for specified interrupt. To get current mask for specified interrupt, call acsc_GetCallbackMask.

Using a mask, you can reduce the number of calls of your callback function. The callback will be called only if the interrupt is caused by an axis/buffer/input that corresponds to non-zero bit in the related mask.

Example

```
// the example shows how to configure the interrupt mask to monitor only
// specific axis
if (!acsc_SetCallbackMask(Handle, // communication handle
                          ACSC_INTR_PHYSICAL_MOTION_END,  // specified
interrupt
                          ACSC_MASK_AXIS_0        // enable interrupt only for
the axis 0                                  ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.32.4 acsc_SetCallbackMaskExt

Description

The function sets the mask for the specified callback, up to 128 axes.

Syntax

int **acsc_SetCallbackMaskExt** (HANDLE Handle, int interrupt, AXMASK_EXT Mask)

Arguments

Handle	Communication Handle
Interrupt	See Callback Interrupts
Mask	The mask to be set If a bit equals 0, the interrupt for corresponding axis/buffer/input does not occur – interrupt is disabled. Use ACSC_MASK_*** constants to set/reset a specific bit. If Mask.AXMASK64 is ACSC_NONE, the interrupts for all 0..63 axes/buffers/inputs are enabled. If Mask.AXMASK128 is ACSC_NONE, the interrupts for all 64..127 axes are enabled. If Mask.AXMASK64 is 0, the interrupts for all 0..63 axes/buffers/inputs are disabled. If Mask.AXMASK128 is 0, the interrupts for axes 64..127 axes are disabled. As default all bits for each interrupt are set to one.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Comments

Extended error information can be obtained by calling acsc_GetLastError

The function sets the bit mask for specified interrupt. To get current mask for specified interrupt, call **acsc_GetCallbackMaskExt**.

Example

```
//the example below shows how to configure the interrupt mask for monitor
axes 64..127
AXMASK_EXT Mask;
Mask.AXMASK64 = 0;
Mask.AXMASK128 = ACSC_NONE; //axes 64..127

if (!acsc_SetCallbackMaskExt(Handle,  //communication handle
        ACSC_INTR_MOTION_START, //specified interrupt
        Mask)                          //axes 64..127 are enabled
{
      printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.32.5 acsc_GetCallbackMask

Description

The function retrieves the mask for specified interrupt.

Syntax

int acsc_GetCallbackMask(HANDLE Handle, int Interrupt, unsigned __int64 *Mask)

Arguments

Handle	Communication handle
Interrupt	See Callback Interrupts.
Mask	The current mask. If a bit equals 0, an interrupt for corresponding axis/buffer/input does not occur; the interrupt is disabled. Use the ACSC_MASK_*** constants to analyze a value of the specific bit. See Callback Interrupt Masks for a detailed description of these constants. As default all bits for each interrupts are set to one.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the bit mask for specified interrupt. To set the mask for specified interrupt, use acsc_SetCallbackMask.

Example

```
// the example shows how to get the mask for specific interrupt
unsigned __int64 Mask;
if (!acsc_GetCallbackMask(      Handle,         // communication handle
                ACSC_INTR_PHYSICAL_MOTION_END,  // specified interrupt
                                        &Mask   // received value
                                        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.32.6 acsc_GetCallbackMaskExt

Description

The function retrieves the mask for specified interrupt, up to 128 axes.

Syntax

int **acsc_GetCallbackMaskExt** (HANDLE Handle, int interrupt, AXMASK_EXT* Mask)

Arguments

Handle	Communication handle
Interrupt	See Callback Interrupts
Mask	The current mask. If a bit equals to 0, an interrupt for corresponding axis/buffer/input does not occur – interrupt is disabled. Use ACSC_MASK_*** constants to analyze a value of the specific bit. By default all bits for each interrupt are set to one.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Comments

The function retrieves the bit mask for specified interrupt. To set current mask for specified interrupt, call **acsc_SetCallbackMaskExt**.

Extended error information can be obtained by calling acsc_GetLastError

Example

```
//the example below shows how to get the mask for specific interrupt
AXMASK_EXT Mask;

Mask.AXMASK64 = 0;
Mask.AXMASK128 = 0;
if(!acsc_GetCallbackMaskExt(Handle,  //communication handle
        ACSC_INTR_MOTION_START, //specified interrupt
        &Mask,                  //received value for
       ))
{
      printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.32.7 acsc_SetCallbackPriority

Description

The function sets the priority for all callback threads.

The function can be used only with the PCI communication channel.

Syntax

int acsc_SetCallbackPriority(HANDLE Handle, int Priority)

Arguments

Handle	Communication handle.
Priority	Specifies the priority value for the callback thread. This parameter can be one of the operating system defined priority levels. For example the Win32 API defines the following priorities: THREAD_PRIORITY_ABOVE_NORMAL THREAD_PRIORITY_BELOW_NORMAL THREAD_PRIORITY_HIGHEST THREAD_PRIORITY_IDLE THREAD_PRIORITY_LOWEST THREAD_PRIORITY_NORMAL THREAD_PRIORITY_TIME_CRITICAL As default, all callback threads have a normal priority.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The operating system uses the priority level of all executable threads to determine which thread gets the next slice of CPU time. Basically, threads are scheduled in a round-robin fashion at each priority level, and only when there are no executable threads at a higher level does scheduling of threads at a lower level take place.

When manipulating priorities, be very careful to ensure that a high-priority thread does not consume all of the available CPU time. See the relevant operating system guides for details.

Example

```
// the example sets the new priority for all callbacks
if (!acsc_SetCallbackPriority(Handle, THREAD_PRIORITY_ABOVE_NORMAL))
{
        printf("set of callback priority error: %d\n", acsc_GetLastError());
}
```

4.33 Variables Management Functions

The Variables Management functions are:

Table 5-30. Variables Management Functions

Function	Description
acsc_DeclareVariable	Creates the persistent global variable.
acsc_ClearVariables	Deletes all persistent global variables.

4.33.1 acsc_DeclareVariable

Description

The function creates the persistent global variable.

Syntax

int acsc_DeclareVariable (HANDLE Handle, int Type, char* Name,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Type	Type of the variable. For an integer variable the type parameter must be **ACSC_INT_TYPE**. For a real variable, the type parameter must be **ACSC_REAL_TYPE**. For a static integer variable the type parameter must be **ACSC_STATIC_INT_TYPE** For a static real variable, the type parameter must be **ACSC_STATIC_REAL_TYPE**
Name	Pointer to the null-terminated ASCII string contained name of the variable.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function acsc_WaitForAsyncCall returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function creates the persistent global variable specified by the parameter **Name** of type specified by the parameter **Type**. The variable can be used as any other standard or global variable.

If it is necessary to declare one or two-dimensional array, the parameter **Name** should also contains the dimensional size in brackets.

The lifetime of a persistent global variable is not connected with any program buffer. The persistent variable survives any change in the program buffers and can be erased only by the acsc_ClearVariables function.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the declaration of scalar variable
acsc_DeclareVariable(Handle,    // communication handle
                ACSC_INT_TYPE,          // integer type
                "MyVar",                // name of the variable
                NULL                    // waiting call
                ));
// example of the declaration of one-dimensional array
acsc_DeclareVariable(Handle,            // communication handle
                ACSC_INT_TYPE,          // integer type
                "MyArr(10)",            // name of the one-dimensional
                                        // array of 10 elements
                NULL                    // waiting call
                ));
// example of the declaration of matrix
acsc_DeclareVariable(Handle,            // communication handle
                ACSC_REAL_TYPE,         // real type
                "MyMatrix(10)(5)",      // name of the matrix of 10 rows
                                        // and 5 columns
                NULL                    // waiting call
                ));
```

4.33.2 acsc_ClearVariables

Description

The function deletes all persistent global variables.

Syntax

int acsc_ClearVariables (HANDLE Handle, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Wait	Pointer to ACSC_WAITBLOCK structure.

	If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function deletes all persistent global variables created by the acsc_DeclareVariable function.

The function can wait for the controller response or can return immediately as specified by the **Wait** argument.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Wait** item until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_ClearVariables
if (!acsc_ClearVariables(Handle,                // communication handle
                         NULL                   // waiting call
                         ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.34 Service Functions

The Service functions are:

Table 5-31. Service Functions

Function	Description
acsc_ GetFirmwareVersion	Retrieves the firmware version of the controller.
acsc_GetSerialNumber	Retrieves the controller serial number.

Function	Description
acsc_GetBuffersCount	The function returns the number of available ACSPL+ programming buffers.
acsc_GetAxesCount	The function returns the number of axes defined in the system.
acsc_GetDBufferIndex	The function returns the index of the D-Buffer.
acsc_GetUMDVersion	The function returns the version of the UMD

4.34.1 acsc_GetFirmwareVersion

Description

The function retrieves the firmware version of the controller.

Syntax

int acsc_GetFirmwareVersion(HANDLE Handle, char* Version, int Count,
int* Received, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Version	Pointer to the buffer that receives the firmware version.
Count	Size of the buffer pointed by **Version.**
Received	Number of characters that were actually received.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the character string that contains the firmware version of the controller. The function will not copy more than **Count** characters to the **Version** buffer. If the buffer is too small, the firmware version can be truncated.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Version, Received,** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetFirmwareVersion
char Firmware[256];
int Received;
if (!acsc_GetFirmwareVersion(   Handle,         // communication handle
                                Firmware,       // buffer for the firmware
                                                // version
                                255,            // size of the buffer
                                &Received,      // number of actually
                                                // received characters
                                NULL            // waiting call
                                ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
else
{
        Firmware[Received] = '\0';
        printf("Firmware version of the controller: %s\n", Firmware);
}
```

4.34.2 acsc_GetSerialNumber

Description

The function retrieves the controller serial number.

Syntax

int acsc_GetSerialNumber(HANDLE Handle, char* SerialNumber, int Count,
int* Received, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
SerialNumber	Pointer to the buffer that receives the serial number.
Count	Size of the buffer pointed by **SerialNumber.**
Received	Number of characters that were actually received.

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the character string that contains the controller serial number. The function will not copy more than **Count** characters to the **SerialNumber** buffer. If the buffer is too small, the serial number can be truncated.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **SerialNumber**, **Received** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetSerialNumber
char SerialNumber[256];
int Received;
if (!acsc_GetSerialNumber(Handle,       // communication handle
        SerialNumber,           // buffer for the serial number
        255,                    // size of the buffer
        &Received,              // number of actually received characters
        NULL                    // waiting call
        ))
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
else
{
        SerialNumber[Received] = '\0';
        printf("Controller serial number: %s\n", SerialNumber);
}
```

4.34.3 acsc_GetBuffersCount

Description

The function returns the number of available ACSPL+ programming buffers.

Syntax

int acsc_GetBuffersCount(HANDLE Handle, double *Value,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Value	Receives the number of available ACSPL+ programming buffers.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
double Value = 0;
if (!acsc_GetBuffersCount(Handle, &Value, NULL))
{
        printf("acsc_GetBuffersCount(): Error Occurred - %d\n",
        acsc_GetLastError());
        return;
}
```

4.34.4 acsc_GetAxesCount

Description

The function returns the number of available axes.

Syntax

int acsc_GetAxesCount(HANDLE Handle, double *Value, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Value	Receives the number of available axes.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
double Value = 0;
if (!acsc_GetAxesCount(Handle, &Value, NULL))
{
        printf("acsc_GetAxesCount(): Error Occurred - %d\n",
        acsc_GetLastError());
        return;
}
```

4.34.5 acsc_GetDBufferIndex

Description

The function returns the index of the D-Buffer.

Syntax

int acsc_GetDBufferIndex(HANDLE Handle, double *Value,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Value	Receives the index of D-Buffer.

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
double Value = 0;
if (!acsc_GetDBufferIndex(Handle, &Value, NULL))
{
        printf("acsc_GetDBufferIndex(): Error Occurred - %d\n",
        acsc_GetLastError());
        return;
}
```

4.34.6 acsc_GetUMDVersion

Description

The function retrieves the SPiiPlus User Mode Driver version number.

Syntax

unsigned int acsc_GetUMDVersion()

Arguments

This function has no arguments.

Return Value

The return value is the 32-bit unsigned integer value, which specifies the binary version number.

Comments

The SPiiPlus User Mode Driver version consists of four (or fewer) numbers separated by points: #.#.#.#. The binary version number is represented by 32-bit unsigned integer value. Each byte of this value specifies one number in the following order: high byte of high word – first number, low byte of high word – second number, high byte of low word – third number and low byte of low word – forth number. For example version "2.10" has the following binary representation (hexadecimal format): 0x020A0000.

First two numbers in the string form are obligatory. Any release version of the library consists of two numbers. The third and fourth numbers specify an alpha or beta version, special or private build, or other version.

Example

```
unsigned int Ver = acsc_GetUMDVersion();
printf("SPiiPlus User Mode Driver version is %d.%d.%d.%d\n", HIBYTE
(HIWORD(Ver)),
        LOBYTE(HIWORD(Ver)), HIBYTE(LOWORD(Ver)), LOBYTE(LOWORD(Ver)));
```

4.35 Error Diagnostic Functions

The Error Diagnostic functions are:

Table 5-32. Error Diagnostic Functions

Function	Description
acsc_GetMotorError	Retrieves the reason why the motor was disabled.
acsc_ GetProgramError	Retrieves the error code of the last program error encountered in the specified buffer.
acsc_ GetEtherCATError	Used to retrieve the last EtherCAT error.

4.35.1 acsc_GetMotorError

Description

The function retrieves the reason for motor disabling.

Syntax

int acsc_GetMotorError(HANDLE Handle, int Axis, int* Error,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions.
Error	Pointer to a variable that receives the reason why the motor was disabled.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result.

	If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the reason for motor disabling.

If the motor is enabled the parameter, **Error** is zero. If the motor was disabled, the parameter **Error** contains the reason for motor disabling.

To get the error explanation, use the acsc_GetErrorString function.

See *SPiiPlus ACSPL+ Programmer's Guide* for all available motor error code descriptions.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Error** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetMotorError
int Error;
char ErrorStr[256];
int Received;
if (acsc_GetMotorError(Handle,  // communication handle
        ACSC_AXIS_0,            // axis 0
        &Error,                 // received value
        NULL                    // waiting call
        ))
{
        if (Error > 0)
        {
                if (acsc_GetErrorString(Handle, Error, ErrorStr, 255,
                                                &Received))
                {
                        ErrorStr[Received] = '\0';
                        printf("Motor error: %d (%s)\n", Error, ErrorStr);
                }
        }
}
else
```

```
{
        if (Error > 0)
        {
                if (acsc_GetErrorString(Handle, Error, ErrorStr, 255,
                                        &Received))
                {
                        ErrorStr[Received] = '\0';
                        printf("Motor error: %d (%s)\n", Error, ErrorStr);
                }
        }
}
else
{
        printf("transaction error: %d\n", acsc_GetLastError());
}
```

4.35.2 acsc_GetProgramError

Description

The function retrieves the error code of the last program error encountered in the specified buffer.

Syntax

int acsc_GetProgramError(HANDLE Handle, int Buffer, int* Error,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Buffer	Number of the program buffer.
Error	Pointer to a variable that receives the error code.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The function retrieves the error code of the last program error encountered in the specified buffer.

If the program is running, the parameter **Error** is zero. If the program terminates for any reason, the parameter **Error** contains the termination code.

To get the error explanation, use the acsc_GetErrorString function.

If **Wait** points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the **Error** and **Wait** items until a call to the acsc_WaitForAsyncCall function.

Example

```
// example of the waiting call of acsc_GetProgramError
int Error;
char ErrorStr[256];
if (acsc_GetProgramError(Handle, // communication handle
                0,               // buffer 0
                &Error,          // received value
                NULL             // waiting call
                ))
{
        if (Error > 0)
        {
                if (acsc_GetErrorString(        Handle, Error, ErrorStr, 255,
                                                &Received))
                {
                                                ErrorStr[Received] = '\0';
        printf("Program error: %d (%s)\n", Error, ErrorStr);
                }
        }
}
else
        {
                printf("transaction error: %d\n", acsc_GetLastError());
        }
```

4.35.3 acsc_GetEtherCATError

Description

The function is used to retrieve the last EtherCAT error.

Syntax

int acsc_GetEtherCATError(HANDLE Handle, int* Error,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Error	Pointer to a variable that receives the last EtherCAT error that has occurred (see ***Comments***).
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Comments

Any EtherCAT error sets **ECST.#OP** to false and the error code is latched in the **ECERR** variable.The EtherCAT errors are given in Table 5-33.

Table 5-33. EtherCAT Errors

Error Code	Error Message	Remarks
6000	General EtherCAT Error	Appears for any unspecified EtherCAT error. In general, it rarely appears.
6001	EtherCAT cable not connected	Check that the EtherCAT connections are firmly seated.
6002	EtherCAT master is in incorrect state	On start up all slaves did not succeed to initialize to full OP state. Can be caused by wrong configuration or a problem in a Slave
6003	Not all EtherCAT frames can be processed	The Master has detected that at least one frame that was sent has not returned. This implies a hardware problem in the cables or Slaves.

Error Code	Error Message	Remarks
6004	EtherCAT Slave error	Slave did not behave according to EtherCAT state machine – internal Slave failure.
6005	EtherCAT initialization failure	The EtherCAT-related hardware in the Master could not be initialized. Check the EtherCAT hardware.
6006	EtherCAT cannot complete the operation	The bus scan could not be completed. This implies hardware level problems in the EtherCAT network.
6007	EtherCAT work count error	Every Slave increments the working counter in the telegram. If this error is triggered, it means that a Slave has failed. Possible root cause: cable interruption, Slave reset, Slave hardware failure,
6008	Not all EtherCAT slaves are operational	One or more of the Slaves has changed its state to other than OP, or it may be due to a Slave restart or internal fault that internally forces the Slave to go to PREOP or SAFEOP.
6009	EtherCAT protocol timeout	The Master has detected that the Slave does not behave as expected for too long, and reports timeout. Implies a Slave hardware problem. Try power down, and system restart.
6010	Slave initialization failed	The Master cannot initialize a Slave by the configuration file. It can be caused by either wrong configuration, or a hardware problem in the Slave.
6011	Bus configuration mismatch	The bus topology differs from that in configuration file.
6012	CoE emergency	A Slave with CoE support has reported an emergency message.
6013	EtherCAT Slave won't enter INIT state	Hardware fault, for example DHD with broken (logic) supply.

Example

```
int Value = 0;
if (!acsc_GetEtherCATError(Handle, &Value, NULL))
{
        printf("acsc_GetEtherCATError(): Error Occurred - %d\n",
        acsc_GetLastError());
```

```
        return;
}
```

4.36 Dual Port RAM (DPRAM) Access Functions

The Dual Port RAM Access functions are:

Table 5-34. Dual Port RAM (DPRAM) Access Functions

Function	Description
acsc_ReadDPRAMInteger	Reads 32-bit integer from DPRAM
acsc_WriteDPRAMInteger	Writes 32-bit integer to DPRAM
acsc_ReadDPRAMReal	Reads 64 real from DPRAM
acsc_WriteDPRAMReal	Writes 64-bit real to DPRAM

DPRAM is not supported in SPiiPlus products.

4.36.1 acsc_ReadDPRAMInteger

Description

The function reads 32-bit integer from the DPRAM.

Syntax

acsc_ReadDPRAMInteger(HANDLE Handle, int address,int *Value)

Arguments

Handle	Communication handle
Address	Address has to be even number from 128 to 1020 (DRAM size is 1024)
Value	Value that was read from DPRAM

Return Value

The function returns non-zero on success.

If the value cannot be read, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

Address has to be even number in the range of 128 to 508, because we use 16-bit alignment when working with DPRAM. Addresses less than 128 are used for internal purposes.

Example

```
acsc_ReadDPRAMInteger(  Handle,          // communication handle
                        0xA0,            // DPRAM address
                        &Value           //Value that receives the result
                        )
```

4.36.2 acsc_WriteDPRAMInteger

Description

The function writes 32-bit integer to the DPRAM.

Syntax

acsc_WriteDPRAMInteger(HANDLE Handle, int address,int Value)

Arguments

Handle	Communication handle
Address	Address has to be even number from 128 to 1020 (DRAM size is 1024)
Value	Value to be written

Return Value

The function returns non-zero on success.

If the value cannot be written, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

Address has to be even number in the range of 128 to 508, because we use 16-bit alignment when working with DPRAM. Addresses less than 128 are used for internal purposes.

Example

```
acsc_WriteDPRAMInteger(Handle,  // communication handle
                0xA0,           // DPRAM address
                0               //Value to write
                )
```

4.36.3 acsc_ReadDPRAMReal

Description

The function reads 64-bit Real from the DPRAM.

Syntax

acsc_ReadDPRAMReal(HANDLE Handle, int address, double *Value)

Arguments

Handle	Communication handle
Address	Address has to be even number from 128 to 1020 (DRAM size is 1024)
Value	Value that was read from DPRAM

Return Value

The function returns non-zero on success.

If the value cannot be read, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

Address has to be even number in the range of 128 to 504, because we use 16-bit alignment when working with DPRAM. Addresses less than 128 are used for internal purposes.

Example

```
acsc_ReadDPRAMReal(     Handle,         // communication handle
                        0xA0,           // DPRAM address
                        &Value          //Value that receives the result
                        )
```

4.36.4 acsc_WriteDPRAMReal

Description

The function writes 64-bit Real to the DPRAM.

Syntax

acsc_WriteDPRAMReal(HANDLE Handle, int address, double Value)

Arguments

Handle	Communication handle
Address	Address has to be even number from 128 to 1020 (DRAM size is 1024)
Value	Value to be written

Return Value

The function returns non-zero on success.

If the value cannot be written, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

Address has to be even number in the range of 128 to 504, because we use 16-bit alignment when working with DPRAM. Addresses less than 128 are used for internal purposes.

Example

```
acsc_WriteDPRAMReal(Handle,     // communication handle
                0xA0,           // DPRAM address
                0               //Value to write
                )
```

4.37 Shared Memory Functions

The Shared Memory functions have been added in support of SPiiPlus SC to enable accessing memory addresses. They cannot be used with any other SPiiPlus family product.

The Shared Memory functions are:

Table 5-35. Shared Memory Functions

Function	Description
acsc_GetSharedMemoryAddress	Reads the address of shared memory variable.
acsc_ReadSharedMemoryInteger	Reads value(s) from an integer shared memory variable.
acsc_WriteSharedMemoryInteger	Writes value(s) to the integer shared memory variable.
acsc_ReadSharedMemoryReal	Reads value(s) from a real shared memory variable.
acsc_WriteSharedMemoryReal	Writes value(s) to the real shared memory variable.

4.37.1 acsc_GetSharedMemoryAddress

Description

The function reads the address of shared memory variable.

Syntax

int acsc_GetSharedMemoryAddress(HANDLE Handle, int NBuf, char* Var,
unsigned int* Address, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle
NBuf	Number of program buffer for local variable or ACSC_NONE for global and standard variable.

Var	Pointer to a null-terminated character string that contains a name of the variable.
Address	Pointer to the variable that receives the address of specified variable.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

4.37.2 acsc_ReadSharedMemoryInteger

Description

The function reads value(s) from an integer shared memory variable.

Syntax

int acsc_ReadSharedMemoryInteger(HANDLE Handle, unsigned int Address,
int From1, int To1, int From2, int To2, int* Values)

Arguments

Handle	Communication handle
Address	Shared memory address of the variable that should be read
From1, To1	Index range (first dimension).
From2, To2	Index range (second dimension).
Values	Pointer to the buffer that receives requested values.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

4.37.3 acsc_WriteSharedMemoryInteger

Description

The function writes value(s) to the integer shared memory variable.

Syntax

int acsc_WriteSharedMemoryInteger(HANDLE Handle, unsigned int Address,
int From1, int To1, int From2, int To2, int* Values)

Arguments

Handle	Communication handle
Address	Shared memory address of the variable that should be read
From1, To1	Index range (first dimension).
From2, To2	Index range (second dimension).
Values	Pointer to the buffer contained values that must be written.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

4.37.4 acsc_ReadSharedMemoryReal

Description

The function reads value(s) from a real shared memory variable.

Syntax

int acsc_ReadSharedMemoryReal(HANDLE Handle, unsigned int Address, int From1,
int To1, int From2, int To2, double* Values)

Arguments

Handle	Communication handle
Address	Shared memory address of the variable that should be read
From1, To1	Index range (first dimension).
From2, To2	Index range (second dimension).
Values	Pointer to the buffer that receives requested values.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

4.37.5 acsc_WriteSharedMemoryReal

Description

The function writes value(s) to the real shared memory variable.

Syntax

int acsc_WriteSharedMemoryReal(HANDLE Handle, unsigned int Address, int From1, int To1, int From2, int To2, double* Values)

Arguments

Handle	Communication handle
Address	Shared memory address of the variable that should be read
From1, To1	Index range (first dimension).
From2, To2	Index range (second dimension).
Values	Pointer to the buffer contained values that must be written.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

4.37.6 Shared Memory Program Example

The following program example demonstrates a use of the Shared Memory functions.

The D-Buffer should contain the definition of the relevant shared memory variable:
global real shm HELLO_VAR(2)(2)

```
unsigned int Address = 0;
double data[2][2] = { {1.112, 2.334}, {4.565, 7.456} };
double read_data[2][2] = {0};
if (!acsc_GetSharedMemoryAddress(Handle, ACSC_NONE,     "HELLO_VAR",
&Address, NULL))
{
printf("Error Getting Address: %d\n", acsc_GetLastError());
        return;
}
if (!acsc_WriteSharedMemoryReal(Handle, Address, 0, 1, 0, 1,
        &(data[0][0])))
{
        printf("Error Reading Variable: %d\n", acsc_GetLastError());
        return;
}
if (!acsc_ReadSharedMemoryReal(Handle, Address, 0, 1, 0, 1,
        &(read_data[0][0])))
{
        printf("Error Reading Variable: %d\n", acsc_GetLastError());
        return;
}
if ((data[0][0] != read_data[0][0]) || (data[0][1] !=
```

```
read_data[0][1]) ||
(data[1][0] != read_data[1][0]) || (data[1][1] !=
read_data[1][1]))
{
        printf("Read data is not equal to written data.\n");
        return;
}
```

4.38 EtherCAT Functions

The EtherCAT functions can be used only with the SPiiPlus family of products

The C Library EtherCAT Functions are:

Table 5-36. EtherCAT Functions

Function	Description
acsc_GetEtherCATState	Used to retrieve the EtherCAT state.
acsc_MapEtherCATInput	Used for raw mapping of network input variables.
acsc_MapEtherCATOutput	Used for raw mapping of network output variables.
acsc_ UnmapEtherCATInputsOutputs	Used for unmapping previously mapped inputs or outputs.
acsc_GetEtherCATSlaveIndex	Used for obtaining the index of an EtherCAT slave.
acsc_GetEtherCATSlaveOffsetV2	Used for obtaining the offset of an EtherCAT slave.
acsc_ GetEtherCATSlaveVendorIDV2	Used for obtaining the Vendor ID of an EtherCAT slave.
acsc_ GetEtherCATSlaveProductIDV2	Used for obtaining the Product ID of an EtherCAT slave.
acsc_GetEtherCATSlaveRevisionV2	Used for obtaining the Revision number of an EtherCAT slave.
acsc_GetEtherCATSlaveType	Used for obtaining the type of an EtherCAT slave.
acsc_GetEtherCATSlaveStateV2	Used for obtaining the machine state of an EtherCAT slave.
acsc_GetEtherCATSlavesCount	Obtain number of slave drives

4.38.1 acsc_GetEtherCATState

Description

The function is used to retrieve the EtherCAT state.

Syntax

int acsc_GetEtherCATState(HANDLE Handle, int* State, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
State	Pointer to a variable that receives EtherCAT State (see ***Comments***).
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Comments

The EtherCAT state is contained in the ACSPL+ **ECST** variable. The EtherCAT state is reflected in the first six bits as given in Table 5-37.

Table 5-37. ECST Bits

Bit	Designator	Description
0	#SCAN	The scan process was performed successfully, that is, the Master was able to detect what devices are connected to it.
1	#CONFIG	There is no deviation between XML and actual setup. The Master succeeded to initialize the network by steps described in configuration file.
2	#INITOK	All bus devices are successfully set to INIT state. The Master started all devices to the initial state.
3	#CONNECTED	Indicates valid Ethernet cable connection to the master. The physical link of EtherCAT cable is OK on the Master side.

Bit	Designator	Description
4	#INSYNC	If DCM is used, indicates synchronization between the Master and the bus.
5	#OP	The EtherCAT bus is operational. The Master successfully turned each Slave into full operational mode and the bus is ready for full operation.
6	#DCSYNC	Distributed clocks are synchronized.
7	#RINGMODE	Ring Topology mode status
8	#RINGCOMM	Ring Communication active status
9	#EXTCONN	External clock is connected
10	#DCXSYNC	External clock/slaves are synchronized

All bits (except **#INSYNC** in some cases) should be true for proper bus functioning, for monitoring the bus state, checking bit **#OP** is enough. Any bus error will reset the **#OP** bit.

Example

```
int Value = 0;
if (!acsc_GetEtherCATState(Handle, &Value, NULL))
{
        printf("acsc_GetEtherCATState(): Error Occurred - %d\n",
        acsc_GetLastError());
        return;
}
```

4.38.2 acsc_MapEtherCATInput

Description

The function is used for raw mapping of network input variables of any size. 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.

The function call is legal only when the EtherCAT state is operable (that is, **ECST.OP=1**).

Syntax

int acsc_MapEtherCATInput(HANDLE Handle, int Flags, int Offset,
char* VariableName, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Flags	Bit-mapped parameter. Currently the value should be 0.
Offset	Internal EtherCAT offset of network input variable extracted from the SPiiPlus MMI Application Studio Communication Terminal **#ETHERCAT** command.
VariableName	Valid name of ACSPL+ variable, global or standard.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
if (!acsc_MapEtherCATInput(Handle, 0, 122, "I0", NULL))
{
        printf("acsc_MapEtherCATInput(): Error Occurred - %d\n",
        acsc_GetLastError());
        return;
}
```

4.38.3 acsc_MapEtherCATOutput

Description

The function is used for raw mapping of network output variables of any size. Once the function is called successfully, the firmware copies the value of specified ACSPL+ variable into the network output variable at the corresponding EtherCAT offset, every controller cycle.

The function call is legal only when the EtherCAT state is operable (that is, **ECST.OP**=**1**).

Syntax

int acsc_MapEtherCATOutput(HANDLE Handle, int Flags, int Offset,
char* VariableName, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Flags	Bit-mapped parameter. Currently the value should be 0.
Offset	Internal EtherCAT offset of network output variable extracted from the SPiiPlus MMI Communication Terminal **#ETHERCAT** command.
VariableName	Valid name of ACSPL+ variable, global or standard.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
if (!acsc_MapEtherCATOutput(Handle, 0, 296, "I1", NULL))
{
        printf("acsc_MapEtherCATOutput(): Error Occurred - %d\n",
        acsc_GetLastError());
        return;
}
```

4.38.4 acsc_UnmapEtherCATInputsOutputs

Description

The function resets all previous mapping defined by acsc_MapEtherCATInput and acsc_MapEtherCATOutput functions.

The function call is legal only when the EtherCAT state is operable (that is, **ECST.OP=1**).

Syntax

int acsc_UnmapEtherCATInputsOutputs(HANDLE Handle,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
if (!acsc_UnmapEtherCATInputsOutputs(Handle, NULL))
{
        printf("acsc_UnmapEtherCATInputsOutputs(): Error Occurred - %d\n",
        acsc_GetLastError());
        return;
}
```

4.38.5 acsc_GetEtherCATSlaveIndex

Description

The function returns the index of EtherCAT slave according to the parameters that are specified.

Syntax

int acsc_GetEtherCATSlaveIndex(HANDLE Handle, int VendorID, int ProductID,
int Count, int* SlaveIndex, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
VendorID	EtherCAT Vendor ID of the slave device.
ProductID	EtherCAT Product ID of the slave device.

Count	Internal count of the device within those devices having the same Product and Vendor IDs.
SlaveIndex	Pointer to a variable that receives the index of the EtherCAT slave.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
double SlaveIndex = 0;
if (!acsc_GetEtherCATSlaveIndex(Handle, 0x00000540, 0x02040000, 1,
        &SlaveIndex, NULL))
{
        printf("acsc_GetEtherCATSlaveIndex(): Error Occurred - %d\n",
        acsc_GetLastError());
        return;
}
```

4.38.6 acsc_GetEtherCATSlaveOffsetV2

Description

The function returns offset of a network variable of the specified EtherCAT slave in specific ECAT network.

Syntax

```
int acsc_GetEtherCATSlaveOffsetV2(HANDLE Handle, int Flags, char*
VariableName, int SlaveIndex, double* SlaveOffset, ACSC_WAITBLOCK* Wait)
```

Arguments

Handle	Communication handle.
Flags	Bit-mapped argument that can include one or more of the following flags:

	ACSC_BIT_OFFSET - bit offset of the variable ACSC_ETHERCAT_NETWORK_0 - first network specifier ACSC_ETHERCAT_NETWORK_1 - second network specifier ACSC_ETHERCAT_NETWORK_0 and ACSC_ETHERCAT_NETWORK_1 flags must not be specified together 0, //No flags are set (equivalent to flag ACSC_ETHERCAT_NETWORK_0)
VariableName	Name of the EtherCAT network variable.
SlaveIndex	Index of the required EtherCAT slave, can be determined by acsc_GetEtherCATSlaveIndex function.
SlaveOffset	Pointer to a variable that receives the offset of the VariableName network variable of specified EtherCAT slave.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero

If the function fails, the return value is zero.

Extended error information can be obtained by calling `acsc_GetLastError`.

Example

```
double Value = 0;
if (!acsc_GetEtherCATSlaveOffsetV2(Handle, ACSC_ETHERCAT_NETWORK_0,
"DIGITAL_INPUTS", (int)SlaveIndex, &Value, NULL))
{
   printf("acsc_GetEtherCATSlaveOffsetV2(): Error Occurred - %d\n",
      acsc_GetLastError());
   return;
}
```

4.38.7 acsc_GetEtherCATSlaveVendorIDV2

Description

The function returns the Vendor ID of the specified EtherCAT slave in the specific ECAT network.

Syntax

int acsc_GetEtherCATSlaveVendorIDV2(HANDLE Handle, int Flags, int SlaveIndex, double* VendorID, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Flags	Bit-mapped argument that can include one or more of the following flags: ACSC_ETHERCAT_NETWORK_0 - first network specifier ACSC_ETHERCAT_NETWORK_1 - second network specifier ACSC_ETHERCAT_NETWORK_0 and ACSC_ETHERCAT_NETWORK_1 flags must not be specified together 0, //No flags are set (equivalent to flag ACSC_ETHERCAT_NETWORK_0)
SlaveIndex	Index of the required EtherCAT slave, can be determined by acsc_ GetEtherCATSlaveIndex function.
VendorID	Pointer to a variable that receives the vendor ID of the specified EtherCAT slave.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero

If the function fails, the return value is zero.

Extended error information can be obtained by calling `acsc_GetLastError`.

Example

```
double VendorID = 0;
if (!acsc_GetEtherCATSlaveVendorIDV2(Handle, ACSC_ETHERCAT_NETWORK_0,
(int)SlaveIndex, &VendorID, NULL))
{
   printf("acsc_GetEtherCATSlaveVendorIDV2():Error Occurred -%d\n",
   acsc_GetLastError());
```

```
   return;
}
```

4.38.8 acsc_GetEtherCATSlaveProductIDV2

Description

The function returns the Product ID of the specified EtherCAT slave in the specific EtherCAT network.

Syntax

```
int acsc_GetEtherCATSlaveProductIDV2(HANDLE Handle, int Flags, int
SlaveIndex, double* ProductID, ACSC_WAITBLOCK* Wait)
```

Syntax

int acsc_GetEtherCATSlaveProductIDV2(HANDLE Handle, int Flags, int SlaveIndex, double* ProductID, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Flags	Bit-mapped argument that can include one or more of the following flags: ACSC_ETHERCAT_NETWORK_0 - first network specifier ACSC_ETHERCAT_NETWORK_1 - second network specifier ACSC_ETHERCAT_NETWORK_0 and ACSC_ETHERCAT_NETWORK_1 flags must not be specified together 0, //No flags are set (equivalent to flag ACSC_ETHERCAT_NETWORK_0)
SlaveIndex	Index of the required EtherCAT slave, can be determined by acsc_ GetEtherCATSlaveIndex function.
ProductID	Pointer to a variable that receives the Product ID of the specified EtherCAT slave.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero

If the function fails, the return value is zero.

Extended error information can be obtained by calling `acsc_GetLastError`.

Example

```
double ProductID = 0;
if (!acsc_GetEtherCATSlaveProductIDV2(Handle, ACSC_ETHERCAT_NETWORK_0 ,
(int)SlaveIndex, &ProductID, NULL))
{
   printf("acsc_GetEtherCATSlaveProductIDV2(): Error Occurred - %d\n",
   acsc_GetLastError());
   return;
}
```

4.38.9 acsc_GetEtherCATSlaveRevisionV2

Description

The function returns the revision of the specified EtherCAT slave in the specific EtherCAT network.

Syntax

```
int acsc_GetEtherCATSlaveRevisionV2(HANDLE Handle, int Flags, int
SlaveIndex, double* Revision, ACSC_WAITBLOCK* Wait)
```

Arguments

Handle	Communication handle.
Flags	Bit-mapped argument that can include one or more of the following flags: ACSC_ETHERCAT_NETWORK_0 - first network specifier ACSC_ETHERCAT_NETWORK_1 - second network specifier ACSC_ETHERCAT_NETWORK_0 and ACSC_ETHERCAT_NETWORK_1 flags must not be specified together 0, //No flags are set (equivalent to flag ACSC_ETHERCAT_NETWORK_0)
SlaveIndex	index of the required EtherCAT slave, can be determined by acsc_GetEtherCATSlaveIndex function.
Revision	Pointer to a variable that receives the Revision of the specified EtherCAT slave.

Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

Return Value

If the function succeeds, the return value is non-zero

If the function fails, the return value is zero.

Extended error information can be obtained by calling `acsc_GetLastError`.

Example

```
double Revision = 0;
if (!acsc_GetEtherCATSlaveRevisionV2(Handle, ACSC_ETHERCAT_NETWORK_0 ,
(int)SlaveIndex, &Revision,NULL))
{
   printf("acsc_GetEtherCATSlaveRevisionV2(): Error Occurred - %d\n",
   acsc_GetLastError());
   return;
}
```

4.38.10 acsc_GetEtherCATSlaveType

Description

The function returns the type of the specified EtherCAT slave.

Syntax

int acsc_GetEtherCATSlaveType(HANDLE Handle, int VendorID, int ProductID,
double* SlaveType, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
VendorID	EtherCAT Vendor ID of the slave device.
ProductID	Pointer to a variable that receives the Product ID of the specified EtherCAT slave.
SlaveType	Pointer to a variable that receives the type of specified EtherCAT slave: 0 - ACS device

	1 - non-ACS servo 2 - non-ACS stepper 3 - non-ACS general -1 - Device not found at slave index
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
double Value = 0;
if (!acsc_GetEtherCATSlaveType(Handle, (int)VendorID, (int)ProductID,
        &Value, NULL))
{
        printf("acsc_GetEtherCATSlaveType(): Error Occurred - %d\n",
        acsc_GetLastError());
        return;
}
```

4.38.11 acsc_GetEtherCATSlaveRegister NEW

Description

The function returns value of the ESC Error Counters Registers of specified EtherCAT slave in specific EtherCAT network.

Syntax

```
int acsc_GetEtherCATSlaveRegister   (HANDLE Handle, int Flags, int
SlaveIndex, int Offset, double *Value, ACSC_WAITBLOCK* Wait)
```

Arguments

Handle	Communication handle.
Flags	Bit-mapped argument that can include one or more of the following flags: ACSC_ETHERCAT_NETWORK_0 - first network specifier ACSC_ETHERCAT_NETWORK_1 - second network specifier ACSC_ETHERCAT_NETWORK_0 and ACSC_ETHERCAT_NETWORK_1 flags must not be specified together 0, //No flags are set (equivalent to flag ACSC_ETHERCAT_NETWORK_0)
SlaveIndex	Index of the required EtherCAT slave, can be determined by acsc_ GetEtherCATSlaveIndex function.
Offset	Register offset in the ESC memory.
Value	Error counter registers
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Comments

The following table lists supported error counter registers.

Table 5-38. EtherCAT Error Counter Registers

Offset	Name	Description
0x300	Port Error Counter (CRC A)	Error Counted at the Auto-Forwarded (per port). Each register contains two counters: > Invalid Frame Counter: 0x300/2/4/6 > RX Error Counter: 0x301/3/5/7
0x302	Port Error Counter (CRC B)	

Offset	Name	Description
0x304	Port Error Counter (CRC C)	
0x306	Port Error Counter (CRC D)	
0x308	Forwarded RX Error Counter (CRC A/B)	Invalid frame with marking from previous ESC detected (per port).
0x309	Forwarded RX Error Counter	
0x30A	Forwarded RX Error Counter (CRC C/D)	
0x30B	Forwarded RX Error Counter	
0x30C	ECAT Processing Unit Error Counter	Invalid frame passing the EtherCAT Processing Unit (additional checks by processing unit).
0x30D	PDI Error Counter	Physical Errors detected by the PDI.
0x310	Lost Link Counter, Port A (IN)	Link Lost events (per port).
0x311	Lost Link Counter, Port B (OUT)	
0x312	Lost Link Counter, Port C	
0x313	Lost Link Counter, Port D	

Return Value

If the function succeeds, the return value is non-zero

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**

Example

```
if (!acsc_ GetEtherCATSlaveRegister(Handle, ACSC_ETHERCAT_NETWORK_0,
(int)SlaveIndex,(int) Offset, &Value, NULL))
{
   printf("acsc_ GetEtherCATSlaveRegister():Error Occurred -%d\n",
```

```
    acsc_GetLastError());
    return;
}
```

4.38.12 acsc_GetEtherCATSlaveStateV2

Description

The function returns machine state of the specified EtherCAT slave in specific EtherCAT network.

Syntax

```
int acsc_GetEtherCATSlaveStateV2(HANDLE Handle, int Flags ,int
SlaveIndex, double* SlaveState, ACSC_WAITBLOCK* Wait)
```

Arguments

Handle	Communication handle.
Flags	Bit-mapped argument that can include one or more of the following flags: ACSC_ETHERCAT_NETWORK_0 - first network specifier ACSC_ETHERCAT_NETWORK_1 - second network specifier ACSC_ETHERCAT_NETWORK_0 and ACSC_ETHERCAT_NETWORK_1 flags must not be specified together 0, //No flags are set (equivalent to flag ACSC_ETHERCAT_NETWORK_0)
SlaveIndex	index of the required EtherCAT slave, can be determined by acsc_ GetEtherCATSlaveIndex function.
SlaveState	Pointer to a variable that receives the state of the specified EtherCAT slave. 1 - INIT 2 - PREOP 4 - SAFEOP 8 - OP If specified slave does not exist or is not accessible, -1 is returned.

Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero

If the function fails, the return value is zero.

Extended error information can be obtained by calling `acsc_GetLastError`.

Example

```
double Value = 0;
if (!acsc_GetEtherCATSlaveStateV2(Handle, ACSC_ETHERCAT_NETWORK_0,
(int)SlaveIndex, &Value, NULL))
{
   printf("acsc_GetEtherCATSlaveStateV2(): Error Occurred - %d\n",
   acsc_GetLastError());
   return;
}
```

4.38.13 acsc_GetEtherCATSlavesCount

Description

The function is used to retrieve the number of slaves in the specified EtherCAT network.

Syntax

```
int acsc_GetEtherCATSlavesCount(HANDLE Handle, int Flags, double*
SlavesCount, ACSC_WAITBLOCK* Wait)
```

Arguments

Handle	Communication handle.
Flags	Bit-mapped argument that can include one or more of the following flags: ACSC_ETHERCAT_NETWORK_0 - first network specifier ACSC_ETHERCAT_NETWORK_1 - second network specifier ACSC_ETHERCAT_NETWORK_0 and ACSC_ETHERCAT_NETWORK_1 flags must not be specified together

	0, //No flags are set (equivalent to flag ACSC_ETHERCAT_NETWORK_0)
SlavesCount	Pointer to a variable that receives the number of slaves in the specified EtherCAT network If there are no slaves in the system, -1 is returned.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero

If the function fails, the return value is zero.

Extended error information can be obtained by calling `acsc_GetLastError`.

Example

```
double Value = 0;
if (!acsc_GetEtherCATSlavesCount(Handle, ACSC_ETHERCAT_NETWORK_0, &Value,
NULL))
{
   printf("acsc_GetEtherCATSlavesCount(): Error Occurred - %d\n",
   acsc_GetLastError());
   return;
}
```

4.39 Position Event Generation (PEG) Functions

The Position Event Generation functions are:

Table 5-39. Position Event Generation (PEG) Functions

Function	Description
acsc_AssignPegNTV2	Used for engine-to-encoder assignment as well as for the additional digital outputs assignment for use as PEG state and PEG pulse outputs - for SPiiPlusNT and SPiiPlusSC controllers only.
acsc_ AssignPegOutputsNT	Used for setting output pins assignment and mapping between **FGP_OUT** signals to the bits of the ACSPL+ **OUT(x)** variable - for SPiiPlusNT and SPiiPlusSC controllers only.
acsc_AssignFastInputsNT	Used to switch MARK_1 physical inputs to ACSPL+ variables as fast general purpose inputs - for SPiiPlusNT and SPiiPlusSC controllers only.
acsc_PegIncNTV2	Sets the parameters for the Incremental PEG mode - for SPiiPlusNT and SPiiPlusSC controllers only.
acscPegRandomNTV2	Sets the parameters for the Random PEG mode - for SPiiPlusNT and SPiiPlusSC controllers only.
acsc_WaitPegReady	Waits for the all values to be loaded and the PEG engine to be ready to respond to movement on the specified axis - for SPiiPlusNT and SPiiPlusSC controllers only.
acsc_StartPegNT	Used to initiate the PEG process - for SPiiPlusNT and SPiiPlusSC controllers only.
acsc_StopPegNT	Used to terminate the PEG process immediately on the specified axis - for SPiiPlusNT and SPiiPlusSC controllers only.

4.39.1 acsc_AssignPegNTV2

This function can be used only with the SPiiPlus family of controllers.

This function replaces acscPegNT, which is now obsolete.

Description

This 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. As of V3.12, a new flag is available calling for fast loading of Random PEG arrays

Syntax

```
int acsc_AssignPegNTV2(HANDLE Handle, int flags, int Axis, int
EngToEncBitCode, int GpOutsBitCode, ACSC_WAITBLOCK* Wait)
```

Arguments

Handle	Communication handle.
Flags	Bit-mapped argument that can include the following flag: ACSC_AMF_FASTLOADINGPEG If included, fast loading of Random PEG arrays is activated.
Axis	PEG axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions..
EngToEncBitCode	Bit code for engines-to-encoders mapping according to the ASSIGNPEG chapter in the *PEG and MARK Operations Application Notes*.
GpOutsBitCode	General Purpose outputs assignment to use as PEG state and PEG pulse outputs according to the ASSIGNPEG chapter in the *PEG and MARK Operations Application Notes*.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
int Axis = 0;
int EngToEncBitCode = 5;
int GpOutsBitCode = 0;
if (!acsc_AssignPegNTV2(Handle, ACSC_AMF_FASTLOADINGPEG, Axis,
   EngToEncBitCode, GpOutsBitCode,
   NULL))
{
```

```
    printf("acsc_AssignPegNTV2(): Error Occurred - %d\n",
       acsc_GetLastError());
    return;
}
```

4.39.2 acsc_AssignPegOutputsNT

This function can be used only with the SPiiPlus family of controllers.

Description

The function is used for setting output 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 the *SPiiPlus Command & Variable Reference Guide*.

Syntax

int acsc_AssignPegOutputsNT(HANDLE Handle, int Axis, int OutputIndex, int BitCode, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	PEG axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions..
OutputIndex	0 for **OUT_0**, 1 for **OUT_1**, ..., 9 for **OUT_9**.
BitCode	Bit code for engine outputs to physical outputs mapping according to the ASSIGNPEG chapter in the *PEG and MARK Operations Application Notes*.
Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
int Axis = 0;
int OutputIndex = 2;
int OutputBitCode = 1;
if (!acsc_AssignPegOutputsNT(Handle, Axis, OutputIndex, OutputBitCode,
NULL))
{
        printf("acsc_AssignPegOutputsNT(): Error Occurred - %d\n",
        acsc_GetLastError());
        return;
}
```

4.39.3 acsc_AssignFastInputsNT

This function can be used only with the SPiiPlus family of controllers.

Description

The function is used to switch **MARK_1** physical inputs to ACSPL+ variables as fast general purpose inputs.

The function is not related to PEG activity. It is included for the sake of completeness, since many times fast inputs are used in applications that use PEG functionality

The function is used for setting input pins assignment and mapping between **FGP_IN** signals to the bits of the ACSPL+ **IN(x)** variable, where x is the index that has been assigned to the controller in the network during System Configuration.

IN is an integer array that can be used for reading the current state of the General Purpose inputs - see the *SPiiPlus Command & Variable Reference Guide*.

Syntax

int acsc_AssignFastInputsNT(HANDLE Handle, int Axis, int InputIndex, int BitCode,
ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	PEG axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions..
InputIndex	0 for **IN_0**, 1 for **IN_1**, ..., 9 for **IN_9**.
BitCode	Bit code for mapping engines inputs to physical inputs the ASSIGNPEG and ASSIGNPOUTS chapters in the *PEG and MARK Operations Application Notes*.

Wait	Pointer to ACSC_WAITBLOCK structure. If **Wait** is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If **Wait** points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
int Axis = 0;
int InputIndex = 1;
int InputBitCode = 7;
if (!acsc_AssignFastInputsNT(Handle, Axis, InputIndex, InputBitCode,
NULL))
{
        printf("acsc_AssignFastInputsNT(): Error Occurred - %d\n",
        acsc_GetLastError());
        return;
}
```

4.39.4 acsc_PegIncNTV2

This function can be used only with the SPiiPlus family of controllers.

Description

The function is used for setting the parameters for the Incremental PEG mode. The Incremental PEG function is defined by first point, last point and the interval.

Syntax

```
int acsc_PegIncNTV2(HANDLE Handle, int Flags, int Axis, double Width,
double FirstPoint, double Interval, double LastPoint, int ErrMapAxis1,
double AxisCoord1, int ErrMapAxis2, double AxisCoord2, int
ErrMapMaxSize,double MinDirDistance,int TbNumber, double TbPeriod,
ACSC_WAITBLOCK* Wait)
```

Arguments

Handle	Communication handle.

Flags	Bit-mapped parameter that can include following flags: **ACSC_AMF_WAIT** - the execution of the PEG is delayed until the **ACSC_STARTPEGNT** function is executed. **ACSC_AMF_INVERT_OUTPUT** - the PEG pulse output is inverted. **ACSC_AMF_ACCURATE** - error accumulation is prevented by taking into account the rounding of the distance between incremental PEG events. You must use this switch if *interval* does not match a whole number of encoder counts. Using this switch is recommended for any application that uses the **acsc_PegIncNTV2** command, whether or not *interval* defines a whole number of encoder counts. **ACSC_AMF_SYNCHRONOUS** - PEG starts synchronously with the motion sequence. **ACSC_AMF_ENDLESS** This flag supports endless incremental PEG. If the flag is set in in the acsc_PegIncNTV2 function, the last_point parameter is optional and ignored now , and the PEG never stops by position. To stop PEG use the **acsc_StopPegNT**command. **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION1D** This flag supports 1D/2D dynamic error compensation for Incremental PEG. 1D/2D error compensation can be defined by the 1D error mapping functions documented in Dynamic Error Compensation section. **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION2D** This flag supports 2D dynamic error compensation for Incremental PEG. See Dynamic Error Compensation for dynamic error compensation function details. This flag requires 2 additional function arguments: `ErrMapAxis1` and `AxisCoord1`. **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION3D** This flag supports 3D dynamic error compensation for Incremental PEG. See Dynamic Error Compensation for dynamic error compensation function details. This flag must be used in combination with **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION2D** and requires 4 additional function arguments: `ErrMapAxis1`, `AxisCoord1`, `ErrMapAxis2`, and `AxisCoord2`.

	ACSC_AMF_MAXIMUM_ARR_SIZE This flag supports 1D/2D dynamic error compensation for Incremental PEG. The `ErrMapMaxSize` parameter must have a relevant value with this flag is set. **ACSC_AMF_MIN_AXIS_DIRECTION** This flag signals that the `MinDirDistance` parameter indicates a value defining actual motion as opposed to jitter. If motion along the axis when error compensation is in force is greater than this value, that motion is used to determine the direction of motion.
Axis	PEG axis: **ACSC_AXIS_0** corresponds to axis 0, **ACSC_AXIS_1** to axis 1, etc. For the axis constants see Axis Definitions..
Width	Width of desired pulse in milliseconds.
FirstPoint	Position where the first pulse is generated.
Interval	Distance between the pulse-generating points.
LastPoint	Position where the last pulse is generated. If the **ACSC_AMF_ENDLESS** flag is declared, then this parameter should be set to **ACS_NONE**.
ErrMapAxis1	The index of the first static axis when error mapping is used with PEG. Use with the **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION2D** flag alone or in combination with **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION3D** .
AxisCoord1	The predefined location value of the first static axis when error map correction compensation is used with PEG. Use with the **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION2D** flag alone or in combination with **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION3D** .
ErrMapAxis2	The index of the second static axis when error mapping is used with PEG. Use with the **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION2D** and **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION3D** flags.
AxisCoord2	The predefined location value of the second static axis when error map correction compensation is used with PEG. Use with the **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION2D** and **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION3D** flags.
ErrMapMaxSize	Maximum array size contained error correction data. Default is 512. Range from 1 to XARRSIZE. A negative value or 0 is interpreted as default.
MinDirDistance	Limit defining actual motion as opposed to jitter. If motion along the axis is greater than this value, that motion is used to determine the direction of motion.

TbNumber	Number of time-based pulses generated after each encoder-based pulse.
TbPeriod	Period of time-based pulses.
Wait	Pointer to **ACSC_WAITBLOCK** structure. If **Wait** is **ACSC_SYNCHRONOUS**, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is **ACSC_IGNORE**, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Version Support

This function is supported from V3.12 and later.

Example

```
int Flags = 0;
double Width = 0.5;
double FirstPoint = 0;
double LastPoint = 10000;
double Interval = 1000;
if (!acsc_PegIncNT(Handle, Flags, Axis, Width, FirstPoint, Interval,
        LastPoint, ACSC_NONE,
        ACSC_NONE, NULL))
{
        printf("acsc_PegIncNT(): Error Occurred - %d\n", acsc_GetLastError());
        return;
}
int Timeout = 5000;
if (!acsc_WaitPegReadyNT(Handle, Axis, Timeout))
{
        printf("acsc_WaitPegReadyNT(): Error Occurred - %d\n",
        acsc_GetLastError());
        return;
}
```

Example with Error Compensation

```
//PEG engine 0, Pulse width=0.1ms , Start Pos. 500,
// Increment 200, Stop Pos.  1000,
// Use 3D error mapping for axis 1 coordinate = 100,
// for axis 2 coordinate = 200

double Width = 0.01;
double FirstPoint = 500;
double Interval = 200;
double LastPoint = 1000;
int ErrMapMaxSize = 1024;

if (!acsc_PegIncNTV2(Handle, ACSC_AMF_DYNAMIC_ERROR_COMPENSATION2D |
ACSC_AMF_DYNAMIC_ERROR_COMPENSATION3D, 0, Width, FirstPoint, Interval,
LastPoint, 1, 100,  2, 200,2, ACSC_NONE, ACSC_NONE, ACSC_NONE, ACSC_NONE,
ACSC_SYNCHRONOUS))
{
printf("acsc_PegIncNTV2(): Error Occurred - %d\n", acsc_GetLastError());
return;
}
int Timeout = 5000;
if (!acsc_WaitPegReadyNT(Handle, Axis, Timeout))
{
printf("acsc_WaitPegReadyNT(): Error Occurred - %d\n",
acsc_GetLastError());
return;
}
```

4.39.5 acscPegRandomNTV2 NEW

This function replaces acsc_PegRandomNT, which is now obsolete.

The new function includes new parameters supporting error mapping. See Dynamic Error Compensation functions for information on dynamic error mapping.

Description

The function sets the parameters for the Random PEG mode. The Random PEG function specifies an array of points where position-based events are to be generated, and includes parameters for error mapping support.

Syntax

```
int acsc_PegRandomNTV2(HANDLE Handle, int Flags, int Axis, double Width,
int Mode, int FirstIndex, int LastIndex, char* PointArray,
char* StateArray, int ErrMapAxis1, double AxisCoord1, int ErrMapAxis2,
double AxisCoord2, double MinDirDistance, int TbNumber, double TbPeriod,
ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Flags	Bit-mapped parameter that can include following flag: Motion Flags the execution of the PEG is delayed until the acsc_StartPegNT function is executed. ACSC_AMF_INVERT_OUTPUTthe PEG pulse output is inverted. ACSC_AMF_SYNCHRONOUS PEG starts synchronously with the motion sequence. ACSC_AMF_DYNAMICLOADINGGPEG If the ACSC_AMF_DYNAMICLOADINGGPEG flag is included, dynamic loading of positions is implemented. ACSC_AMF_MODULE : A new ACSC_AMF_MODULE flag is introduced to support modulo axis. The Positions Arrays loaded once, provides pulses every Modulo cycle. The Position values should be inside the Modulo range. The PEG engine must be assigned to Modulo axis. **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION1D** This flag supports 1D/2D dynamic error compensation for Incremental PEG. 1D/2D error compensation can be defined by the 1D error mapping functions documented in Dynamic Error Compensation section. **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION2D** This flag supports 2D dynamic error compensation for Incremental PEG. See Dynamic Error Compensation for dynamic error compensation function details. This flag requires 2 additional function arguments: `ErrMapAxis1` and `AxisCoord1`. **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION3D** This flag supports 3D dynamic error compensation for Incremental PEG. See Dynamic Error Compensation for dynamic error compensation function details. This flag must be used in combination with **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION2D** and requires 4 additional function arguments: `ErrMapAxis1`, `AxisCoord1`, `ErrMapAxis2`, and `AxisCoord2`. **ACSC_AMF_MIN_AXIS_DIRECTION** This flag signals that the `MinDirDistance` parameter indicates a value defining actual motion as opposed to jitter. If motion along the axis when error compensation is in force is greater than this value, that motion is used to determine the direction of motion.

Axis	PEG axis: **ACSC_AXIS_0** corresponds to axis 0, **ACSC_AXIS_1** to axis 1, etc. For the axis constants see Axis Definitions..
Width	Width of desired pulse in milliseconds.
Mode	Output signal configuration according to the ASSIGNPEG chapter in the *PEG and MARK Operations Application Notes*..
FirstIndex	Index of position in **PointArray** where the first pulse is generated.
LastIndex	Index of position in **PointArray** where the last pulse is generated.
PointArray	Null-terminated string containing the name of the real array that stores positions at which PEG pulse are to be generated The array must be declared as a global variable by an ACSPL+ program or by the acsc_DeclareVariable function.
StateArray	Null-terminated string containing the name of the integer array that stores desired output state at each position. The array must be declared as a global variable by an ACSPL+ program or by the **acsc_DeclareVariable** function. If output state change is not desired, this parameter should be NULL.
ErrMapAxis1	The index of the first static axis when error mapping is used with PEG. Use with the **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION2D** flag alone or in combination with **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION3D** .
AxisCoord1	The predefined location value of the first static axis when error map correction compensation is used with PEG. Use with the **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION2D** flag alone or in combination with **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION3D** .
ErrMapAxis2	The index of the second static axis when error mapping is used with PEG. Use with the **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION2D** and **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION3D** flags.
AxisCoord2	The predefined location value of the second static axis when error map correction compensation is used with PEG. Use with the **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION2D** and **ACSC_AMF_DYNAMIC_ERROR_COMPENSATION3D** flags.
MinDirDistance	Limit defining actual motion as opposed to jitter. If motion along the axis is greater than this value, that motion is used to determine the direction of motion.
TbNumber	Number of time-based pulses generated after each encoder-based pulse.

TbPeriod	Period of time-based pulses.
Wait	Pointer to **ACSC_WAITBLOCK** structure. If **Wait** is **ACSC_SYNCHRONOUS**, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is **ACSC_IGNORE**, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Comments

The function initiates random PEG generation (see *SPiiPlus ACSPL+ Programmer's Guide*).

StateArray should be NULL if output state won't change because of PEG.

The time-based pulse generation is optional, if it is not used, TbPeriod and TbNumber should be ACSC_NONE.

If Wait points to a valid ACSC_WAITBLOCK structure, the calling thread must not use or delete the Wait item until a call to the acsc_WaitForAsyncCall function.

Supported from V3.12.

Example

```
int Mode = 0;
int FirstIndex = 0;
int LastIndex = 10;
if (!acsc_PegRandomNT(Handle, Flags, Axis, Width, Mode, FirstIndex,
LastIndex, "ARR", "STAT",
ACSC_NONE, ACSC_NONE, NULL))
{
        printf("acsc_PegRandomNT(): Error Occurred - %d\n",
        acsc_GetLastError());
        return;
}
int Timeout = 5000;
if (!acsc_WaitPegReadyNT(Handle, Axis, Timeout))
{
        printf("acsc_WaitPegReadyNT(): Error Occurred - %d\n",
        acsc_GetLastError());
        return;
}
```

Example with Dynamic Error Compensation

```
int Mode = 0;
int FirstIndex = 0;
int LastIndex = 10;

if (!acsc_PegRandomNTV2(Handle, ACSC_AMF_DYNAMIC_ERROR_COMPENSATION2D |
ACSC_AMF_DYNAMIC_ERROR_COMPENSATION3D, Axis, Width, Mode, FirstIndex,
LastIndex, "ARR", "STAT", 1, 100,  2, 200,  ACSC_NONE, ACSC_NONE, ACSC_
NONE, NULL))
{
printf("acsc_PegRandomNTV2(): Error Occurred - %d\n",
acsc_GetLastError());
return;
}
int Timeout = 5000;
if (!acsc_WaitPegReadyNT(Handle, Axis, Timeout))
{
printf("acsc_WaitPegReadyNT(): Error Occurred - %d\n",
acsc_GetLastError());
return;
}
```

4.39.6 acsc_WaitPegReady

This function can be used only with the SPiiPlus family of controllers.

Description

The function waits for the all values to be loaded and the PEG engine to be ready to respond to movement on the specified axis.

The function can be used in both the Incremental and Random PEG modes.

Syntax

int acsc_WaitPegReadyNT(HANDLE Handle, int Axis, int Timeout)

Arguments

Handle	Communication handle.
Axis	PEG axis: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions..
Timeout	Maximum waiting time, in milliseconds. If Timeout is INFINITE, the function's time-out interval never elapses.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
int Timeout = 5000;
if (!acsc_WaitPegReadyNT(Handle, Axis, Timeout))
{
        printf("acsc_WaitPegReadyNT(): Error Occurred - %d\n",
        acsc_GetLastError());
        return;
}
```

4.39.7 acsc_StartPegNT

This function can be used only with the SPiiPlus family of controllers.

Description

The function is used to initiate the PEG process.

Syntax

int acsc_StartPegNT(HANDLE Handle, int Axis, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	PEG axis: **ACSC_AXIS_0** corresponds to axis 0, ACSC_AXIS_1 to axis 1, etc. For the axis constants see Axis Definitions..
Wait	Pointer to **ACSC_WAITBLOCK** structure. If **Wait** is **ACSC_SYNCHRONOUS**, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is **ACSC_IGNORE**, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

```
if (!acsc_StartPegNT(Handle, Axis, NULL))
{
        printf("acsc_StartPegNT(): Error Occurred - %d\n",
        acsc_GetLastError());
        return;
}
```

4.39.8 acsc_StopPegNT

This function can be used only with the SPiiPlus family of controllers.

Description

The function is used to terminate the PEG process immediately on the specified axis. The function can be used in both the Incremental and Random PEG modes.

Syntax

int acsc_StopPegNT(HANDLE Handle, int Axis, ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
Axis	PEG axis: **ACSC_AXIS_0** corresponds to axis 0, **ACSC_AXIS_1** to axis 1, etc. For the axis constants see Axis Definitions..
Wait	Pointer to **ACSC_WAITBLOCK** structure. If **Wait** is **ACSC_SYNCHRONOUS**, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is **ACSC_IGNORE**, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
if (!acsc_StopPegNT(Handle, Axis, NULL))
{
        printf("acsc_StopPegNT(): Error Occurred - %d\n",
        acsc_GetLastError());
        return;
}
```

4.40 Dynamic Error Compensation

4.40.1 acsc_DynamicErrorCompensationOn NEW

Description

acsc_DynamicErrorCompensationOn function receives axis index and zone index parameters. The function activates error correction for the mechanical error compensation for the specified zone.

Syntax

```
int acsc_DynamicErrorCompensationOn(HANDLE Handle, int Axis, int Zone,
ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc. For the axis constants see Axis Definitions.
Zone	The zone index, valid numbers are: 0, 1, 2, ... up to the maximum number of zones (up to 10) minus 1. If '-1' is specified, all zones of specified axis will be affected.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero

If the function fails, the return value is zero.

Extended error information can be obtained by calling acsc_GetLastError

Example

```
if (!acsc_DynamicErrorCompensationOn(hComm, ACSC_AXIS_0, 0, ACSC_
SYNCHRONOUS))
{
   printf("acsc_DynamicErrorCompensationOn():Error Occurred -%d\n",
   acsc_GetLastError());
```

```
   return;
}
```

4.40.2 acsc_DynamicErrorCompensationOff

Description

acsc_DynamicErrorCompensationOff function receives axis index and zone index parameters . The function deactivates error mapping correction for the mechanical error compensation for the specified zone.

Syntax

```
int acsc_DynamicErrorCompensationOff(HANDLE Handle, int Axis, int Zone,
ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc. For the axis constants see Axis Definitions.
Zone	The zone index, valid numbers are: 0, 1, 2, ... up to the maximum number of zones (10) minus 1. If '-1' is specified, all zones of specified axis will be affected.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero

If the function fails, the return value is zero.

Extended error information can be obtained by calling acsc_GetLastError

Example

```
if (!acsc_DynamicErrorCompensationOff(hComm, ACSC_AXIS_0, 0, ACSC_
SYNCHRONOUS))
{
   printf("acsc_DynamicErrorCompensationOff():Error Occurred -%d\n",
   acsc_GetLastError());
```

```
    return;
}
```

4.40.3 acsc_DynamicErrorCompensation1D

Description

The acsc_DynamicErrorCompensation1D function configures and activates 1D error correction for the mechanical error compensation for the specified zone, so that the compensated reference position will be calculated by subtracting the linearly (by default) interpolated error from the desired position so that the actual value will be closer to the desired value. The calculation assumes fixed intervals between points inside the zone.

Syntax

```
acsc_DynamicErrorCompensation1D(HANDLE Handle, int Flags, int Axis, int
Zone, double Base0, double Step0, char* CorrectionMapVariable, int
ReferencedAxisOrAnalogInput, ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc. For the axis constants see Axis Definitions.
Flags	ACSC_DECOMP_PREVENT_COMP_INDEX_MARK_PEG Prevent applying dynamic error compensation on INDEX, MARK, and PEG values ACSC_DECOMP_REFERENCED_AXIS Specifies that the mechanical error compensation will be calculated based on the feedback from the axis specified by the optional parameter. ACSC_DECOMP_ANALOG_INPUT Specifies that the mechanical error compensation will be calculated based on the feedback from the analog input indicated by the optional parameter. 0, //No flags are set
Zone	The zone index, valid numbers are: 0, 1, 2, ... up to the maximum number of zones (10) minus 1. If '-1' is specified, all zones of specified axis will be affected.
Base0	A real number representing the axis command that corresponds to the first point in correction table for mechanical error compensation.
Step0	A real number representing the fixed interval distance between the two adjacent axis commands.

correction_map	The name of a real one-dimensional array that specifies correction table for mechanical error compensation. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
referenced_ axis_or_analog_ input	The index of the axis, or the index of the analog input that the mechanical error compensation will be calculated based on its feedback.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**

Example

```
char Correction[11]= "Correction";
if (!acsc_DynamicErrorCompensation1D(hComm,0,ACSC_AXIS_0, 0,0,10,
Correction, ACSC_NONE, ACSC_SYNCHRONOUS)){
printf("acsc_DynamicErrorCompensation1D():Error Occurred -%d\n",
   acsc_GetLastError());
   return;
}
```

4.40.4 acsc_DynamicErrorCompensationN1D

Description

The acsc_DynamicErrorCompensationN1D function configures and activates 1D error correction for the mechanical error compensation for the specified zone, so that the compensated reference position will be calculated by subtracting the linearly (by default) interpolated error from the desired position so that the actual value will be closer to the desired value. The calculation is based on an arbitrary network of points inside the zone.

Syntax

```
acsc_DynamicErrorCompensationN1D(HANDLE Handle, int Flags, int Axis, int
Zone, char* AxisCommands, char* CorrectionMapVariable, int
```

```
ReferencedAxisOrAnalogInput, ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc. For the axis constants see Axis Definitions.
Flags	ACSC_DECOMP_PREVENT_COMP_INDEX_MARK_PEG Prevent applying dynamic error compensation on INDEX, MARK, and PEG values ACSC_DECOMP_REFERENCED_AXIS Specifies that the mechanical error compensation will be calculated based on the feedback from the axis specified by the optional parameter. ACSC_DECOMP_ANALOG_INPUT Specifies that the mechanical error compensation will be calculated based on the feedback from the analog input indicated by the optional parameter. 0, //No flags are set
Zone	The zone index, valid numbers are: 0, 1, 2, ... up to the maximum number of zones (10) minus 1. If '-1' is specified, all zones of specified axis will be affected.
axis_command	The name of a real one-dimensional array that specifies axis command values used for correction table of mechanical error compensation. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map	The name of a real one-dimensional array that specifies correction table for mechanical error compensation. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
referenced_ axis_or_analog_ input	[Optional] The index of the axis, or the index of the analog input that the mechanical error compensation will be calculated based on its feedback.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Example

```
char Correction[11]= "Correction";
char AxisCoordinates[17] = "Axis_Coordinates";
if (!acsc_DynamicErrorCompensationN1D(hComm,NULL,ACSC_AXIS_0, 0,
AxisCoordinates, Correction, ACSC_NONE, ACSC_SYNCHRONOUS)){
printf("acsc_DynamicErrorCompensationN1D():Error Occurred -%d\n",
   acsc_GetLastError());
   return;
}
```

4.40.5 acsc_DynamicErrorCompensationA1D

Description

The acsc_DynamicErrorCompensationA1D function configures and activates 1D error correction for the mechanical error compensation for the specified zone, so that the compensated reference position will be calculated by multiplying the scaling factor by the desired position so that the actual value will be closer to the desired value.

Syntax

```
acsc_DynamicErrorCompensationA1D(HANDLE Handle, int Flags, int Axis, int
Zone, double scaling_factor, double offset, ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc. For the axis constants see Axis Definitions.
Flags	ACSC_DECOMP_PREVENT_COMP_INDEX_MARK_PEG Prevent applying dynamic error compensation on INDEX, MARK, and PEG values 0, //No flags are set
Zone	The zone index, valid numbers are: 0, 1, 2, ... up to the maximum number of zones (10) minus 1. If '-1' is specified, all zones of specified axis will be affected.
scaling_ factor	The scaling factor for the linear alignment that will be used for mechanical error compensation. The allowed range for the scaling factor is 0>2.

offset	The offset for the linear alignment that will be used for mechanical error compensation. The offset is actually the mechanical error compensation for the 0-point location.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Example

```
if (!acsc_DynamicErrorCompensationA1D(hComm,NULL,ACSC_AXIS_0, 0,
 1.2, ACSC_NONE, ACSC_SYNCHRONOUS)){
printf("acsc_DynamicErrorCompensationA1D():Error Occurred -%d\n",
   acsc_GetLastError());
   return;
}
```

4.40.6 acsc_DynamicErrorCompensation2D

Description

The acsc_DynamicErrorCompensation2D function configures and activates 2D error correction for the mechanical error compensation of the 'axis0' command for the specified zone, so that the compensated reference position will be calculated by subtracting the linearly (by default) interpolated error from the desired position so that the actual value will be closer to the desired value.

Syntax

```
acsc_DynamicErrorCompensation2D(HANDLE Handle, int Flags, int Axis0, int
Axis1,int Zone, double Base0, double Step0, double Base1, double Step1,
char* CorrectionMapVariable, int ReferencedAxisOrAnalogInput0, int
ReferencedAxisOrAnalogInput1,ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Axis0	The index of the first axis that the mechanical error compensation will be applied to. Valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
Axis1	The index of the second axis participating in 2D mechanical error compensation. Valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1
Flags	ACSC_DECOMP_00 The mechanical error compensation will be applied to 'axis0' (default) ACSC_DECOMP_01 The mechanical error compensation will be applied to 'axis1' ACSC_DECOMP_REFERENCED_AXIS Specifies that the mechanical error compensation will be calculated based on the feedback from the axis specified by the optional parameter. ACSC_DECOMP_ANALOG_INPUT Specifies that the mechanical error compensation will be calculated based on the feedback from the analog input indicated by the optional parameter. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_FIRST_ANALOG_ INPUT Specifies that the first optional parameter will be regarded as an analog input index. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_SECOND_ANALOG_ INPUT Specifies that the second optional parameter will be regarded as an analog input index. 0, //No flags are set
Zone	The zone index, valid numbers are: 0, 1, 2, ... up to the maximum number of zones (10) minus 1. If '-1' is specified, all zones of specified axis will be affected.
Base0	A real number representing the 'axis0' command that corresponds to the first point in correction table for mechanical error compensation.
Step0	A real number representing the fixed interval distance between the two adjacent 'axis0' commands.
Base1	A real number representing the 'axis1' command that corresponds to the first point in correction table for mechanical error compensation.

Step1	A real number representing the fixed interval distance between the two adjacent 'axis1' commands.
correction_map	The name of a real two-dimensional array that specifies correction table for mechanical error compensation. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
referenced_ axis_or_analog_ input0	The index of the first axis, or the index of the first analog input whose feedback will be used to calculate the mechanical error compensation.
referenced_ axis_or_analog_ input1	The index of the second axis, or the index of the second analog input whose feedback will be used to calculate the mechanical error compensation.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Example

```
char Correction[11]= "Correction";
if (!acsc_DynamicErrorCompensation2D(hComm,0,ACSC_AXIS_0, ACSC_AXIS_1, 0,
0,10,3,5, Correction, ACSC_NONE, ACSC_NONE, ACSC_SYNCHRONOUS)){
printf("acsc_DynamicErrorCompensation2D():Error Occurred -%d\n",
   acsc_GetLastError());
   return;
}
```

4.40.7 acsc_DynamicErrorCompensationN2D

Description

The acsc_DynamicErrorCompensationN2D function configures and activates 2D error correction for the mechanical error compensation of the 'axis0' parameter for the specified zone, so that the compensated reference position will be calculated by subtracting the linearly (by default)

interpolated error from the desired position so that the actual value will be closer to the desired value. The calculation is based on an arbitrary network of points inside the zone.

Syntax

```
acsc_DynamicErrorCompensationN1D(HANDLE Handle, int Flags, int Axis0, int
Axis1,int Zone, char* Axis0Commands, char* Axis1Commands,  char*
CorrectionMapVariable,int ReferencedAxisOrAnalogInput0, int
ReferencedAxisOrAnalogInput1, ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Axis0	The index of the first axis that the mechanical error compensation will be applied to. Valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
Axis1	The index of the second axis participating in 2D mechanical error compensation. Valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1
Flags	ACSC_DECOMP_00 The mechanical error compensation will be applied to 'axis0' (default) ACSC_DECOMP_01 The mechanical error compensation will be applied to 'axis1' ACSC_DECOMP_REFERENCED_AXIS Specifies that the mechanical error compensation will be calculated based on the feedback from the axis specified by the optional parameter. ACSC_DECOMP_ANALOG_INPUT Specifies that the mechanical error compensation will be calculated based on the feedback from the analog input indicated by the optional parameter. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_FIRST_ANALOG_ INPUT Specifies that the first optional parameter will be regarded as an analog input index. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_SECOND_ANALOG_ INPUT Specifies that the second optional parameter will be regarded as an analog input index. 0, //No flags are set
Zone	The zone index, valid numbers are: 0, 1, 2, ... up to the maximum number of zones (10) minus 1. If '-1' is specified, all zones of specified axis will be affected.
Axis0_command	The name of a real one-dimensional array that specifies 'axis0' command

	values used for correction table of mechanical error compensation. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
Axis1_command	The name of a real one-dimensional array that specifies 'axis1' command values used for correction table of mechanical error compensation. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map	The name of a real two-dimensional array that specifies correction table for mechanical error compensation. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
referenced_ axis_or_analog_ input0	The index of the first axis, or the index of the first analog input whose feedback will be used to calculate the mechanical error compensation.
referenced_ axis_or_analog_ input1	The index of the second axis, or the index of the second analog input whose feedback will be used to calculate the mechanical error compensation.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Example

```
char Correction[11]= "Correction";
char Axis0Coordinates[18] = "Axis0_Coordinates";
char Axis1Coordinates[18] = "Axis0_Coordinates";
if (!acsc_DynamicErrorCompensationN2D(hComm,NULL,ACSC_AXIS_0, ACSC_AXIS_
1, 0, Axis0Coordinates, Axis1Coordinates, Correction, ACSC_NONE, ACSC_
NONE, ACSC_SYNCHRONOUS)){
printf("acsc_DynamicErrorCompensationN2D():Error Occurred -%d\n",
   acsc_GetLastError());
```

```
    return;
}
```

4.40.8 acsc_DynamicErrorCompensationA2D

Description

The acsc_DynamicErrorCompensationA2D function configures and activates 2D error correction for the mechanical error compensation of the specified axis for the specified zone, so that the compensated reference position will be calculated by taking into account the angle for the orthogonality correction so that the actual value will be closer to the desired value.

Syntax

```
acsc_DynamicErrorCompensationA2D(HANDLE Handle, int Flags, int Axis0, int
Axis1, int Zone, double Angle, ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Axis0	The index of the first axis that the mechanical error compensation will be applied to. Valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
Axis1	The index of the second axis participating in 2D mechanical error compensation. Valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1
Flags	ACSC_DECOMP_00 The mechanical error compensation will be applied to 'axis0' (default) ACSC_DECOMP_01 The mechanical error compensation will be applied to 'axis1' 0, //No flags are set
Zone	The zone index, valid numbers are: 0, 1, 2, ... up to the maximum number of zones (10) minus 1. If '-1' is specified, all zones of specified axis will be affected.
Angle	The angle for the orthogonality correction that will be used for mechanical error compensation. The allowed range for the angle is [-45°, 45°].

Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Example

```
if (!acsc_DynamicErrorCompensationA2D(hComm,NULL,ACSC_AXIS_0, ACSC_AXIS_
1, 0,  30, ACSC_SYNCHRONOUS)){
printf("acsc_DynamicErrorCompensationA2D():Error Occurred -%d\n",
   acsc_GetLastError());
   return;
}
```

4.40.9 acsc_DynamicErrorCompensation3D2

Description

The acsc_DynamicErrorCompensation3D2 function configures and activates 3D error correction for the mechanical error compensation of the 'axis0', 'axis1', and 'axis2' parameters for the specified zone, so that the compensated reference position will be calculated by adding the linearly (by default) interpolated error from the desired position so that the actual value will be closer to the desired value.

Syntax

```
acsc_DynamicErrorCompensation3D2(HANDLE Handle, int Flags, int Axis0, int
Axis1,int Axis2,int Zone, double Base0, double Step0, double Base1,
double Step1, double Base2, double Step2, char* CorrectionMap0Variable,
char* CorrectionMap1Variable, int ReferencedAxisOrAnalogInput0,int
ReferencedAxisOrAnalogInput1, int ReferencedAxisOrAnalogInput2,
ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Axis0	The index of the first axis that the mechanical error compensation will be

	applied to. Valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
axis1	The index of the second axis participating in 3D mechanical error compensation, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
axis2	The index of the second axis participating in 3D mechanical error compensation, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
Flags	ACSC_DECOMP_00 The mechanical error compensation will be applied to 'axis0' (default) ACSC_DECOMP_01 The mechanical error compensation will be applied to 'axis1' ACSC_DECOMP_02 The mechanical error compensation will be applied to 'axis1' ACSC_DECOMP_REFERENCED_AXIS Specifies that the mechanical error compensation will be calculated based on the feedback from the axis specified by the optional parameter. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_FIRST_ANALOG_INPUT Specifies that the first optional parameter will be regarded as an analog input index. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_SECOND_ANALOG_INPUT Specifies that the second optional parameter will be regarded as an analog input index. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_THIRD_ANALOG_INPUT Specifies that the third optional parameter will be regarded as an analog input index. 0, //No flags are set
Zone	The zone index, valid numbers are: 0, 1, 2, ... up to the maximum number of zones (10) minus 1. If '-1' is specified, all zones of specified axis will be affected.
Base0	A real number representing the 'axis0' command that corresponds to the first point in correction table for mechanical error compensation.
Step0	A real number representing the fixed interval distance between the two adjacent 'axis0' commands.
Base1	A real number representing the 'axis1' command that corresponds to the first point in correction table for mechanical error compensation.

Step1	A real number representing the fixed interval distance between the two adjacent 'axis1' commands.
Base2	A real number representing the 'axis2' command that corresponds to the first point in correction table for mechanical error compensation.
Step2	A real number representing the fixed interval distance between the two adjacent 'axis2' commands.
correction_ map0	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '0' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map1	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '1' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
referenced_ axis_or_analog_ input0	The index of the first axis, or the index of the first analog input whose feedback will be used to calculate the mechanical error compensation.
referenced_ axis_or_analog_ input1	The index of the second axis, or the index of the second analog input whose feedback will be used to calculate the mechanical error compensation.
referenced_ axis_or_analog_ input2	The index of the third axis, or the index of the third analog input whose feedback will be used to calculate the mechanical error compensation.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Example

```
char Correction1[11]= "Correction";
char Correction2[12]= "Correction2";
if (!acsc_DynamicErrorCompensation3D2(hComm,
0, // flags
ACSC_AXIS_0, //axis0
ACSC_AXIS_1, // axis1
ACSC_AXIS_2, // axis2
0, //Zone
0, //Base0
10, //Step0
5,//Base1
15,//Step1
10,// Base2
20, //Step2
Correction1, //Correction map 0
Correction2, //Correction map 1
ACSC_NONE, // referenced_axis_or_analog_input0
ACSC_NONE, // referenced_axis_or_analog_input1
ACSC_NONE, // referenced_axis_or_analog_input2
ACSC_SYNCHRONOUS))
{
printf("acsc_DynamicErrorCompensation3D2():Error Occurred -%d\n",acsc_
GetLastError());
   return;
}
```

4.40.10 acsc_DynamicErrorCompensation3D3 NEW

Description

The acsc_DynamicErrorCompensation3D3 function configures and activates 3D error correction for the mechanical error compensation of the 'axis0', 'axis1', and 'axis2' parameters for the specified zone, so that the compensated reference position will be calculated by adding the linearly (by default) interpolated error from the desired position so that the actual value will be closer to the desired value.

Syntax

```
acsc_DynamicErrorCompensation3D3(HANDLE Handle, int Flags, int Axis0, int
Axis1,int Axis2,int Zone, double Base0, double Step0, double Base1,
double Step1, double Base2, double Step2, char* CorrectionMap0Variable,
char* CorrectionMap1Variable, char* CorrectionMap2Variable,int
ReferencedAxisOrAnalogInput0, int ReferencedAxisOrAnalogInput1,
int ReferencedAxisOrAnalogInput2, ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Axis0	The index of the first axis that the mechanical error compensation will be applied to. Valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
axis1	The index of the second axis participating in 3D mechanical error compensation, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
axis2	The index of the second axis participating in 3D mechanical error compensation, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
Flags	ACSC_DECOMP_00 The mechanical error compensation will be applied to 'axis0' (default) ACSC_DECOMP_01 The mechanical error compensation will be applied to 'axis1' ACSC_DECOMP_02 The mechanical error compensation will be applied to 'axis1' ACSC_DECOMP_REFERENCED_AXIS Specifies that the mechanical error compensation will be calculated based on the feedback from the axis specified by the optional parameter. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_FIRST_ANALOG_INPUT Specifies that the first optional parameter will be regarded as an analog input index. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_SECOND_ANALOG_INPUT Specifies that the second optional parameter will be regarded as an analog input index. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_THIRD_ANALOG_INPUT Specifies that the third optional parameter will be regarded as an analog input index. 0, //No flags are set
Zone	The zone index, valid numbers are: 0, 1, 2, ... up to the maximum number of zones (10) minus 1. If '-1' is specified, all zones of specified axis will be affected.
Base0	A real number representing the 'axis0' command that corresponds to the first point in correction table for mechanical error compensation.
Step0	A real number representing the fixed interval distance between the two adjacent 'axis0' commands.

Base1	A real number representing the 'axis1' command that corresponds to the first point in correction table for mechanical error compensation.
Step1	A real number representing the fixed interval distance between the two adjacent 'axis1' commands.
Base2	A real number representing the 'axis2' command that corresponds to the first point in correction table for mechanical error compensation.
Step2	A real number representing the fixed interval distance between the two adjacent 'axis2' commands.
correction_ map0	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '0' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map1	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '1' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map2	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '2' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
referenced_ axis_or_analog_ input0	The index of the first axis, or the index of the first analog input whose feedback will be used to calculate the mechanical error compensation.
referenced_ axis_or_analog_ input1	The index of the second axis, or the index of the second analog input whose feedback will be used to calculate the mechanical error compensation.
referenced_ axis_or_analog_ input2	The index of the second axis, or the index of the third analog input whose feedback will be used to calculate the mechanical error compensation.

Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Example

```
char Correction0[12]= "Correction0";
char Correction1[12]= "Correction1";
char Correction2[12]= "Correction2";
if (!acsc_DynamicErrorCompensation3D3(hComm,
0, // flags
ACSC_AXIS_0, //axis0
ACSC_AXIS_1, // axis1
ACSC_AXIS_2, // axis2
0, //Zone
0, //Base0
10, //Step0
5,//Base1
15,//Step1
10,// Base2
20, //Step2
Correction0, //Correction map 0
Correction1, //Correction map 1
Correction2, //Correction map 2
ACSC_NONE, // referenced_axis_or_analog_input0
ACSC_NONE, // referenced_axis_or_analog_input1
ACSC_NONE, // referenced_axis_or_analog_input2
ACSC_SYNCHRONOUS))
{
printf("acsc_DynamicErrorCompensation3D3():Error Occurred -%d\n",acsc_
GetLastError());
   return;
}
```

4.40.11 acsc_DynamicErrorCompensation3D5

Description

The acsc_DynamicErrorCompensation3D5 function configures and activates 3D error correction for the mechanical error compensation of the 'axis0', 'axis1', and 'axis2' parameters for the specified zone, so that the compensated reference position will be calculated by adding the linearly (by default) interpolated error from the desired position so that the actual value will be closer to the desired value.

Syntax

```
acsc_DynamicErrorCompensation3D5(HANDLE Handle, int Flags, int Axis0, int
Axis1,int Axis2,int Zone, double Base0, double Step0, double Base1,
double Step1, double Base2, double Step2, char* CorrectionMap0Variable,
char* CorrectionMap1Variable, char* CorrectionMap2Variable,
char* CorrectionMap3Variable, char* CorrectionMap4Variable,
int ReferencedAxisOrAnalogInput0, int ReferencedAxisOrAnalogInput1,
int ReferencedAxisOrAnalogInput2, ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Axis0	The index of the first axis that the mechanical error compensation will be applied to. Valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
axis1	The index of the second axis participating in 3D mechanical error compensation, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
axis2	The index of the second axis participating in 3D mechanical error compensation, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.

Flags	ACSC_DECOMP_00 The mechanical error compensation will be applied to 'axis0' (default) ACSC_DECOMP_01 The mechanical error compensation will be applied to 'axis1' ACSC_DECOMP_02 The mechanical error compensation will be applied to 'axis1' ACSC_DECOMP_REFERENCED_AXIS Specifies that the mechanical error compensation will be calculated based on the feedback from the axis specified by the optional parameter. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_FIRST_ANALOG_INPUT Specifies that the first optional parameter will be regarded as an analog input index. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_SECOND_ANALOG_INPUT Specifies that the second optional parameter will be regarded as an analog input index. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_THIRD_ANALOG_INPUT Specifies that the third optional parameter will be regarded as an analog input index. 0, //No flags are set
Zone	The zone index, valid numbers are: 0, 1, 2, ... up to the maximum number of zones (10) minus 1. If '-1' is specified, all zones of specified axis will be affected.
Base0	A real number representing the 'axis0' command that corresponds to the first point in correction table for mechanical error compensation.
Step0	A real number representing the fixed interval distance between the two adjacent 'axis0' commands.
Base1	A real number representing the 'axis1' command that corresponds to the first point in correction table for mechanical error compensation.
Step1	A real number representing the fixed interval distance between the two adjacent 'axis1' commands.
Base2	A real number representing the 'axis2' command that corresponds to the first point in correction table for mechanical error compensation.
Step2	A real number representing the fixed interval distance between the two adjacent 'axis2' commands.

correction_ map0	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '0' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map1	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '1' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map2	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '2' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map3	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '3' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_ map4	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '4' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
referenced_ axis_or_analog_ input0	The index of the first axis, or the index of the first analog input whose feedback will be used to calculate the mechanical error compensation.
referenced_ axis_or_analog_ input1	The index of the second axis, or the index of the second analog input whose feedback will be used to calculate the mechanical error compensation.
referenced_ axis_or_analog_ input2	The index of the second axis, or the index of the third analog input whose feedback will be used to calculate the mechanical error compensation.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Example

```
char Correction0[12]= "Correction0";
char Correction1[12]= "Correction1";
char Correction2[12]= "Correction2";
char Correction3[12]= "Correction3";
char Correction4[12]= "Correction4";
if (!acsc_DynamicErrorCompensation3D5(hComm,
0, // flags
ACSC_AXIS_0, //axis0
ACSC_AXIS_1, // axis1
ACSC_AXIS_2, // axis2
0, //Zone
0, //Base0
10, //Step0
5,//Base1
15,//Step1
10,// Base2
20, //Step2
Correction0, //Correction map 0
Correction1, //Correction map 1
Correction2, //Correction map 2
Correction3, //Correction map 3
Correction4, //Correction map 4
ACSC_NONE, // referenced_axis_or_analog_input0
ACSC_NONE, // referenced_axis_or_analog_input1
ACSC_NONE, // referenced_axis_or_analog_input2
ACSC_SYNCHRONOUS))
{
printf("acsc_DynamicErrorCompensation3D5():Error Occurred -%d\n",acsc_
GetLastError());
   return;
}
```

4.40.12 acsc_DynamicErrorCompensation3DA

Description

The acsc_DynamicErrorCompensation3DA function configures and activates 3D error correction for the mechanical error compensation of the ‘axis0’, ‘axis1’, and 'axis2' parameters for the specified zone, so that the compensated reference position will be calculated by adding the linearly (by default) interpolated error from the desired position so that the actual value will be closer to the desired value.

Syntax

```
acsc_DynamicErrorCompensation3DA(HANDLE Handle, int Flags, int Axis0, int
Axis1,
int Axis2, int Zone, double Base0, double Step0, double Base1, double
Step1,
double Base2, double Step2, char* CorrectionMap0Variable,
char* CorrectionMap1Variable, char* CorrectionMap2Variable,
char* CorrectionMap3Variable, char* CorrectionMap4Variable,
char* CorrectionMap5Variable, char* CorrectionMap6Variable,
char* CorrectionMap7Variable, char* CorrectionMap8Variable,
char* CorrectionMap9Variable,
int ReferencedAxisOrAnalogInput0, int ReferencedAxisOrAnalogInput1,
int ReferencedAxisOrAnalogInput2, ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Axis0	The index of the first axis that the mechanical error compensation will be applied to. Valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
axis1	The index of the second axis participating in 3D mechanical error compensation, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
axis2	The index of the second axis participating in 3D mechanical error compensation, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.

Flags	ACSC_DECOMP_00 The mechanical error compensation will be applied to 'axis0' (default) ACSC_DECOMP_01 The mechanical error compensation will be applied to 'axis1' ACSC_DECOMP_02 The mechanical error compensation will be applied to 'axis1' ACSC_DECOMP_REFERENCED_AXIS Specifies that the mechanical error compensation will be calculated based on the feedback from the axis specified by the optional parameter. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_FIRST_ANALOG_INPUT Specifies that the first optional parameter will be regarded as an analog input index. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_SECOND_ANALOG_INPUT Specifies that the second optional parameter will be regarded as an analog input index. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_THIRD_ANALOG_INPUT Specifies that the third optional parameter will be regarded as an analog input index. 0, //No flags are set
Zone	The zone index, valid numbers are: 0, 1, 2, ... up to the maximum number of zones (10) minus 1. If '-1' is specified, all zones of specified axis will be affected.
Base0	A real number representing the 'axis0' command that corresponds to the first point in correction table for mechanical error compensation.
Step0	A real number representing the fixed interval distance between the two adjacent 'axis0' commands.
Base1	A real number representing the 'axis1' command that corresponds to the first point in correction table for mechanical error compensation.
Step1	A real number representing the fixed interval distance between the two adjacent 'axis1' commands.
Base2	A real number representing the 'axis2' command that corresponds to the first point in correction table for mechanical error compensation.
Step2	A real number representing the fixed interval distance between the two adjacent 'axis2' commands.

correction_ map0	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '0' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map1	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '1' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map2	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '2' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map3	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '3' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_ map4	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '4' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_ map5	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '5' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_ map6	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '6' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map7	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '7' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_ map8	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '8' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_ map9	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '9' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).

referenced_ axis_or_analog_ input0	The index of the first axis, or the index of the first analog input whose feedback will be used to calculate the mechanical error compensation.
referenced_ axis_or_analog_ input1	The index of the second axis, or the index of the second analog input whose feedback will be used to calculate the mechanical error compensation.
referenced_ axis_or_analog_ input2	The index of the second axis, or the index of the third analog input whose feedback will be used to calculate the mechanical error compensation.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Example

```
char Correction0[12]= "Correction0";
char Correction1[12]= "Correction1";
char Correction2[12]= "Correction2";
char Correction3[12]= "Correction3";
char Correction4[12]= "Correction4";
char Correction5[12]= "Correction5";
char Correction6[12]= "Correction6";
char Correction7[12]= "Correction7";
char Correction8[12]= "Correction8";
char Correction9[12]= "Correction9";
if (!acsc_DynamicErrorCompensation3DA(hComm,
0, // flags
ACSC_AXIS_0, //axis0
ACSC_AXIS_1, // axis1
ACSC_AXIS_2, // axis2
0, //Zone
```

```
0, //Base0
10, //Step0
5,//Base1
15,//Step1
10,// Base2
20, //Step2
Correction0, //Correction map 0
Correction1, //Correction map 1
Correction2, //Correction map 2
Correction3, //Correction map 3
Correction4, //Correction map 4
Correction5, //Correction map 5
Correction6, //Correction map 6
Correction7, //Correction map 7
Correction8, //Correction map 8
Correction9, //Correction map 9
ACSC_NONE, // referenced_axis_or_analog_input0
ACSC_NONE, // referenced_axis_or_analog_input1
ACSC_NONE, // referenced_axis_or_analog_input2

ACSC_SYNCHRONOUS))
{
printf("acsc_DynamicErrorCompensation3DA():Error Occurred -%d\n",acsc_
GetLastError());
   return;
}
```

4.40.13 acsc_DynamicErrorCompensationN3D2

Description

The acsc_DynamicErrorCompensationN3D2 function configures and activates 3D error correction for the mechanical error compensation of the 'axis0', 'axis1', and 'axis2' parameters for the specified zone, so that the compensated reference position will be calculated by adding the linearly (by default) interpolated error from the desired position so that the actual value will be closer to the desired value.

Syntax

```
acsc_DynamicErrorCompensationN3D2(HANDLE Handle, int Flags, int Axis0,
int Axis1,
int Axis2,int Zone, char* Axis0Commands, char* Axis1Commands, char*
Axis2Commands,
char* CorrectionMap0Variable,char* CorrectionMap1Variable,
int ReferencedAxisOrAnalogInput0, int ReferencedAxisOrAnalogInput1,
int ReferencedAxisOrAnalogInput2, ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Axis0	The index of the first axis that the mechanical error compensation will be applied to. Valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
axis1	The index of the second axis participating in 3D mechanical error compensation, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
axis2	The index of the second axis participating in 3D mechanical error compensation, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
Flags	ACSC_DECOMP_00 The mechanical error compensation will be applied to 'axis0' (default) ACSC_DECOMP_01 The mechanical error compensation will be applied to 'axis1' ACSC_DECOMP_02 The mechanical error compensation will be applied to 'axis1' ACSC_DECOMP_REFERENCED_AXIS Specifies that the mechanical error compensation will be calculated based on the feedback from the axis specified by the optional parameter. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_FIRST_ANALOG_ INPUT Specifies that the first optional parameter will be regarded as an analog input index. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_SECOND_ANALOG_ INPUT Specifies that the second optional parameter will be regarded as an analog input index. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_THIRD_ANALOG_ INPUT Specifies that the third optional parameter will be regarded as an analog input index. 0, //No flags are set
Zone	The zone index, valid numbers are: 0, 1, 2, ... up to the maximum number of zones (10) minus 1. If '-1' is specified, all zones of specified axis will be affected.

axis0_command	The name of a real one-dimensional array that specifies 'axis0' command values used for correction table of mechanical error compensation. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
axis1_command	The name of a real one-dimensional array that specifies 'axis1' command values used for correction table of mechanical error compensation. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
axis2_command	The name of a real one-dimensional array that specifies 'axis2' command values used for correction table of mechanical error compensation. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_ map0	The name of a real two-dimensional array that specifies a correction table for mechanical error compensation in relation to the first specified coordinate of the Z-axis. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map1	The name of a real two-dimensional array that specifies a correction table for mechanical error compensation in relation to the second specified coordinate of the Z-axis. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
referenced_ axis_or_analog_ input0	The index of the first axis, or the index of the first analog input whose feedback will be used to calculate the mechanical error compensation.
referenced_ axis_or_analog_ input1	The index of the second axis, or the index of the second analog input whose feedback will be used to calculate the mechanical error compensation.
referenced_ axis_or_analog_ input2	The index of the second axis, or the index of the third analog input whose feedback will be used to calculate the mechanical error compensation.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Example

```
char Correction0[12]= "Correction0";
char Correction1[12]= "Correction1";
char Axis0Coordinates[17] = "Axis0Coordinates";
char Axis1Coordinates[17] = "Axis1Coordinates";
char Axis2Coordinates[17] = "Axis2Coordinates";
if (!acsc_DynamicErrorCompensationN3D2(hComm,
0, // flags
ACSC_AXIS_0, //axis0
ACSC_AXIS_1, // axis1
ACSC_AXIS_2, // axis2
0, //Zone
Axis0Coordinates, // The name of a real one-dimensional array that
specifies 'axis0'
Axis1Coordinates, // The name of a real one-dimensional array that
specifies 'axis1'
Axis2Coordinates,// The name of a real one-dimensional array that
specifies 'axis0'
Correction0, //Correction map 0
Correction1, //Correction map 1
ACSC_NONE, // referenced_axis_or_analog_input0
ACSC_NONE, // referenced_axis_or_analog_input1
ACSC_NONE, // referenced_axis_or_analog_input2
ACSC_SYNCHRONOUS))
{
printf("acsc_DynamicErrorCompensationN3D2():Error Occurred -%d\n",acsc_
GetLastError());
   return;
}
```

4.40.14 acsc_DynamicErrorCompensationN3D3

Description

The acsc_DynamicErrorCompensationN3D3 function configures and activates 3D error correction for the mechanical error compensation of the 'axis0', 'axis1', and 'axis2' parameters for the specified zone, so that the compensated reference position will be calculated by adding the linearly (by default) interpolated error from the desired position so that the actual value will be closer to the desired value.

Syntax

```
acsc_DynamicErrorCompensationN3D3(HANDLE Handle, int Flags, int Axis0,
int Axis1,
int Axis2,int Zone, char* Axis0Commands, char* Axis1Commands, char*
Axis2Commands,
char* CorrectionMap0Variable,char* CorrectionMap1Variable,
char* CorrectionMap2Variable,  int ReferencedAxisOrAnalogInput0,
```

```
int ReferencedAxisOrAnalogInput1, int ReferencedAxisOrAnalogInput2,
ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Axis0	The index of the first axis that the mechanical error compensation will be applied to. Valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
axis1	The index of the second axis participating in 3D mechanical error compensation, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
axis2	The index of the second axis participating in 3D mechanical error compensation, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
Flags	ACSC_DECOMP_00 The mechanical error compensation will be applied to 'axis0' (default) ACSC_DECOMP_01 The mechanical error compensation will be applied to 'axis1' ACSC_DECOMP_02 The mechanical error compensation will be applied to 'axis1' ACSC_DECOMP_REFERENCED_AXIS Specifies that the mechanical error compensation will be calculated based on the feedback from the axis specified by the optional parameter. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_FIRST_ANALOG_INPUT Specifies that the first optional parameter will be regarded as an analog input index. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_SECOND_ANALOG_INPUT Specifies that the second optional parameter will be regarded as an analog input index. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_THIRD_ANALOG_INPUT Specifies that the third optional parameter will be regarded as an analog input index. 0, //No flags are set
Zone	The zone index, valid numbers are: 0, 1, 2, ... up to the maximum number of zones (10) minus 1. If '-1' is specified, all zones of specified axis will be affected.

axis0_command	The name of a real one-dimensional array that specifies 'axis0' command values used for correction table of mechanical error compensation. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
axis1_command	The name of a real one-dimensional array that specifies 'axis1' command values used for correction table of mechanical error compensation. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
axis2_command	The name of a real one-dimensional array that specifies 'axis2' command values used for correction table of mechanical error compensation. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_ map0	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '0' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map1	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '1' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map2	The name of a real two-dimensional array that specifies a correction table for mechanical error compensation in relation to the third specified coordinate of the Z-axis. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
referenced_ axis_or_analog_ input0	The index of the first axis, or the index of the first analog input whose feedback will be used to calculate the mechanical error compensation.
referenced_ axis_or_analog_ input1	The index of the second axis, or the index of the second analog input whose feedback will be used to calculate the mechanical error compensation.
referenced_ axis_or_analog_ input2	The index of the second axis, or the index of the third analog input whose feedback will be used to calculate the mechanical error compensation.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the

	operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Example

```
char Correction0[12]= "Correction0";
char Correction1[12]= "Correction1";
char Correction2[12]= "Correction2";
char Axis0Coordinates[17] = "Axis0Coordinates";
char Axis1Coordinates[17] = "Axis1Coordinates";
char Axis2Coordinates[17] = "Axis2Coordinates";
if (!acsc_DynamicErrorCompensationN3D3(hComm,
0, // flags
ACSC_AXIS_0, //axis0
ACSC_AXIS_1, // axis1
ACSC_AXIS_2, // axis2
0, //Zone
Axis0Coordinates, // The name of a real one-dimensional array that
specifies 'axis0'
Axis1Coordinates, // The name of a real one-dimensional array that
specifies 'axis1'
Axis2Coordinates,// The name of a real one-dimensional array that
specifies 'axis0'
Correction0, //Correction map 0
Correction1, //Correction map 1
Correction2, //Correction map 2
ACSC_NONE, // referenced_axis_or_analog_input0
ACSC_NONE, // referenced_axis_or_analog_input1
ACSC_NONE, // referenced_axis_or_analog_input2
ACSC_SYNCHRONOUS))
{
printf("acsc_DynamicErrorCompensationN3D3():Error Occurred -%d\n",acsc_
GetLastError());
   return;
}
```

4.40.15 acsc_DynamicErrorCompensationN3D5

Description

The acsc_DynamicErrorCompensationN3D5 function configures and activates 3D error correction for the mechanical error compensation of the 'axis0', 'axis1', and 'axis2' parameters for the specified zone, so that the compensated reference position will be calculated by adding the linearly (by default) interpolated error from the desired position so that the actual value will be closer to the desired value.

Syntax

```
acsc_DynamicErrorCompensationN3D5(HANDLE Handle, int Flags, int Axis0,
int Axis1,
int Axis2,int Zone, char* Axis0Commands, char* Axis1Commands, char*
Axis2Commands,
char* CorrectionMap0Variable,char* CorrectionMap1Variable,
char* CorrectionMap2Variable, char* CorrectionMap3Variable,
char* CorrectionMap4Variable, int ReferencedAxisOrAnalogInput0,
int ReferencedAxisOrAnalogInput1, int ReferencedAxisOrAnalogInput2,
ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Axis0	The index of the first axis that the mechanical error compensation will be applied to. Valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
axis1	The index of the second axis participating in 3D mechanical error compensation, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
axis2	The index of the second axis participating in 3D mechanical error compensation, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.

Flags	ACSC_DECOMP_00 The mechanical error compensation will be applied to 'axis0' (default) ACSC_DECOMP_01 The mechanical error compensation will be applied to 'axis1' ACSC_DECOMP_02 The mechanical error compensation will be applied to 'axis1' ACSC_DECOMP_REFERENCED_AXIS Specifies that the mechanical error compensation will be calculated based on the feedback from the axis specified by the optional parameter. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_FIRST_ANALOG_INPUT Specifies that the first optional parameter will be regarded as an analog input index. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_SECOND_ANALOG_INPUT Specifies that the second optional parameter will be regarded as an analog input index. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_THIRD_ANALOG_INPUT Specifies that the third optional parameter will be regarded as an analog input index. 0, //No flags are set
Zone	The zone index, valid numbers are: 0, 1, 2, ... up to the maximum number of zones (10) minus 1. If '-1' is specified, all zones of specified axis will be affected.
axis0_command	The name of a real one-dimensional array that specifies 'axis0' command values used for correction table of mechanical error compensation. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
axis1_command	The name of a real one-dimensional array that specifies 'axis1' command values used for correction table of mechanical error compensation. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
axis2_command	The name of a real one-dimensional array that specifies 'axis2' command values used for correction table of mechanical error compensation. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map0	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '0' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).

correction_map1	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '1' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map2	The name of a real two-dimensional array that specifies a correction table for mechanical error compensation in relation to the third specified coordinate of the Z-axis. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map3	The name of a real two-dimensional array that specifies a correction table for mechanical error compensation in relation to the fourth specified coordinate of the Z-axis. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_ map4	The name of a real two-dimensional array that specifies a correction table for mechanical error compensation in relation to the fifth specified coordinate of the Z-axis. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
referenced_ axis_or_analog_ input0	The index of the first axis, or the index of the first analog input whose feedback will be used to calculate the mechanical error compensation.
referenced_ axis_or_analog_ input1	The index of the second axis, or the index of the second analog input whose feedback will be used to calculate the mechanical error compensation.
referenced_ axis_or_analog_ input2	The index of the second axis, or the index of the third analog input whose feedback will be used to calculate the mechanical error compensation.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Example

```
char Correction0[12]= "Correction0";
char Correction1[12]= "Correction1";
char Correction2[12]= "Correction2";
char Correction3[12]= "Correction3";
char Correction4[12]= "Correction4";
char Axis0Coordinates[17] = "Axis0Coordinates";
char Axis1Coordinates[17] = "Axis1Coordinates";
char Axis2Coordinates[17] = "Axis2Coordinates";
if (!acsc_DynamicErrorCompensationN3D5(hComm,
0, // flags
ACSC_AXIS_0, //axis0
ACSC_AXIS_1, // axis1
ACSC_AXIS_2, // axis2
0, //Zone
Axis0Coordinates, // The name of a real one-dimensional array that
specifies 'axis0'
Axis1Coordinates, // The name of a real one-dimensional array that
specifies 'axis1'
Axis2Coordinates,// The name of a real one-dimensional array that
specifies 'axis0'
Correction0, //Correction map 0
Correction1, //Correction map 1
Correction2, //Correction map 2
Correction3, //Correction map 3
Correction4, //Correction map 4
ACSC_NONE, // referenced_axis_or_analog_input0
ACSC_NONE, // referenced_axis_or_analog_input1
ACSC_NONE, // referenced_axis_or_analog_input2
ACSC_SYNCHRONOUS))
{
printf("acsc_DynamicErrorCompensationN3D5():Error Occurred -%d\n",acsc_
GetLastError());
   return;
}
```

4.40.16 acsc_DynamicErrorCompensationN3DA

Description

The acsc_DynamicErrorCompensationN3DA function configures and activates 3D error correction for the mechanical error compensation of the 'axis0' , 'axis1' , and 'axis2' parameters for the specified zone, so that the compensated reference position will be calculated by adding the linearly (by default) interpolated error from the desired position so that the actual value will be closer to the desired value.

Syntax

```
acsc_DynamicErrorCompensationN3DA(HANDLE Handle, int Flags, int Axis0,
int Axis1,
```

```
int Axis2,int Zone, char* Axis0Commands, char* Axis1Commands, char*
Axis2Commands,
char* CorrectionMap0Variable, char* CorrectionMap1Variable,
char* CorrectionMap2Variable, char* CorrectionMap3Variable,
char* CorrectionMap4Variable, char* CorrectionMap5Variable,
char* CorrectionMap6Variable, char* CorrectionMap7Variable,
char* CorrectionMap8Variable, char* CorrectionMap9Variable,
int ReferencedAxisOrAnalogInput0, int ReferencedAxisOrAnalogInput1,
int ReferencedAxisOrAnalogInput2, ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Axis0	The index of the first axis that the mechanical error compensation will be applied to. Valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
axis1	The index of the second axis participating in 3D mechanical error compensation, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.
axis2	The index of the second axis participating in 3D mechanical error compensation, valid numbers are: 0, 1, 2, ... up to the number of axes in the system minus 1.

Flags	ACSC_DECOMP_00 The mechanical error compensation will be applied to 'axis0' (default) ACSC_DECOMP_01 The mechanical error compensation will be applied to 'axis1' ACSC_DECOMP_02 The mechanical error compensation will be applied to 'axis1' ACSC_DECOMP_REFERENCED_AXIS Specifies that the mechanical error compensation will be calculated based on the feedback from the axis specified by the optional parameter. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_FIRST_ANALOG_INPUT Specifies that the first optional parameter will be regarded as an analog input index. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_SECOND_ANALOG_INPUT Specifies that the second optional parameter will be regarded as an analog input index. ACSC_DECOMP_REFERENCED_AXIS | ACSC_DECOMP_THIRD_ANALOG_INPUT Specifies that the third optional parameter will be regarded as an analog input index. 0, //No flags are set
Zone	The zone index, valid numbers are: 0, 1, 2, ... up to the maximum number of zones (10) minus 1. If '-1' is specified, all zones of specified axis will be affected.
axis0_command	The name of a real one-dimensional array that specifies 'axis0' command values used for correction table of mechanical error compensation. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
axis1_command	The name of a real one-dimensional array that specifies 'axis1' command values used for correction table of mechanical error compensation. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
axis2_command	The name of a real one-dimensional array that specifies 'axis2' command values used for correction table of mechanical error compensation. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map0	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '0' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).

correction_map1	The name of a real two-dimensional array that specifies correction table for mechanical error compensation in relation to axis 2 step '1' coordinate. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map2	The name of a real two-dimensional array that specifies a correction table for mechanical error compensation in relation to the third specified coordinate of the Z-axis. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map3	The name of a real two-dimensional array that specifies a correction table for mechanical error compensation in relation to the fourth specified coordinate of the Z-axis. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_ map4	The name of a real two-dimensional array that specifies a correction table for mechanical error compensation in relation to the fifth specified coordinate of the Z-axis. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_ map5	The name of a real two-dimensional array that specifies a correction table for mechanical error compensation in relation to the sixth specified coordinate of the Z-axis. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_ map6	The name of a real two-dimensional array that specifies a correction table for mechanical error compensation in relation to the seventh specified coordinate of the Z-axis. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_map7	The name of a real two-dimensional array that specifies a correction table for mechanical error compensation in relation to the eighth specified coordinate of the Z-axis. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_ map8	The name of a real two-dimensional array that specifies a correction table for mechanical error compensation in relation to the ninth specified coordinate of the Z-axis. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
correction_ map9	The name of a real two-dimensional array that specifies a correction table for mechanical error compensation in relation to the tenth specified coordinate of the Z-axis. The array type should be GLOBAL REAL STATIC (defined in D-Buffer).
referenced_ axis_or_analog_ input0	The index of the first axis, or the index of the first analog input whose feedback will be used to calculate the mechanical error compensation.

referenced_ axis_or_analog_ input1	The index of the second axis, or the index of the second analog input whose feedback will be used to calculate the mechanical error compensation.
referenced_ axis_or_analog_ input2	The index of the second axis, or the index of the third analog input whose feedback will be used to calculate the mechanical error compensation.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Example

```
char Correction0[12]= "Correction0";
char Correction1[12]= "Correction1";
char Correction2[12]= "Correction2";
char Correction3[12]= "Correction3";
char Correction4[12]= "Correction4";
char Correction5[12]= "Correction5";
char Correction6[12]= "Correction6";
char Correction7[12]= "Correction7";
char Correction8[12]= "Correction8";
char Correction9[12]= "Correction9";
char Axis0Coordinates[17] = “Axis0Coordinates”;
char Axis1Coordinates[17] = “Axis1Coordinates”;
char Axis2Coordinates[17] = “Axis2Coordinates”;
if (!acsc_DynamicErrorCompensationN3DA(hComm,
0, // flags
ACSC_AXIS_0, //axis0
ACSC_AXIS_1, // axis1
ACSC_AXIS_2, // axis2
0, //Zone
Axis0Coordinates, // The name of a real one-dimensional array that
```

```
specifies 'axis0'
Axis1Coordinates, // The name of a real one-dimensional array that
specifies 'axis1'
Axis2Coordinates,// The name of a real one-dimensional array that
specifies 'axis0'
Correction0, //Correction map 0
Correction1, //Correction map 1
Correction2, //Correction map 2
Correction3, //Correction map 3
Correction4, //Correction map 4
Correction5, //Correction map 5
Correction6, //Correction map 6
Correction7, //Correction map 7
Correction8, //Correction map 8
Correction9, //Correction map 9
ACSC_NONE, // referenced_axis_or_analog_input0
ACSC_NONE, // referenced_axis_or_analog_input1
ACSC_NONE, // referenced_axis_or_analog_input2
ACSC_SYNCHRONOUS))
{
printf("acsc_DynamicErrorCompensationN3DA():Error Occurred -%d\n",acsc_
GetLastError());
   return;
}
```

4.40.17 acsc_DynamicErrorCompensationRemove

Description

The acsc_DynamicErrorCompensationRemove function receives axis index and zone index parameters. The function deactivates error correction for the mechanical error compensation for the specified zone.

Syntax

```
int acsc_DynamicErrorCompensationRemove(HANDLE Handle, int Axis, int
Zone, ACSC_WAITBLOCK* Wait);
```

Arguments

Handle	Communication handle.
Axis	Axis constant: ACSC_AXIS_0 corresponds to axis 0, ACSC_AXIS_1 – to axis 1, etc. For the axis constants see Axis Definitions.
Zone	The zone index, valid numbers are: 0, 1, 2, ... up to the maximum number of zones (10) minus 1. If '-1' is specified, all zones of specified axis will be affected.
Wait	Pointer to ACSC_WAITBLOCK structure. If Wait is ACSC_SYNCHRONOUS, the function returns when the controller

	response is received. If Wait points to a valid ACSC_WAITBLOCK structure, the function returns immediately. The calling thread must then call the acsc_ WaitForAsyncCall function to retrieve the operation result. If Wait is ACSC_IGNORE, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Example

```
if (!acsc_DynamicErrorCompensationRemove (hComm, ACSC_AXIS_0, 0, ACSC_
SYNCHRONOUS))
{
  printf("acsc_DynamicErrorCompensationRemove ():Error Occurred -%d\n",
   acsc_GetLastError());
   return;
}
```

4.41 Emergency Stop Functions

The Emergency Stop functions are:

Table 5-40. Emergency Stop Functions

Function	Description
acsc_RegisterEmergencyStop	Initiates Emergency Stop functionality.
acsc_UnregisterEmergencyStop	Deactivates Emergency Stop functionality.

4.41.1 acsc_RegisterEmergencyStop

Description

The function initiates the Emergency Stop functionality for calling application.

Syntax

int acsc_RegisterEmergencyStop();

Arguments

None

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

SPiiPlus UMD (User Mode Driver) and the C Library provide the user application with the ability to open/close the Emergency Stop button. Clicking the Emergency Stop button sends a **stop** command to all motions and motors to all channels, thereby stopping all motions and disabling all motors.

Figure 5-1. Emergency Stop Button

In the SPiiPlus UMD, when it has been selected, the Emergency Stop button stops all motions and disables all motors. Previously only SPiiPlus MMI provided such functionality. Now such functionality is also available for user applications.

Calling **acsc_RegisterEmergencyStop** will cause an Emergency Stop button to appear in the right bottom corner of the computer screen. If this button is already displayed, that is, activated by another application, a new button does not appear, but all functionality is available for the new application. Clicking the Emergency Stop button causes the stopping of all motions and motors command to all channels that are used in the calling application.

Calling **acsc_RegisterEmergencyStop** requires having the local host SPiiPlus UMD running, even if it is used through a remote connection, because the Emergency Stop button is part of the local SPiiPlus UMD. If the local SPiiPlus UMD is not running, the function fails.

Only a single call is required per application. It can be placed anywhere in code, even before the opening of communication with controllers.

An application can remove the Emergency Stop button by calling **acsc_UnregisterEmergencyStop**. The Emergency Stop button disappears if there are no additional registered applications using it. Termination of SPiiPlus UMD also removes the Emergency Stop button, so restarting the SPiiPlus UMD requires calling **acsc_RegisterEmergencyStop()** again.

Registering the Emergency Stop button more than once per application is meaningless, but the function succeeds anyway. In order to ensure that the Emergency Stop button is active, it is recommended to place a call to **acsc_RegisterEmergencyStop** after each call to any of **OpenComm***()** functions.

Example

```
int OpenCommunication()
{
        HANDLE Handle = OpenCommEthernet( // It can be any kind of OpenComm***
                "10.0.0.100", // TCP-IP address of the controller
                ACSC_SOCKET_DGRAM_PORT// point-to-point communication
                );
        if (Handle == ACSC_INVALID)
```

```
        {
                printf("error opening communication: %d\n",
                        acsc_GetLastError());
                return 0;
        }
        if (!acsc_RegisterEmergencyStop())
        {
                printf("Registration of EStop is failed. Error is: %d\n",
                acsc_GetLastError());
                return 0;
        }
        return 1;
}
```

4.41.2 acsc_UnregisterEmergencyStop

Description

The function terminates the Emergency Stop functionality for the calling application.

Syntax

int acsc_UnregisterEmergencyStop();

Arguments

None

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

The SPiiPlus User Mode Driver (UMD) and C Library provide user applications with the ability to activate or deactivate the Emergency Stop button displayed in the UMD.

Calling **acsc_UnregisterEmergencyStop** will cause application not to respond to the user clicking the Emergency Stop button displayed on the screen. If there are no other applications that have registered the Emergency Stop functionality (through the acsc_RegisterEmergencyStopfunction), the button will disappear.

Unregistering Emergency Stop more than once per application is meaningless, but the function will succeed anyway.

Example

```
int CloseCommunication()
{
        if (!acsc_UnregisterEmergencyStop())
        {
```

```
        printf("Unregistration of EStop is failed. Error is: %d\n",
                acsc_GetLastError());
                return 0;
        }
        if (!acsc_CloseComm(Handle))
        {
        printf("Error closing communication: %d\n", acsc_GetLastError());
        }
        return 1;
}
```

4.42 Application Save/Load Functions

The Application Save/Load functions are:

Table 5-41. Application Save/Load Functions

Function	Description
acsc_ AnalyzeApplication	Analyzes application file and returns information about the file components.
acsc_LoadApplication	Loads selected components of user application from a file on the host PC and saves it in the controller's flash memory.
acsc_SaveApplication	Saves selected components of user application from the controller's flash memory to a file on the host PC.
acsc_FreeApplication	Frees memory previously allocated by the acsc_AnalyzeApplication function.

4.42.1 acsc_AnalyzeApplication

Description

The function analyzes an application file and returns information about the file's components, such as, saved ACSPL+ programs, configuration parameters, user files, etc.

Syntax

int acsc_AnalyzeApplication(HANDLE Handle, char* fileName,
ACSC_APPSL_INFO** info (, ACSC_WAITBLOCK* Wait))

Arguments

Handle	Communication handle.
fileName	Filename (with included path) of the Application file.

Info	Pointer to the application information descriptor, defined by the ACSC_APPSL_INFO structure, must be explicitly initialized to NULL before running this function The **acsc_AnalyzeApplication** function is solely responsible for initializing this structure.
Wait	**Wait** has to be **ACSC_SYNCHRONOUS**, since only synchronous calls are supported for this function.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

If **fileName** is NULL, the current Controller Application will be analyzed; otherwise, the file specified by **fileName**, from the local hard disk, will be analyzed.

Example

```
ACSC_APPSL_INFO* ainfo = NULL;
if (!acsc_AnalyzeApplication(hComm,"C:\\needed_application_file.spi",
&ainfo))
        printf("AnalyzeApplication error: %d\n", acsc_GetLastError());
```

4.42.2 acsc_LoadApplication

Description

The function loads a section of data from the host PC disk and saves it in the controller's files.

Syntax

int acsc_LoadApplication(HANDLE Handle const char * fileName,
ACSC_APPLSL_INFO* info (, ACSC_WAITBLOCK* Wait))

Arguments

Handle	Communication handle.
fileName	Filename (with included path) of the Application file.
Info	Pointer to the Application information descriptor, defined by the ACSC_APPSL_INFO structure. The acsc_AnalyzeApplication function is solely responsible for initializing this structure.

Wait	**Wait** has to be ACSC_SYNCHRONOUS, since only synchronous calls are supported for this function.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

None.

Example

```
ACSC_APPSL_INFO* ainfo = NULL;
if (!acsc_AnalyzeApplication(hComm,"C:\\needed_application_file.spi",
&ainfo,NULL))
        printf("AnalyzeApplication error: %d\n", acsc_GetLastError());
if (!acsc_LoadApplication(hComm,"C:\\needed_application_file.spi",
ainfo,NULL))
        printf("LoadApplication error: %d\n", acsc_GetLastError());
```

4.42.3 acsc_SaveApplication

Description

The function saves a user application from the controller to a file on the host PC.

Syntax

int acsc_SaveApplication(HANDLE Handle, const char * fileName,
ACSC_APPLSL_INFO* info (, ACSC_WAITBLOCK* Wait))

Arguments

Handle	Communication handle.
fileName	Filename (with included path) of the Application file.
Info	Pointer to the Application information descriptor, defined by the ACSC_APPSL_INFO structure. The acsc_AnalyzeApplication function is solely responsible for initializing this structure.
Wait	**Wait** has to be ACSC_SYNCHRONOUS, since only synchronous calls are supported for this function.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

None.

Example

```
ACSC_APPSL_INFO* ainfo = NULL;
if (!acsc_AnalyzeApplication(hComm,NULL, &ainfo,NULL))
        printf("AnalyzeApplication error: %d\n", acsc_GetLastError());
if (!acsc_SaveApplication(hComm,"C:\\needed_application_file.spi",
ainfo,NULL))
        printf("SaveApplication error: %d\n", acsc_GetLastError());
```

4.42.4 acsc_FreeApplication

Description

The function frees memory previously allocated by the acsc_AnalyzeApplication function.

Syntax

int acsc_FreeApplication(ACSC_APPLSL_INFO* Info)

Arguments

Info	Pointer to the Application information descriptor, defined by the ACSC_APPSL_INFO structure. The acsc_AnalyzeApplication function is solely responsible for initializing this structure.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Example

```
ACSC_APPSL_INFO* ainfo = NULL;
if (!acsc_AnalyzeApplication(hComm,"C:\\needed_application_file.spi",
&ainfo,NULL))
        printf("AnalyzeApplication error: %d\n", acsc_GetLastError());
...
if (!acsc_FreeApplication(ainfo,NULL))
        printf("FreeApplication error: %d\n", acsc_GetLastError());
```

4.43 Reboot Functions

The Reboot functions are:

Table 5-42. Reboot Functions

Function	Description
acsc_ControllerReboot	Reboots controller and waits for process completion.
acsc_ControllerFactoryDefault	Reboots controller, restores factory default settings and waits for process completion.

4.43.1 acsc_ControllerReboot

Description

The function reboots controller and waits for process completion.

Syntax

int acsc_ControllerReboot(HANDLE Handle, int Timeout)

Arguments

Handle	Communication handle.
Timeout	Maximum waiting time in milliseconds.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

Example

```
hComm = acsc_OpenComm… ;
        if (hComm == ACSC_INVALID)
        {
printf("Error while opening communication: %d\n", acsc_GetLastError());
                return -1;
        }
        printf ("Communication with the controller was established
        successfully!\n");
        if (!acsc_ControllerReboot(hComm,30000)){
                printf("ControllerReboot error: %d\n", acsc_GetLastError());
                return -1;
        }
        printf ("Controller rebooted successfully, closing communication\n");
        acsc_CloseComm(hComm);
        hComm = acsc_OpenComm… ; //reopen communication
        if (hComm == ACSC_INVALID)
        {
printf("Error while reopening communication after reboot: %d\n", acsc_
GetLastError());
                return -1;
        }
printf ("Communication with the controller after reboot, was established
successfully!\n");
```

4.43.2 acsc_ControllerFactoryDefault

Description

The function reboots controller, restores factory default settings and waits for process completion.

Syntax

int acsc_ControllerFactoryDefault(HANDLE Handle, int Timeout)

Arguments

Handle	Communication handle.
Timeout	Maximum waiting time in milliseconds.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Comments

Example

```
hComm = acsc_OpenComm… ;
        if (hComm == ACSC_INVALID)
        {
printf("Error while opening communication: %d\n", acsc_GetLastError());
                return -1;
        }
        printf ("Communication with the controller was established
successfully \n");
        if (!acsc_ControllerFactoryDefault(hComm,30000)){
                printf("ControllerFactoryDefault error: %d\n",
acsc_GetLastError ());
                return -1;
        }
        printf ("Controller restarted successfully, closing communication\n");
        acsc_CloseComm(hComm);
        hComm = acsc_OpenComm… ; //reopen communication
        if (hComm == ACSC_INVALID)
        {
printf("Error while reopening communication after restart: %d\n",
acsc_GetLastError());
                return -1;
        }
        printf ("Communication with the controller after reboot, was
established successfully!\n");
```

4.44 Host-Controller File Operations

Host PC files can be copied to controller's non-volatile memory and user files can be deleted from the controller's non-volatile memory as described in this section.

Table 5-43. Host-Controller File Functions

Function	Description
acsc_CopyFileToController	The function copies files from the host PC to the controller's non-volatile memory.
acsc_ DeleteFileFromController	The function deletes user files from the controller's non-volatile memory.

4.44.1 acsc_CopyFileToController

Description

The function copies file from host PC to controller’s non-volatile memory.

Syntax

int _ACSCLIB_ WINAPI acsc_CopyFileToController(HANDLE Handle,
 char* SourceFileName, char* DestinationFileName,

ACSC_WAITBLOCK* Wait)

Arguments

Handle	Communication handle.
SourceFileName	Pointer to the null-terminated character string that contains name of the source file on host PC.
DestinationFileName	Pointer to the null-terminated character string that contains name of the destination file in controller's flash.
Wait	Pointer to **ACSC_WAITBLOCK** structure. If **Wait** is **ACSC_SYNCHRONOUS**, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is **ACSC_IGNORE**, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Extended error information can be obtained by calling **acsc_GetLastError**.

Example

```
if (!acsc_CopyFileToController(Handle, "C:\\ECAT.XML", "C:\\ECAT.XML",
NULL))
{
        printf("acsc_CopyFileToController(): Error Occurred - %d\n",
acsc_GetLastError());
        return;
}
```

4.44.2 acsc_DeleteFileFromController

Description

The function deletes user files from controller's non-volatile memory.

Syntax

int _ACSCLIB_ WINAPI acsc_DeleteFileFromController(HANDLE Handle, char* FileName, ACSC_WAITBLOCK* Wait)Arguments

Handle	Communication handle.
FileName	Pointer to the null-terminated character string that contains name of the user file on controller's non-volatile memory
Wait	Pointer to **ACSC_WAITBLOCK** structure. If **Wait** is **ACSC_SYNCHRONOUS**, the function returns when the controller response is received. If **Wait** points to a valid **ACSC_WAITBLOCK** structure, the function returns immediately. The calling thread must then call the acsc_WaitForAsyncCall function to retrieve the operation result. If **Wait** is **ACSC_IGNORE**, the function returns immediately. In this case, the operation result is ignored by the library and cannot be retrieved to the calling thread.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
if (!acsc_DeleteFileFromController(Handle, "MYARRAY", NULL))
{
     printf("acsc_DeleteFileFromController(): Error Occurred - %d\n",
         acsc_GetLastError());
}
```

4.45 Save to Flash

This section describes the function dealing with saving to flash:

Table 5-44. Save to Flash Function

Function	Description
acsc_ ControllerSaveToFlash	The function saves user application to the controller's non-volatile memory.

4.45.1 acsc_ControllerSaveToFlash

Description

The function saves user application to the controller's non-volatile memory.

Syntax

int _ACSCLIB_ WINAPI acsc_ControllerSaveToFlash(HANDLE Handle, int* Parameters, int* Buffers, int* SPPrograms, char* UserArrays)

Arguments

Handle	Communication handle.
Parameters	Array of parameters constants. Each element specifies system parameters or one involved axis: **ACSC_SYSTEM** corresponds to system parameters; **ACSC_AXIS_0** corresponds to axis 0, **ACSC_AXIS_1** to axis 1, etc. If all parameters need to be specified, **ACSC_PAR_ALL** should be used. After the last axis, one additional element must be located that contains -1 which marks the end of the array.
Buffers	Array of buffer constants. Each element specifies one involved buffer: **ACSC_BUFFER_0** corresponds to buffer 0, **ACSC_BUFFER_1** to buffer 1, etc. If all buffers need to be specified, **ACSC_BUFFER_ALL** should be used. After the last buffer, one additional element must be located that contains -1 which marks the end of the array.
SPPrograms	Array of Servo Processor (SP) constants. Each element specifies one involved SP: **ACSC_SP_0** corresponds to SP 0, **ACSC_SP_1** to SP 1, etc. If all SPs need to be specified, **ACSC_SP_ALL** should be used. After the last SP, one additional element must be located that contains -1 which marks the end of the array.  Servo Processor (SP) constants should be specified only if custom SP programs are used, otherwise this parameter should be **NULL**.
UserArrays	User Arrays list - Pointer to the null-terminated string. The string contains chained names of user arrays, separated by '\r'(13) character. If there is no need to save user arrays to controller's non-volatile memory, this parameter should be **NULL**.

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

 Extended error information can be obtained by calling **acsc_GetLastError**.

Example

```
int Axes[] = { ACSC_PAR_ALL, -1 };
int Buffers[] = { ACSC_BUFFER_ALL, -1 };
int SPPrograms[] = { ACSC_SP_1, -1 };
char UserArrays[] = "MyArray\rMyArray2\rMyArray3";
Example 1 - save all axes:
--------------------------
if (!acsc_ControllerSaveToFlash(
       Handle,                          // communication handle
       Axes,                            // Array of axis constants
       NULL,                            // Array of buffer constants
       NULL,                            // Array of SP constants
       NULL                             // User Arrays list
       ))
{
       printf("acsc_ControllerSaveToFlash(): Error Occurred - %d\n",
       acsc_GetLastError());
}
Example 2 - save all axes, all buffers, SP1 program, and User Arrays:
-----------------------------------------------------------------------
if (!acsc_ControllerSaveToFlash(
       Handle,                          // communication handle
       Axes,                            // Array of axis constants
       Buffers,                         // Array of buffer constants
       SPPrograms,                      // Array of SP constants
       UserArrays                       // User Arrays list
       ))
{
       printf("acsc_ControllerSaveToFlash(): Error Occurred - %d\n",
       acsc_GetLastError());
}
```

4.46 SPiiPlusSC Management

This section describes functions dealing with SPiiPlusSC management:

Table 5-45. SPiiPlusSC Management Functions

Function	Description
acsc_StartSPiiPlusSC	The function starts the SPiiPlusSC controller.
acsc_StopSPiiPlusSC	The function stops the SPiiPlusSC controller.

4.46.1 acsc_StartSPiiPlusSC

Description

The function starts the SPiiPlusSC controller.

Syntax

int _ACSCLIB_ WINAPI acsc_StartSPiiPlusSC()

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
// The function starts SPiiPlusSC controller
if (!acsc_StartSPiiPlusSC())
{
       printf("acsc_StartSPiiPlusSC(): Error Occurred - %d\n",
              acsc_GetLastError());
}
```

4.46.2 acsc_StopSPiiPlusSC

Description

The function stops the SPiiPlusSC controller.

Syntax

int _ACSCLIB_ WINAPI acsc_StopSPiiPlusSC()

Return Value

If the function succeeds, the return value is non-zero.

If the function fails, the return value is zero.

Example

```
// The function stops SPiiPlusSC controller
if (!acsc_StopSPiiPlusSC())
{
       printf("acsc_StopSPiiPlusSC(): Error Occurred - %d\n",
              acsc_GetLastError());
}
```

4.47 FRF Library

The C language FRF library is separate from the ACS C language library, and is available for an additional licensing fee.

4.47.1 acsc_FRF_Measure

Description

This function initializes FRF measurement.

Syntax

int acsc_FRF_Measure(HANDLE Handle, FRF_INPUT* inputParams, FRF_OUTPUT **outputParams, int *errorCode);

Arguments

Handle	Communications Handle
FRF_INPUT*	Array of input parameters
FRF_OUTPUT **	Pointer to array in which to write output results
int*	Pointer to error code

Return Value

Returns int indicating function success or failure.

Example

```
HANDLE ch;
char* ip;

ip = (char*)"10.0.0.100";
ch = acsc_OpenCommEthernet(ip, 701);
FRF_INPUT* input = new FRF_INPUT();
FRF_OUTPUT* output;
int errorCode = 0;
acsc_FRF_InitInput(input, &errorCode);
input->axis = 0;
input->loopType = FRF_LOOP_TYPE::PositionVelocity;
input->excitationType = FRF_EXCITATION_TYPE::WhiteNoise;
// Chirp type - Irrelevant for WhiteNoise excitation signal
input->chirpType = FRF_CHIRP_TYPE::LinearChirp;
input->filterType = FRF_WINDOW_TYPE::Hamming; // Basic choice for
WhiteNoise excitation signal
input->overlap = FRF_OVERLAP::HalfSignal; // Basic choice for WhiteNoise
excitation signal
input->frequencyDistributionType = FRF_FREQUENCY_DISTRIBUTION_
TYPE::Logarithmic;
// Frequency range parameters
input->startFreqHz = 30;
input->endFreqHz = 3000;
input->freqPerDec = 50;
input->highResolutionStart = 500;
input->highResolutionFreqPerDec = 500;
// Excitation amplitude
input->excitationAmplitudePercentIp = 1;
input->durationSec = 1;
input->numberOfRepetitions = 5;
// User defined signal definition
```

```
input->userDefinedExcitationSignal = nullptr;
// Recalculate parameters
input->inRaw = nullptr;
input->outRaw = nullptr;
input->lengthRaw = -1;
input->recalculate = true;
input->recalculate = false;
acsc_FRF_Measure(ch, input, &output, &errorCode);
```

4.47.2 acsc_FRF_Stop

Description

Function aborts FRF measurement. If FRF was aborted "outputParams" will not contain valid data (all pointers will be set to nullptr) and errorCode will reflect that FRF was aborted by the user.

Syntax

int acsc_FRF_Stop(HANDLE Handle, int Axis, int* errorCode);

Arguments

Handle	Communications Handle
int	axis to abort
int*	pointer to error code

Return Value

Returns int indicating function success or failure.

Example

```
int errorCode = 0;
acsc_FRF_Stop(ch, 0, &errorCode);
```

4.47.3 acsc_FRF_CalculateMeasurementDuration

Description

This function calculates the required duration of measurement (DurationSec parameter of FRFInput class) in order to satisfy the specified frequency resolution.

Syntax

int acacsc_FRF_CalculateMeasurementDuration(FRF_DURATION_CALCULATION_PARAMETERS *params, double *duration, int *errorCode);

Arguments

FRF_DURATION_CALCULATION_PARAMETERS *	Pointer to input parameters for duration calculation
double *duration	Address at which to write calculated duration
int*	Pointer to error code

Return Value

Returns int indicating function success or failure.

Example

```
int errorCode = 0;
double duration = 0;
FRF_DURATION_CALCULATION_PARAMETERS* input = new FRF_DURATION_
CALCULATION_PARAMETERS();
input->startFreqHz = 10;
input->endFreqHz = 1000;
input->freqPerDec = 100;
input->highResolutionStart = 500;
input->highResolutionFreqPerDec = 500;
input->frequencyDistributionType = FRF_FREQUENCY_DISTRIBUTION_TYPE::
Logarithmic;
input->frequencyHzResolutionForLinear = 0.1; // irrelevant if linear
frequency distribution selected

acsc_FRF_CalculateMeasurementDuration(input, &duration, &errorCode);
```

4.47.4 acsc_FRF_InitInput

Description

Initializes the inputParams structure of the "acsc_FRF_Measure" function to valid initial values.

Syntax

int acsc_FRF_InitInput(FRF_INPUT* inputParams, int* errorCode);

Arguments

FRF_INPUT*	Pointer to structure to be filled with valid initial values
int*	Pointer to error code

Return Value

Returns int indicating function success or failure.

Example

```
FRF_INPUT inputParams;
acsc_FRF_InitInput(FRF_INPUT* inputParams, int* errorCode);
```

4.47.5 acsc_FRF_FreeOutput

Description

Releases memory allocated for "outputParams" during calling to "acsc_FRF_CalculateMeasurementDuration"

Syntax

int acsc_FRF_FreeOutput(FRF_OUTPUT* inputParams, int* errorCode);

Arguments

FRF_OUTPUT *	Pointer to structure to be released
int*	Pointer to error code

Return Value

Returns int indicating function success or failure.

Example

```
FRF_INPUT* input = new FRF_INPUT();
int errorCode = 0;
acsc_FRF_InitInput(input, &errorCode);
```

4.47.6 acsc_FRF_ReadServoParameters

Description

The function reads all required servo parameters for calculation of Controller, OpenLoop, ClosedLoop etc.

Syntax

int acsc_FRF_ReadServoParameters(HANDLE Handle, int axis, SERVO_PARAMETERS* servoParameters, int* errorCode);

Arguments

Handle	Communications Handle
int	axis for parameter retrieval

SERVO_PARAMETERS*	Pointer to structure to be filled with servo parameters
int*	Pointer to error code

Return Value

Returns int indicating function success or failure.

Example

```
int errorCode = 0, axis = 0, port = 701;
HANDLE ch;
char* ip;
SERVO_PARAMETERS* servoParameters = static_cast<SERVO_PARAMETERS*>(malloc
(sizeof(SERVO_PARAMETERS)));
ip = (char*)"10.0.0.100";
ch = HANDLE();
ch = acsc_OpenCommEthernet(ip, port);
acsc_FRF_ReadServoParameters(ch, axis, servoParameters, &errorCode);
```

4.47.7 acsc_FRF_CalculateControllerFRD

Description

The function calculates controller FRD (Frequency Response Data) based on servo parameters, servo loop and frequency vector.

Syntax

int acsc_FRF_CalculateControllerFRD(SERVO_PARAMETERS* servoParameters, FRF_LOOP_TYPE loopType, double* frequencyHz, int frequencyLength, FRD **controllerFRD, int *errorCode);

Arguments

SERVO_PARAMETERS	Servo parameters for axis
FRF_LOOP_TYPE	Loop type to calculate
double*	array of input frequency values
int	length of frequency values array
int*	Pointer to error code

Return Value

FRD structure with calculation results

Example

```
HANDLE ch;
char* ip;
```

```
ip = (char*)"10.0.0.100";
ch = acsc_OpenCommEthernet(ip, 701);
SERVO_PARAMETERS* servoParameters = new SERVO_PARAMETERS();
int errorCode = 0, axis = 0;
acsc_FRF_ReadServoParameters(ch, axis, servoParameters, &errorCode);
int frequenciesLength = 3;
double* frequencies = new double[frequenciesLength] {100, 200, 300};

FRD* controllerFRD = new FRD();
acsc_FRF_CalculateControllerFRD(servoParameters, FRF_LOOP_
TYPE::PositionVelocity, frequencies, frequenciesLength, &controllerFRD,
&errorCode);
```

4.47.8 acsc_FRF_CalculateOpenLoopFRD

Description

This function calculates open loop FRD based on servo parameters, servo loop and frequency vector.

Syntax

int acsc_FRF_CalculateOpenLoopFRD(SERVO_PARAMETERS* servoParameters, FRD* plant, FRF_LOOP_TYPE loopType, FRD** openLoop, int* errorCode);

Arguments

SERVO_PARAMETERS*	Pointer to servo parmeters
FRD*	Plant FRD information
FRF_LOOP_TYPE	Loop type
FRD**	Storage for calculated values
int*	Pointer to error code

Return Value

Returns int indicating function success or failure.

Example

```
Example:
// Example shows how open loop may be calculated based on measured plant
and controller with
// adjusted parameters

HANDLE ch;
char* ip;

ip = (char*)"10.0.0.100";
ch = acsc_OpenCommEthernet(ip, 701);
```

```
FRF_INPUT* input = new FRF_INPUT();
FRF_OUTPUT* output;
int errorCode = 0;
acsc_FRF_InitInput(input, &errorCode);
//Assign proper values for "input"
acsc_FRF_Measure(ch, input, &output, &errorCode);
ServoParameters servoParameters = Ch.FRFReadServoParameters((Axis)0);
double origSLVKP = servoParameters->SLVKP;
servoParameters->SLVKP = origSLVKP*1.1;

FRD* openLoop;

acsc_FRF_CalculateOpenLoopFRD(servoParameters, output->plant, input-
>loopType, &openLoop, &errorCode);
```

4.47.9 acsc_FRF_CalculateClosedLoopFRD

Description

This function calculates closed loop FRD based on servo parameters, servo loop and frequency vector.

Syntax

int acsc_FRF_CalculateClosedLoopFRD(SERVO_PARAMETERS* servoParameters, FRD* plant, FRF_LOOP_TYPE loopType, FRD** closedLoop, int* errorCode);

Arguments

SERVO_PARAMETERS	servo parmeters
FRD*	plant FRD information
FRF_LOOP_TYPE	Loop type
FRD**	Storage for calculated values
int*	Pointer to error code

Return Value

Returns int indicating function success or failure.

Example

```
// Example shows how closed loop may be calculated based on measured
plant and controller with
// adjusted parameters

HANDLE ch;
char* ip;

ip = (char*)"10.0.0.100";
```

```
ch = acsc_OpenCommEthernet(ip, 701);
FRF_INPUT* input = new FRF_INPUT();
FRF_OUTPUT* output;
int errorCode = 0;
acsc_FRF_InitInput(input, &errorCode);
//Assign proper values for "input"
acsc_FRF_Measure(ch, input, &output, &errorCode);
ServoParameters servoParameters = Ch.FRFReadServoParameters((Axis)0);
double origSLVKP = servoParameters->SLVKP;
servoParameters->SLVKP = origSLVKP*1.1;

FRD* closedLoop;

acsc_FRF_CalculateClosedLoopFRD(servoParameters, output->plant, input-
>loopType, &closedLoop, &errorCode);
```

4.47.10 acsc_FRF_CalculateStabilityMargins

Description

The function calculates required stability margins based on the frequency response data of the open loop.

Syntax

int acsc_FRF_CalculateStabilityMargins(FRD* openLoop, FRF_STABILITY_MARGINS** stabilityMargins, int *errorCode);

Arguments

Handle	Communications Handle
FRD*	Pointer to FRD data
FRF_STABILITY_MARGINS**	Pointer to stability margins calculated
int*	error code

Return Value

Returns int indicating function success or failure.

Example

```
Api Ch = new Api();
Ch.OpenCommEthernet("10.0.0.100", 701);
FRF_INPUT* input = new FRF_INPUT();
acsc_FRF_InitInput(input, &errorCode);
//Assign proper valuer for "input"

FRF_OUTPUT* output;
acsc_FRF_Measure(ch, input, &output, &errorCode);
SERVO_PARAMETERS* servoParameters;
```

```
int axis = 0;
acsc_FRF_ReadServoParameters(ch, axis, servoParameters, &errorCode);
double origSLVKP = servoParameters->SLVKP;
servoParameters->SLVKP = origSLVKP * 1.1;
FRD* openLoop;
acsc_FRF_CalculateOpenLoopFRD(servoParameters, output->plant, input-
>loopType, &openLoop, &errorCode);
FRF_STABILITY_MARGINS* stabilityMargins;

acsc_FRF_CalculateStabilityMargins(output->plant, &stabilityMargins,
&errorCode);
```

4.47.11 acsc_FRF_FreeFRD

Description

Frees memory allocated for FRD

Syntax

int acsc_FRF_FreeFRD(FRD* inputParams, int* errorCode);

Arguments

FRD *	FRD data structure to release
int*	Pointer to error code

Return Value

Returns int indicating function success or failure.

Example

acsc_FRF_FreeOutput(output, &errorCode);

4.47.12 acsc_FRF_FreeStabilityMargins

Description

Function frees memory allocated for stability margins.

Syntax

int acsc_FRF_Stop(HANDLE Handle, int Axis, int* errorCode);

Arguments

Handle	Communications Handle
int	axis to abort
int*	Pointer to error code

Return Value

Returns int indicating function success or failure.

Example

```
int errorCode = 0;
acsc_FRF_Stop(ch, 0, &errorCode);
```

4.47.13 acsc_FFT

Description

Function calculates fast Fourier transform.

Syntax

int acsc_FFT(double* in, double* outReal, double* outImag, int length, int* errorCode);

Arguments

double*	Input data
outReal*	Real part of FFT
outImag*	Imaginary part of FFT
length	length of in and out arrays
int*	Pointer to error code

Return Value

Returns int indicating function success or failure.

4.47.14 acsc_JitterAnalysis

Description

Function executes jitter analysis.

Syntax

int _ACSCLIB_ WINAPI acsc_JitterAnalysis(JITTER_ANALYSIS_INPUT *in, JITTER_ANALYSIS_OUTPUT **out, int *errorCode);

Arguments

JITTER_ANALYSIS_INPUT*	Data for analysis
JITTER_ANALYSIS_OUTPUT**	Pointer to storage for analysis results
int*	Poiniter to error code

Return Value

Returns int indicating function success or failure.

Example

```
Api Ch = new Api();
Ch.OpenCommEthernet("10.0.0.100", 701);
JITTER_ANALYSIS_INPUT* jitterInput = new JITTER_ANALYSIS_INPUT();
JITTER_ANALYSIS_OUTPUT* jitterOutput = new JITTER_ANALYSIS_OUTPUT();   int
axis = 0;
char* dataCollectionArray = (char*)"dataCollectionPE(100000)";
acsc_DeclareVariable(ch, ACSC_REAL_TYPE, dataCollectionArray, NULL);
double CTIME, EFAC, sampleDuration = 5; //sample for 5 seconds
acsc_ReadReal(ch, -1, (char*)"CTIME", -1,-1,-1,-1, &CTIME, NULL);
acsc_ReadReal(ch, -1, (char*)"EFAC", axis, axis, -1, -1, &EFAC, NULL);
int nmumberOfSamples = (int)(sampleDuration * 20000);
//issue servo processor data collection
char* transaction = (char*)"SPDC/r dataCollectionPE, 100000, 1/20000, 0,
getspa(0,\"axes[0].PE\")";
char inBuf[101];
int receivedLength;
acsc_Transaction(ch, transaction, strlen(transaction), inBuf, 100,
&receivedLength, NULL);
Sleep(5000);
double* jitter = new double[nmumberOfSamples];
acsc_ReadReal(ch, -1, (char*)"dataCollectionPE", 0, nmumberOfSamples - 1,
-1, -1, jitter, NULL);

jitterInput->desiredFrequencyResolutionHz = 1;
jitterInput->frequencyBandsHz = new double[6] {0, 100, 150, 300, 500,
1000};
jitterInput->frequencyBandsHzLength = 6;
jitterInput->jitter = new double[nmumberOfSamples];
for (int k = 0; k < nmumberOfSamples; k++)
       jitterInput->jitter[k] = jitter[k] * EFAC;

jitterInput->jitterLength = nmumberOfSamples;
jitterInput->jitterFrequencyBandsCumulativeAmplitudeRMSthreshold = new
double[5] {1e-6, 2e-6, 1e-6, 2e-6, 1e-6};
jitterInput->samplingFrequencyHz = 20000;
jitterInput->windowType = FRF_WINDOW_TYPE::Hanning;
int errorCode = 0;
acsc_JitterAnalysis(jitterInput, &jitterOutput, &errorCode);
```

4.47.15 acsc_FRF_CrossCouplingMeasure

Description

This function executes cross-coupling effects measurement.

Syntax

```
acsc_FRF_CrossCouplingMeasure(HANDLE Handle, FRF_CROSS_COUPLING_INPUT*
inputParams, FRF_CROSS_COUPLING_OUTPUT** outputParams, int* errorCode)
```

Example

```
FRF_CROSS_COUPLING_INPUT* input = new FRF_CROSS_COUPLING_INPUT();
FRF_CROSS_COUPLING_OUTPUT* output;

int axes[] = { 0, 1 };
input->axes = axes;
input->axesLength = 2;
input->startFreqHz = 10;
input->endFreqHz = 1000;
input->freqPerDec = 30;
input->frequencyDistributionType = FRF_FREQUENCY_DISTRIBUTION_
TYPE::Linear;
input->excitationAmplitudePercentIp = new double[2];
input->excitationAmplitudePercentIp[0] = 1;
input->excitationAmplitudePercentIp[1] = 1;
input->numberOfRepetitions = 3;
input->durationSec = 0.02;
input->highResolutionStart = 10000;
input->highResolutionFreqPerDec = 500;
input->crossCouplingType = FRF_CROSS_COUPLING_TYPE::CompleteOpen;
acsc_FRF_CrossCouplingMeasure(ch, inputCC, &output, &errorCode);
```

5. Error Codes

Any error code greater than 1000 is a controller error defined in the *SPiiPlus Command & Variable Reference Guide*.

Table 6-1. Error Codes

Format	Error	Description
ACSC_ONLYSYNCHRONOUS	101	Asynchronous call is not supported.
ACSC_ENOENTLOGFILE	102	No such file or directory. This error is returned by the acsc_OpenLogFile function if a component of a path does not specify an existing directory.
ACS_OLD_FW	103	The FW version does not support the current C Library version. This error is returned by one of the **ACSC_OPENCOMM** functions, such as, acsc_OpenCommSerial. Upgrade the FW of the controller.
ACSC_MEMORY_OVERFLOW	104	Controllers reply is too long.
ACSC_EBADFLOGFILE	109	Internal library error: Invalid file handle.
ACSC_EINVALLOGFILE	122	Internal library error: Cannot open Log file.
ACSC_EMFILELOGFILE	124	Too many open files. This error is returned by the acsc_OpenLogFile function if no more file handles available.
ACSC_ENOSPCLOGFILE	128	No space left on device. This error is returned by the acsc_WriteLogFile function if no more space for writing is available on the device (for example, when the disk is full).
ACSC_TIMEOUT	130	A time out occurred while waiting for a controller response. This error indicates that during specified time-out the controller did not respond or the response was invalid.

Format	Error	Description
ACSC_SIMULATOR_NOT_RUN	131	An attempt to stop simulator was made without it being run.
ACSC_INITFAILURE	132	Communication initialization failure. Returned by one of the **ACSC_OPENCOMM** functions, such as, acsc_OpenCommSerial, in the following cases: The specified communication parameters are invalid The corresponding physical connection is not established The controller does not respond for specified communication channel.
ACSC_SIMULATOR_RUN_EXT	133	The default ports are occupied by another application, preventing the simulator from executing.
ACSC_INVALIDHANDLE	134	Invalid communication handle. Specified communication handle must be a handle returned by one of the **acsc_Open*** ** functions, such as, acsc_OpenCommSerial.
ACSC_ALLCHANNELSBUSY	135	All channels are busy. The maximum number of the concurrently opened communication channels is 10.
ACSC_SIMULATOR_NOT_SET	136	Necessary parameters for the simulator were not set via the SPiiPlus User Mode Driver, preventing the simulator from executing.
ACSC_RECEIVEDTOOLONG	137	Received message is too long (more than size of user buffer). This error cannot be returned and is present for compatibility with previous versions of the library.
ACSC_INVALIDBUFSIZE	138	The program string is long. This error is returned by one of the, acsc_AppendBuffer or acsc_LoadBuffer function if ACSPL+ program contains a string longer than 2032 bytes.

Format	Error	Description
ACSC_INVALIDPARAMETERS	139	Function parameters are invalid.
ACSC_CLOSEDHISTORYBUF	140	History buffer is closed.
ACSC_EMPTYNAMEVAR	141	Name of variable must be specified.
ACSC_INPUTPAR	142	Error in index specification. This error is returned by the acsc_ReadInteger, acsc_ReadReal, acsc_WriteInteger, or acsc_WriteReal functions if the parameters From1, To1, From2, To2 were specified incorrectly.
ACSC_RECEIVEDTOOSMALL	143	Controller reply contains less values than expected. This error is returned by the acsc_ReadInteger or acsc_ReadReal functions.
ACSC_FUNCTIONNOTSUPPORTED	145	Function is not supported in current version.
ACSC_INITHISTORYBUFFAILED	147	Internal error: Error of the history buffer initialization.
ACSC_CLOSEDMESSAGEBUF	150	Unsolicited messages buffer is closed.
ACSC_SETCALLBACKERROR	151	Callback registration error. This error is returned by the acsc_GetCallbackMask function for any of the communication channels. In the present version of library, only PCI Bus communication supports user callbacks.
ACSC_CALLBACKALREADYSET	152	Callback function has been already installed. This error is returned by the acsc_InstallCallback function if the application tries to install another callback function for the same interrupt that was already used. Only one callback can be installed for each interrupt.
ACSC_CHECKSUMERROR	153	Checksum of the controller response is incorrect.

Format	Error	Description
ACSC_ REPLIESSEQUENCEERROR	154	Internal library error: The controller replies sequence is invalid.
ACSC_WAITFAILED	155	Internal library error: WaitForSingleObject function returns error.
ACSC_INITMESSAGEBUFFAILED	157	Internal library error: Error of the unsolicited messages buffer initialization.
ACSC_OPERATIONABORTED	158	Non-waiting call has been aborted. This error occurs in the following cases: acsc_CancelOperation function returns this error if the corresponding call has been aborted by user request. acsc_WaitForAsyncCall function returns this error if the parameter Wait contains invalid pointer.
ACSC_ CANCELOPERATIONERROR	159	Error of the non-waiting call cancellation. This error is returned by the acsc_ CancelOperation function if the parameter Wait contains invalid pointer.
ACSC_COMMANDSQUEUEFULL	160	Queue of transmitted commands is full. The maximum number of the concurrently transmitted commands is 256. Check how many waiting calls were initiated and how many non-waiting (asynchronous) calls are in progress so far.
ACSC_SENDINGFAILED	162	The library cannot send to the specified communication channel. Check physical connection with the controller (or settings) and try to reconnect.
ACSC_RECEIVINGFAILED	163	The library cannot receive from the specified communication channel. Check physical connection with the controller (or settings) and try to reconnect.
ACSC_CHAINSENDINGFAILED	164	Internal library error: Sending of the chain is failed.

Format	Error	Description
ACSC_DUPLICATED_IP	165	Specified IP address is duplicated.
ACSC_APPLICATION_NOT_FOUND	166	There is no Application with such Handle.
ACSC_ARRAY_EXPECTED	167	Array name was expected.
ACSC_INVALID_FILE_FORMAT	168	The file is not a valid ANSI data file.
ACSC_APPSL_CRC	171	Application Saver Loader CRC error.
ACSC_APPSL_HEADERCRC	172	Application Saver Loader Header CRC error.
ACSC_APPSL_FILESIZE	173	Application Saver Loader File Size error.
ACSC_APPSL_FILEOPEN	174	Application Saver Loader File Open error.
ACSC_APPSL_UNKNOWNFILE	175	Application Saver Loader Unknown File error.
ACSC_APPSL_VERERROR	176	Application Saver Loader Format Version error.
ACSC_APPSL_SECTION_SIZE	177	Application Saver Loader Section Size is Zero.
ACSC_TLSERROR	179	Internal library error: Thread local storage error.
ACSC_INITDRIVERFAILED	180	Error of the PCI driver initialization. Returned by the acsc_GetPCICards function in the following cases: SPiiPlus PCI driver is not installed correctly or the version of the SPiiPlus PCI Bus driver is incorrect In this case, it is necessary to reinstall the SPiiPlus PCI driver (WINDRIVER) and the library.
ACSC_INVALIDPOINTER	185	Pointer to the buffer is invalid. Returned by the acsc_WaitForAsyncCall function if the parameter Buf is not the same pointer that was specified for SPiiPlus C function call.
ACSC_SETPRIORITYERROR	189	Specified priority for the callback thread cannot be set. Returned by the acsc_SetCallbackPriority function in the following cases:

Format	Error	Description
		Specified priority value is not supported by the operating system or cannot be set by the function or the function was called by any of the communication channels. In the present version of library, only PCI Bus communication supports user callbacks.
ACSC_DIRECTDPRAMACCESS	190	Cannot access DPRAM directly through any channel but PCI and Direct. Returned by DPRAM access functions, when attempting to call them with Serial or Ethernet channels.
ACSC_INVALID_DPRAM_ADDR	192	Invalid DPRAM address was specified Returned by DPRAM access functions, when attempting to access illegal address
ACSC_OLD_SIMULATOR	193	This version of simulator does not support work with DPRAM. Returned by DPRAM access functions, when attempting to access old version Simulator that does not support DPRAM.
ACSC_FILE_NOT_FOUND	195	Returned by functions that work with host file system when a specified filename is not found. Check the path and filename.
ACSC_SERVEREXCEPTION	197	The application cannot establish communication with the SPiiPlus UMD. Returned by one of the **ACSC_OPENCOMM** functions. Check the following: SPiiPlus UMD is loaded (whether the UMD icon  appears in the Task tray). SPiiPlus UMD shows an error message. In case of remote connection, access from a remote application is enabled.
ACSC_STOPPED_RESPONDING	198	The controller does not reply for more than 20 seconds.

Format	Error	Description
		Returned by any function that exchanges data with the controller. Check the following: Controller is powered on (MPU LED is green) Controller connected properly to host Controller executes a time consuming command like compilation of a large program, save to flash, load to flash, etc.
ACSC_DLL_UMD_VERSION[1]	199	The DLL and the UMD versions are not compatible. Returned by one of the **ACSC_OPENCOMM** functions, such as, acsc_OpenCommSerial. Verify that the files ACSCL.DLL and ACSCSRV.EXE are of the same version.
ACSC_FRF_INPUT_START_ FREQUENCY_OUT_OF_RANGE	200	FRF - start frequency is out of range
ACSC_FRF_INPUT_END_ FREQUENCY_OUT_OF_RANGE	201	FRF - end frequency is out of range
ACSC_FRF_INPUT_START_ FREQUENCY_IS_HIGHER_ THAN_END_FREQUENCY	202	FRF - start frequency is above or equal to the end frequency
ACSC_FRF_INPUT_ FREQPERDEC_OUT_OF_RANGE	203	FRF - frequency per decade parameter is out of range
ACSC_FRF_INPUT_HR_ FREQPERDEC_OUT_OF_RANGE	204	FRF - high resolution frequency per decade parameter is out of range
ACSC_FRF_INPUT_ FREQUENCY_RESOLUTION_ LINEAR_OUT_OF_RANGE	205	FRF - linear frequency resolution is out of range
ACSC_FRF_INPUT_ AMPLITUDE_OUT_OF_RANGE	206	FRF - amplitude out of range
ACSC_FRF_INPUT_AXIS_OUT_ OF_RANGE	207	FRF - axis is out of range
ACSC_FRF_INPUT_NUMBER_ OF_REPETITIONS_OUT_OF_ RANGE	208	FRF - number of repetitions is out of range

Format	Error	Description
ACSC_FRF_INPUT_DURATION_OUT_OF_RANGE	209	FRF - duration is out of range
ACSC_FRF_INPUT_ENUM_OUT_OF_RANGE	210	FRF - ENUM is out of range
ACSC_FRF_MEMORY_ALLOCATION_FAILED_AT_HOST	211	FRF - memory allocation failed at host computer
ACSC_FRF_DATA_READ_FROM_CONTROLLER_INCONSISTENT	212	FRF - internal error. Data inconsistent
ACSC_FRF_DSP_DOESNT_HAVE_REQUIRED_PARAMETERS	213	FRF - dsp doesn't support frf measurement
ACSC_FRF_FAILED_TO_COMMUNICATE_WITH_CONTROLLER	214	FRF - failed to communicate with controller
FRF ACSC_FRF_FAILED_TO_READ_SERVO_PARAMETERS	215	FRF - internal error. Failed to read servo parameters
ACSC_FRF_DUMMY_AXIS_NOT_SUPPORTED	216	FRF - dummy axis is not supported
ACSC_FRF_MOTOR_SHOULD_BE_SET_TO_CLOSED_LOOP	217	FRF - motor should be set to closed loop
ACSC_FRF_MOTOR_SHOULD_BE_ENABLED	218	FRF - motor should be enabled
ACSC_FRF_MOTOR_SHOULD_COMMUTATED	219	FRF - motor should be commutated",
ACSC_FRF_SPDC_IS_ALREADY_IN_PROGRESS	220	FRF - data collection already in progress, process aborted
ACSC_FRF_ABORTED_BY_USER	221	FRF - measurement aborted by the user
ACSC_FRF_MOTOR_DISABLED_DURING_MEASUREMENT	222	FRF - motor was disabled during measurement
ACSC_FRF_DISABLE_OR_FAULT_OCCURED_DURING_MEASUREMENT	223	FRF - disable or fault occured during measurement

Format	Error	Description
ACSC_FRF_FAULT_OCCURED_DURING_MEASUREMENT	224	FRF - fault occured during measurement
ACSC_FRF_ARRAY_SIZES_INCOMATIBLE	225	FRF - internal error, array size is incompatible
ACSC_FRF_NUMBER_OF_POINTS_SHOULD_BE_POSITIVE	226	FRF - internal error, number of points should be positive
ACSC_FRF_MEMORY_ALLOCATION_FAILED_AT_CONTROLLER	227	FRF - memory allocation failed at controller
ACSC_FRF_EXCITATION_DURATION_IS_TOO_LONG	228	FRF - excitation duration is too long
ACSC_FRF_USER_DEFINED_EXCITATION_SIGNAL_REQUIRED_BUT_NOT_DEFINED	229	FRF - user defined excitation type required but excitation signal was not available
ACSC_FRF_USER_DEFINED_EXCITATION_SIGNAL_OUT_OF_BOUNDARIES	230	FRF - user defined signal excitation values are out of range
ACSC_FRF_FRD_LENGTH_TOO_SHORT	231	FRF - FRD length is too short
ACSC_FRF_FRD_FREQUENCIES_SHOULD_BE_CONTINUOUSLY_INCREASING	232	FRF - FRD Frequencies should be continuously increasing
ACSC_JITTER_ANALYSIS_JITTER_ARRAY_TOO_SHORT	233	Jitter array is too short
ACSC_JITTER_ANALYSIS_SAMPLING_FREQUENCY_NOT_VALID	234	Jitter sampling freqeuncy is too low
ACSC_JITTER_ANALYSIS_FWINDOW_TYPE_NOT_SUPPORTED	235	Jitter analysis window type is not supported
ACSC_JITTER_ANALYSIS_FREQUENCY_RANGE_NOT_VALID	236	Jitter analysis frequency range is not valid

Format	Error	Description
ACSC_JITTER_ANALYSIS_ FREQUENCY_RESOLUTION_ NOT_VALID	237	Jitter analysis frequency resolution is not valid
ACSC_LICENSE_COMMON_ PROBLEM	238	Common license problem
ACSC_LICENSE_DONGLE_NOT_ FOUND	239	License dongle not found
ACSC_LICENSE_ENTRY_NOT_ FOUND	240	License entry not found
ACSC_LICENSE_INVALID_ HANDLE	241	Invalid license handle
ACSC_LICENSE_NO_DATA_ AVAILABLE	242	No license data available
ACSC_LICENSE_INVALID_PN	243	Invalid license part number
ACSC_SC_INCORRECT_PROC_ ALLOC	244	SPiiPlusSC - incorrect Windows Processor Allocation
ACSC_SC_MISSING_DRIVERS[1]	245	SPiiPlusSC drivers are missing
ACSC_SC_INCORRECT_ MEMORY[1]	246	SPiiPlusSC - incorrect Memory Reservations
ACSC_SC_RTOS_SERVICE[1]	247	SPiiPlusSC - RTOS Service is not running
ACSC_SC_REBOOT[1]	248	SPiiPlusSC - System requires reboot
ACSC_SC_DONGLE_VERSION[1]	249	Detected CmStick Version of Dongle is too old

[1]Errors 244-249 are relevant to the **SPiiPlusSC** product only

6. Constants

This chapter presents the constants that are incorporated in the SPiiPlus C Library.

6.1 General

6.1.1 ACSC_SYNCHRONOUS

Description

Indicates a synchronous call.

Value

0

6.1.2 ACSC_INVALID

Description

Invalid communication handle.

Value

-1

6.1.3 ACSC_NONE

Description

Placeholder for redundant values, like the second index in a one-dimensional array.

Value

-1

6.1.4 ACSC_IGNORE

Description

Used for non-waiting calls with neglect of operation results.

Value

0xFFFFFFFF

6.1.5 ACSC_INT_TYPE

Description

Integer type of the variable.

Value

1

6.1.6 ACSC_STATIC_INT_TYPE

Description

Integer type of a static variable.

Value

3

6.1.7 ACSC_REAL_TYPE

Description

Real type of the variable.

Value

2

6.1.8 ACSC_STATIC_REAL_TYPE

Description

Real type of a static variable.

Value

4

6.1.9 ACSC_COUNTERCLOCKWISE

Description

Counterclockwise rotation.

Value

1

6.1.10 ACSC_CLOCKWISE

Description

Clockwise rotation.

Value

-1

6.1.11 ACSC_POSITIVE_DIRECTION

Description

A move in positive direction.

Value

1

6.1.12 ACSC_NEGATIVE_DIRECTION

Description

A move in negative direction.

Value

-1

6.2 General Communication Options

6.2.1 ACSC_COMM_USECHECKSUM

Description

The communication mode when each command is sent to the controller with checksum and the controller also responds with checksum

Value

0x000000001

6.2.2 ACSC_COMM_AUTORECOVER_HW_ERROR

Description

When a hardware error is detected in the communication channel and this bit is set, the library automatically repeats the transaction without counting iterations. By default, this flag is not set.

Value

0x000000002

6.3 Ethernet Communication Options

6.3.1 ACSC_SOCKET_DGRAM_PORT

Description

The library opens Ethernet communication using the connection-less socket and UDP communication protocol.

Value

700

6.3.2 ACSC_SOCKET_STREAM_PORT

Description

The library opens Ethernet communication using the connection-oriented socket and TCP communication protocol.

Value

701

6.4 Axis Definitions

The general format for any axis definition is:

ACSC_AXIS_index

where **index** is a number that ranges between 0 and 127, such as **ACSC_AXIS_0**, **ACSC_AXIS_1**, **ACSC_AXIS_127**, etc. The axis constant contains the value associated with the **index**, that is, **ACSC_AXIS_0** has a value of 0, **ACSC_AXIS_1** has a value of 1, and so forth.

6.5 Buffer Definitions

The general format for any buffer definition is:

ACSC_BUFFER_index

Where index is a number that ranges between 0 and 64, such as **ACSC_BUFFER_0**, **ACSC_BUFFER_1**, **ACSC_BUFFER_64**, etc. The axis constant contains the value associated with the index, that is, **ACSC_BUFFER_0** has a value of 0, **ACSC_BUFFER_1** has a value of 1, and so forth. **ACSC_BUFFER_ALL** stands for all buffers.

6.6 Servo Processor (SP) Definitions

The general format for any SP definition is:

ACSC_SP_index

Where index is a number that ranges between 0 and 63, such as **ACSC_SP_0**, **ACSC_SP_1**, **ACSC_SP_63**, etc. The axis constant contains the value associated with the index, that is, **ACSC_SP_0** has a value of 0, **ACSC_SP_1** has a value of 1, and so forth. **ACSC_SP_ALL** stands for all SPs.

6.7 Motion Flags

6.7.1 ACSC_AMF_WAIT

Description

The controller plans the motion but doesn't start it until the acsc_Go function is executed.

Position Event Generation (PEG): The execution of the PEG is delayed until the acsc_StartPegNT function is executed.

Value

0x00000001

6.7.2 ACSC_AMF_RELATIVE

Description

The value of the point coordinate is relative to the end point coordinate of the previous motion.

Value

0x00000002

6.7.3 ACSC_AMF_VELOCITY

Description

The motion uses the specified velocity instead of the default velocity.

Value

0x00000004

6.7.4 ACSC_AMF_ENDVELOCITY

Description

The motion comes to the end point with the specified velocity

Value

0x00000008

6.7.5 ACSC_AMF_POSITIONLOCK

Description

The slaved motion uses position lock. If the flag is not specified, velocity lock is used.

Value

0x00000010

6.7.6 ACSC_AMF_VELOCITYLOCK

Description

The slaved motion uses velocity lock.

Value

0x00000020

6.7.7 ACSC_AMF_CYCLIC

Description

The motion uses the point sequence as a cyclic array: after positioning to the last point it does positioning to the first point and continues.

Value

0x00000100

6.7.8 ACSC_AMF_VARTIME

Description

The time interval between adjacent points of the spline (arbitrary path) motion is non-uniform and is specified along with an each added point. If the flag is not specified, the interval is uniform.

Value

0x00000200

6.7.9 ACSC_AMF_CUBIC

Description

Use a cubic interpolation between the specified points (third-order spline) for the spline (arbitrary path) motion. If the flag is not specified, linear interpolation is used (first-order spline).

Currently third-order spline is not supported.

Value

0x00000400

6.7.10 ACSC_AMF_EXTRAPOLATED

Description

Segmented slaved motion: if a master value travels beyond the specified path, the last or the first segment is extrapolated.

Value

0x00001000

6.7.11 ACSC_AMF_STALLED

Description

Segmented slaved motion: if a master value travels beyond the specified path, the motion stalls at the last or first point.

Value

0x00002000

6.7.12 ACSC_AMF_SYNCHRONOUS

Description

Position Event Generation (PEG): Start PEG synchronously with the motion sequence.

Value

0x00008000

6.7.13 ACSC_AMF_MAXIMUM

Description

Multi-axis motion does not use the motion parameters from the leading axis but calculates the maximum allowed motion velocity, acceleration, deceleration and jerk of the involved axes.

Value

0x00004000

6.7.14 ACSC_AMF_JUNCTIONVELOCITY

Description

Decelerate to corner.

Value

0x00010000

6.7.15 ACSC_AMF_ANGLE

Description

Do not treat junction as a corner, if junction angle is less than or equal to the specified value in radians.

Value

0x00020000

6.7.16 ACSC_AMF_USERVARIABLES

Description

Synchronize user variables with segment execution.

Value

0x00040000

6.7.17 ACSC_AMF_INVERT_OUTPUT

Description

Position Event Generation (PEG): The PEG pulse output is inverted.

Value

0x00080000

6.7.18 ACSC_AMF_CURVEVELOCITY

Description

Decelerate to curvature discontinuity point.

Value

0x00100000

6.7.19 ACSC_AMF_CORNERDEVIATION

Description

Use a corner rounding option with the specified permitted deviation.

Value

0x00200000

6.7.20 ACSC_AMF_CORNERRADIUS

Description

Use a corner rounding option with the specified permitted curvature.

Value

0x00400000

6.7.21 ACSC_AMF_CORNERLENGTH

Description

Use automatic corner rounding option.

Value

0x00800000

6.7.22 ACSC_AMF_DWELLTIME

Description

Dwell time between segments.

Value

0x00100000

6.7.23 ACSC_AMF_BSEGTIME

Description

Segment time.

Value

0x00004000

6.7.24 ACSC_AMF_BSEGACC

Description

Segment acceleration time.

Value

0x00020000

6.7.25 ACSC_AMF_BSEGJERK

Description

Segment jerk time.

Value

0x00008000

6.7.26 ACSC_AMF_CURVEAUTO

Description

Automatic curve calculations

Value

0x01000000

6.7.27 ACSC_AMF_AXISLIMIT

Description

Axis velocity limitation enforcement

Value

0x00002000

6.8 Data Collection Flags

6.8.1 ACSC_DCF_TEMPORAL

Description

Temporal data collection. The sampling period is calculated automatically according to the collection time.

Value

0x00000001

6.8.2 ACSC_DCF_CYCLIC

Description

Cyclic data collection uses the collection array as a cyclic buffer and continues infinitely. When the array is full, each new sample overwrites the oldest sample in the array.

Value

0x00000002

6.8.3 ACSC_DCF_SYNC

Description

Starts data collection synchronously to a motion. Data collection started with the **ACSC_DCF_SYNC** flag is called ***axis data collection***.

Value

0x00000004

6.8.4 ACSC_DCF_WAIT

Description

Creates synchronous data collection, but does not start until the acsc_Go function is called. This flag can only be used with the ACSC_DCF_SYNC flag.

Value

0x00000008

6.9 Motor State Flags

6.9.1 ACSC_MST_ENABLE

Description

Motor is enabled

Value

0x00000001

6.9.2 ACSC_MST_INPOS

Description

Motor has reached a target position.

Value

0x00000010

6.9.3 ACSC_MST_MOVE

Description

Motor is moving.

Value

0x00000020

6.9.4 ACSC_MST_ACC

Description

Motor is accelerating.

Value

0x00000040

6.10 Axis State Flags

6.10.1 ACSC_AST_LEAD

Description

Axis is leading in a group.

Value

0x00000001

6.10.2 ACSC_AST_DC

Description

Axis data collection is in progress.

Value

0x00000002

6.10.3 ACSC_AST_PEG

Description

PEG for the specified axis is in progress.

Value

0x00000004

6.10.4 ACSC_AST_MOVE

Description

Axis is moving.

Value

0x00000020

6.10.5 ACSC_AST_ACC

Description

Axis is accelerating.

Value

0x00000040

6.10.6 ACSC_AST_DECOMPON

Description

Dynamic error compensation is active

Value

0x04000000

6.10.7 ACSC_AST_SEGMENT

Description

Construction of segmented motion for the specified axis is in progress.

Value

0x00000080

6.10.8 ACSC_AST_VELLOCK

Description

Slave motion for the specified axis is synchronized to master in velocity lock mode.

Value

0x00000100

6.10.9 ACSC_AST_POSLOCK

Description

Slave motion for the specified axis is synchronized to master in position lock mode.

Value

0x00000200

6.11 Index and Mark State Flags

6.11.1 ACSC_IST_IND

Description

Primary encoder index of the specified axis is latched.

Value

0x00000001

6.11.2 ACSC_IST_IND2

Description

Secondary encoder index of the specified axis is latched.

Value

0x00000002

6.11.3 ACSC_IST_MARK

Description

MARK1 signal has been generated and position of the specified axis was latched.

Value

0x00000004

6.11.4 ACSC_IST_MARK2

Description

MARK2 signal has been generated and position of the specified axis was latched.

Value

0x00000008

6.12 Program State Flags

6.12.1 ACSC_PST_COMPILED

Description

Program in the specified buffer is compiled.

Value

0x00000001

6.12.2 ACSC_PST_RUN

Description

Program in the specified buffer is running.

Value

0x00000002

6.12.3 ACSC_PST_SUSPEND

Description

Program in the specified buffer is suspended after the step execution or due to breakpoint in debug mode.

Value

0x00000004

6.12.4 ACSC_PST_DEBUG

Description

Program in the specified buffer is executed in debug mode, i.e. breakpoints are active.

Value

0x00000020

6.12.5 ACSC_PST_AUTO

Description

Auto routine in the specified buffer is running.

Value

0x00000080

6.13 Safety Control Masks

6.13.1 ACSC_SAFETY_RL

Description

Motor fault - Hardware Right Limit

Value

0x00000001

6.13.2 ACSC_SAFETY_LL

Description

Motor fault - Hardware Left Limit

Value

0x00000002

6.13.3 ACSC_SAFETY_NETWORK

Description

Network error.

Value

4

6.13.4 ACSC_SAFETY_HOT

Description

Motor fault - Motor Overheat

Value

0x00000010

6.13.5 ACSC_SAFETY_SRL

Description

Motor fault - Software Right Limit

Value

0x00000020

6.13.6 ACSC_SAFETY_SLL

Description

Motor fault - Software Left Limit

Value

0x00000040

6.13.7 ACSC_SAFETY_ENCNC

Description

Motor fault - Primary Encoder Not Connected

Value

0x00000080

6.13.8 ACSC_SAFETY_ENC2NC

Description

Motor fault - Secondary Encoder Not Connected

Value

0x00000100

6.13.9 ACSC_SAFETY_DRIVE

Description

Motor fault - Driver Alarm

Value

0x00000200

6.13.10 ACSC_SAFETY_ENC

Description

Motor fault - Primary Encoder Error

Value

0x00000400

6.13.11 ACSC_SAFETY_ENC2

Description

Motor fault - Secondary Encoder Error

Value

0x00000800

6.13.12 ACSC_SAFETY_PE

Description

Motor fault - Position Error

Value

0x00001000

6.13.13 ACSC_SAFETY_CPE

Description

Motor fault - Critical Position Error

Value

0x00002000

6.13.14 ACSC_SAFETY_VL

Description

Motor fault - Velocity Limit

Value

0x00004000

6.13.15 ACSC_SAFETY_AL

Description

Motor fault - Acceleration Limit

Value

0x00008000

6.13.16 ACSC_SAFETY_CL

Description

Motor fault - Current Limit

Value

0x00010000

6.13.17 ACSC_SAFETY_SP

Description

Motor fault - Servo Processor Alarm

Value

0x00020000

6.13.18 ACSC_SAFETY_PROG

Description

System fault - Program Error

Value

0x02000000

6.13.19 ACSC_SAFETY_MEM

Description

System fault - Memory Overflow

Value

0x04000000

6.13.20 ACSC_SAFETY_TIME

Description

System fault - MPU Overuse

Value

0x08000000

6.13.21 ACSC_SAFETY_ES

Description

System fault - Hardware Emergency Stop

Value

0x10000000

6.13.22 ACSC_SAFETY_INT

Description

System fault - Servo Interrupt

Value

0x20000000

6.13.23 ACSC_SAFETY_INTGR

Description

System fault - File Integrity

Value

0x40000000

See the *SPiiPlus ACSPL+ Command & Variable Reference Guide* for detailed explanations of faults

6.14 Callback Interrupts

There are three types of Callback Interrupts:

- Hardware Callback Interrupts
- Software Callback Interrupts
- User Callback Interrupts

6.14.1 Hardware Callback Interrupts

6.14.1.1 ACSC_INTR_EMERGENCY

Description

EMERGENCY STOP signal has been generated.

Value

15

6.14.2 Software Callback Interrupts

6.14.2.1 ACSC_INTR_PHYSICAL_MOTION_END

Description

Physical motion has finished.

Value

16

6.14.2.2 ACSC_INTR_LOGICAL_MOTION_END

Description

Logical motion has finished.

Value

17

6.14.2.3 ACSC_INTR_MOTION_FAILURE

Description

Motion has been interrupted due to a fault.

Value

18

6.14.2.4 ACSC_INTR_MOTOR_FAILURE

Description

Motor has been disabled due to a fault.

Value

19

6.14.2.5 ACSC_INTR_PROGRAM_END

Description

ACSPL+ program has finished.

Value

20

6.14.2.6 ACSC_INTR_ACSPL_PROGRAM_EX

Description

ACSPL+ program has generated the interrupt by INTERRUPTEX command.

Value

21

6.14.2.7 ACSC_INTR_ACSPL_PROGRAM

Description

ACSPL+ program has generated the interrupt by **INTERRUPT** command.

Value

22

6.14.2.8 ACSC_INTR_MOTION_START

Description

Motion Starts

Value

24

6.14.2.9 ACSC_INTR_MOTION_PHASE_CHANGE

Description

Motion Profile Phase changes

Value

25

6.14.2.10 ACSC_INTR_TRIGGER

Description

AST.#TRIGGER bit goes high

Value

26

6.14.2.11 ACSC_INTR_NEWSEGM

Description

AST.#NEWSEGM bit goes high.

Value

27

6.14.2.12 ACSC_INTR_SYSTEM_ERROR

Description

System error has occurred.

Value

28

6.14.2.13 ACSC_INTR_ETHERCAT_ERROR

Description

EtherCAT error has occurred

Value

29

6.14.3 User Callback Interrupts

6.14.3.1 ACSC_INTR_COMM_CHANNEL_CLOSED

Description

Communication channel has been closed.

Value

32

6.14.3.2 ACSC_INTR_SOFTWARE_ESTOP

Description

ACS EStop button was clicked.

Value

33

6.15 Callback Interrupt Masks

Table 7-1. Callback Interrupt Masks

Bit Name	Bit	Desc.	Interrupt
ACSC_MASK_AXIS_0 ... ACSC_MASK_AXIS_63	0 ... 63	Axis 0 ... Axis 63	ACSC_INTR_PHYSICAL_MOTION_END, ACSC_INTR_LOGICAL_MOTION_END, ACSC_INTR_MOTION_FAILURE, ACSC_INTR_MOTION_START, ACSC_INTR_MOTION_PHASE_CHANGE, ACSC_INTR_TRIGGER

Bit Name	Bit	Desc.	Interrupt
ACSC_MASK_AXIS_64 ACSK_MASK_AXIS_127	0 ... 63	Axis 64 ... Axis 127	ACSC_INTR_PHYSICAL_MOTION_END, ACSC_INTR_LOGICAL_MOTION_END, ACSC_INTR_MOTION_FAILURE, ACSC_INTR_MOTION_START, ACSC_INTR_MOTION_PHASE_CHANGE, ACSC_INTR_TRIGGER
ACSC_MASK_BUFFER_0 ... ACSC_MASK_BUFFER_63	0 ... 63	Buffer 0 ... Buffer 63	ACSC_INTR_PROGRAM_END, ACSC_INTR_COMMAND, ACSC_INTR_ACSPL_PROGRAM

6.16 Configuration Keys

Table 7-2. Configuration Keys

Key Name	Key	Description
ACSC_CONF _WORD1_KEY	1	Bit 6 defines HSSI route, bit 7 defines source for interrupt generation.
ACSC_CONF _INT_EDGE_KEY	3	Sets the interrupt edge to be positive or negative.
ACSC_CONF _ENCODER_KEY	4	Sets encoder type: A&B or analog.
ACSC_CONF_OUT_KEY	29	Sets the specified output pin to be one of the following: OUT0 PEG Brake
ACSC_CONF _MFLAGS9_KEY	204	Controls value of **MFLAGS.9**
ACSC_CONF_DIGITAL_SOURCE_KEY	205	Assigns use of OUT0 signal: general purpose output or PEG output.
ACSC_CONF_SP_OUT_PINS_KEY	206	Reads SP output pins.
ACSC_CONF_BRAKE_OUT_KEY	229	Controls brake function.

6.16.1 System Information Keys

Table 7-3. System Information Keys

Key Name	Key	Description
ACSC_SYS_MODEL_KEY	1	The SPiiPlus model number.
ACSC_SYS_VERSION_KEY	2	The SPiiPlus version number.
ACSC_SYS_NBUFFERS_KEY	10	Number of regular ACSPL+ program buffers.
ACSC_SYS_DBUF_INDEX_KEY	11	D-buffer index.
ACSC_SYS_NAXES_KEY	13	Total number of axes (in current configuration).
ACSC_SYS_NNODES_KEY	14	Number of EtherCAT nodes.
ACSC_SYS_NDCCH_KEY	15	Number of data collection channels per Servo Processor.
ACSC_SYS_ECAT_KEY	16	EtherCAT support: 1 - Yes 0 - No

7. Structures

This chapter details the structures that are available for SPiiPlus C programming.

7.1 ACSC_WAITBLOCK

Description

The structure is used for non-waiting calls of the SPiiPlus C functions.

Syntax

```
struct
{
 HANDLE Event;
 int Ret;
} ACSC_WAITBLOCK;
```

Arguments

Event	Not used
Ret	The completion of a task

Comments

To initiate a non-waiting call the user thread declares the ACSC_WAITBLOCK structure and passes the pointer to this structure to SPiiPlus C function as parameter. When a thread activates a non-waiting call, the library passes the request to an internal thread that sends the command to the controller and then monitors the controller responses. When the controller responds to the command, the internal thread stores the response in the internal buffer. The calling thread can retrieve the response with help of the acsc_WaitForAsyncCall function and validate the **Ret**.

The error codes stored in **Ret** are the same that the acsc_GetLastError function returns.

7.2 ACSC_PCI_SLOT

Description

The structure defines a physical location of PCI card. This structure is used in the acsc_GetPCICards function and contains the information about detected PCI card.

Syntax

```
struct
{
 unsigned int BusNumber;
 unsigned int SlotNumber;
 unsigned int Function;
} ACSC_PCI_SLOT;
```

Arguments

BusNumber	The Bus number
SlotNumber	Slot number of the controller card
Function	PCI function of the controller card

Comments

The **SlotNumber** can be used in the **acsc_OpenCommPCI** call in order to open communication with a specific card. Other members have no use in SPiiPlus C Library.

7.3 ACSC_HISTORYBUFFER

Description

The structure defines a state of the history and message buffers. This structure is used by the acsc_OpenHistoryBuffer and acsc_OpenMessageBuffer functions.

Syntax

```
struct
{
 int Max;
 int Cur;
 int Ring;
 char* Buf
} ACSC_HISTORYBUFFER;
```

Arguments

Max	Buffer size
Cur	Number of bytes currently stored in the buffer
Ring	Circular index in the buffer
Buf	Pointer to the buffer

Comments

Max is equal to the requested Size and never changes its value.

Cur is a number of bytes currently stored in the buffer. From the beginning, **Cur** is zero, then grows up to **Max,** and then remains unchanged.

Ring is a circular index in the buffer: if the buffer is full, the most recent byte is stored in position **Ring**-1, the earliest byte is stored in position **Ring**.

The user program must never change the members in the structure or write to the history buffer. However, the user program can read the structure and the buffer directly. If doing so, the user should be aware that the library includes a separate thread that watches the replies from the controller. For this reason the contents of the buffer and structure members can change asynchronously to the user thread.

7.4 ACSC_CONNECTION_DESC

Description

The structure defines controller connection for an application. Used in the acsc_GetConnectionsList and acsc_TerminateConnection functions.

Syntax

```
struct
{
 char Application[100];
 HANDLE Handle;
 DWORD ProcessId
} ACSC_CONNECTION_DESC;
```

Arguments

Application	Name of the application, maximum of 100 characters.
handle	The channel's Handle.
ProcessID	The ID of the process.

7.5 ACSC_CONNECTION_INFO

Description

The structure provides information about specified controller connection for an application. Used in the acsc_GetConnectionInfo function.

Syntax

```
struct
{
 ACSC_CONNECTION_TYPE Type;
 int SerialPort;
 int SerialBaudRate;
 int PCISlot;
 int EthernetProtocol;
 char EthernetIP[100];
 int EthernetPort;
} ACSC_CONNECTION_INFO;
```

Arguments

Type	Connection Type
SerialPort	Communication channel of serial communication: 1 corresponds to COM1, 2 – to COM2, etc.
SerialBaudRate	Communication rate of serial communication in bits per second (baud).

PCISlot	Number of the PCI slot of the controller card.
EthernetProtocol	Ethernet protocol.
EthernetIP	Pointer to a null-terminated character string that contains the network address of the controller in symbolic or TCP/IP dotted form.
EthernetPort	Service port.

7.6 AXMASK_EXT

Description

The structure is used in functions supporting 128 axes.

Syntax

```
typedef struct _AXMASK_EXT
{
    unsigned __int64 AXMASK64; //0-63 axes
    unsigned __int64 AXMASK128; //64-127
    unsigned __int64 Reserved1;
    unsigned __int64 Reserved2;
} AXMASK_EXT;
```

Related Functions

acsc_InstallCallbackExt, **acsc_SetCallbackMaskExt**,

7.7 Application Save/Load Structures

The following structures are used with the Application Save/Load functions:

7.7.1 ACSC_APPSL_STRING

Description

The structure defines a string in the application file.

Syntax

```
struct
{
 int length;
 char *string
} ACSC_APPSL_STRING;
```

Arguments

length	Length of the string
string	Pointer to the start of the string

7.7.2 ACSC_APPSL_SECTION

Description

The structure defines the application section to be loaded or saved.

Syntax

```
struct
{
 ACSC_APPSL_FILETYPE type;
 ACSC_APPSL_STRING filename;
 ACSC_APPSL_STRING description;
 int size;
 int offset;
 int CRC;
 int inuse;
 int error;
 char *data
} ACSC_APPSL_SECTION;
```

Arguments

type	Section type (see ACSC_APPSL_FILETYPE for a description).
filename	Section filename.
description	Section description.
size	Size, in bytes, of the data.
offset	Offset in the data section.
CRC	Data CRC.
inuse	0 - Not in use. 1 - In use.
error	Associated error code.
data	Pointer to the start of the data.

7.7.3 ACSC_APPSL_ATTRIBUTE

Description

The structure defines an attribute key-value pair.

Syntax

```
struct
{
 ACSC_APPSL_STRING key;
 ACSC_APPSL_STRING value
} ACSC_APPSL_ATTRIBUTE;
```

Arguments

key	Attribute's key.
value	The value of the key.

7.7.4 ACSC_APPSL_INFO

Description

The structure defines an application file structure, including the header, attributes and file sections.

Syntax

```
struct
{
 ACSC_APPSL_STRING filename;
 ACSC_APPSL_STRING description;
 int isNewFile;
 int ErrCode;
 int attributes_num;
 ACSC_APPSL_ATTRIBUTE *attributes;
 int sections_num;
 ACSC_APPSL_SECTION *sections;
} ACSC_APPSL_INFO;
```

Arguments

filename	Name of the file.
description	File description.
isNewFile	1 - File is new. 0 - File exists and has been modified.
ErrCode	Error code from the controller.
attributes_num	Number of file attributes.
attributes	Pointer to file attributes.
sections_num	Number of file sections.
sections	Pointer to start of file sections.

7.8 FRF Structures

7.8.1 FRF_INPUT

Description

The structure defines the input for the FRF excitation signal.

Syntax

```
struct
{
 int axis;
loopType FRF_LOOP_TYPE ;
excitationType FRF_EXCITATION_TYPE;
FRF_CHIRP_TYPE chirpType;
windowType FRF_WINDOW_TYPE;
overlap FRF_OVERLAP;
FrequencyDistributionType FRF_FREQUENCY_DISTRIBUTION_TYPE;
double startFreqHz;
double endFreqHz;
int freqPerDec;
double highResolutionStart;
int highResolutionFreqPerDec;
double excitationAmplitudePercentIp;
double durationSec;
int numberOfRepetitions;
double *userDefinedExcitationSignal
int userDefinedExcitationSignalLength;
double *inRaw;
double *outRaw;
int lengthRaw;
bool recalculate;
} FRF_INPUT;
```

Arguments

int	axis	Axis where excitation signal is injected
FRF_LOOP_TYPE	loopType	ACS specific definition of servo loop that defines the type of Plant that will be measured

FRF_ EXCITATION_ TYPE	excitationType	Defines the excitation signal type **White noise**: generated for duration specified by **durationSec** and **numberOfRepetitions**. Signal is fully uncorrelated (not pseudo-random noise). Standard deviation is determined by the **excitationAmplitudePercentIp** . **ChirpPeriodic**: frequency range defined by the startFreqHz and endFreqHz is covered during the time period defined by the durationSec. The signal is repeated as defined in **numberOfRepetitions**. **UserDefined:** The signal is repeated as defined in **numberOfRepetitions**.
FRF_CHIRP_TYPE	chirpType	Defines the type of Chirp signal. Applicable only if excitationType is *ChirpPeriodic*
FRF_WINDOW_ TYPE	windowType	Defines the window type for filtering the signals before computing the FFT
FRF_OVERLAP	overlap	Defines the amount of signals overlap for Welch averaging

FRF_FREQUENCY_DISTRIBUTION_TYPE	DistributionType	Describes distribution of points of the resulted Plant. Has two options: Logarithmic: in this case user may specify **startFreqHz**, **endFreqHz**, **freqPerDec**, **HighResolutionStart** and **highResolutionFreqPerDec**. If **HighResolutionStart** falls between **startFreqHz** and **endFreqHz** and **highResolutionFreqPerDec** > **freqPerDec** then frequency range will have two regions with different frequency densities. Otherwise only the **startFreqHz**, **endFreqHz**, **freqPerDec** will determine the frequency range and density. Linear: frequency range is determined by the **startFreqHz** and **endFreqHz**, while frequency resolution is by duration of the single measurement.
double	startFreqHz	Defines start frequency of the resulted plant in Hz
double	endFreqHz	Defines endfrequency of the resulted plant in Hz
int	freqPerDec	Defines number of points per decade for standard resolution
double	highResolutionStart	Frequency where high resolution starts. High resolution feature is active only if highResolutionStart is within (startFreqHz-endFreqHz) range
int	highResolutionFreqPerDec	Defines number of frequencies per decade for high resolution range. High resolution feature is active only if highResolutionStart is within (startFreqHz-endFreqHz) range

double	excitationAmplitudePercentIp	Excitation amplitude of the signal defined in % of the drives peak current. In case of white noise this parameter determines the standard deviation of the noise.
double	durationSec	Duration of a single excitation in seconds. For periodic chirp it is the time required to go through all frequencies once. This parameter is ignored if selected excitationType = UserDefined
int	numberOfRepititions	Number of repetitions of the excitation signal. In case of **excitationType** = ChirpPeriodic or UserDefined the signal is repeated according to **numberOfRepetitions** value. In case of white noise overall excitation duration will be excitation duration multiplied by the **numberOfRepetitions**.
double*	userDefinedExcitationSignal	Pointer to user defined excitation signal
int	userDefinedExcitationSignalLength	Length of the user defined excitation signal.
double*	inRaw	Raw excitation signal as measured during FRF calculation. Used only if **recalculate** = true
double*	outRaw	Raw measures signal as measured during FRF calculation. Used only if **recalculate** = true
int	lengthRaw	length of the raw signals. Used only if **recalculate** = true
bool	recalculate	Recalculates the FRF instead of measuring it again. For debug only.

7.8.2 FRF_OUTPUT

Description

The structure holds data returned by the FRF.

Syntax

```
struct
{
 FRD* plant;
FRD* controller;
FRD* openLoop;
FRD* closedLoop;
FRD* sensitivity;
FRD* coherence;
FRF_LOOP_TYPE loopType;
FRF_STABILITY_MARGINS* stabilityMargins;
double excitationAmplitude;
double* inRaw;
double* outRaw;
int lengthRaw;
} FRF_OUTPUT;
```

Arguments

FRD*	plant	Measured frequency response data (FRD) of the Plant
FRD*	controller	Calculated FRD of the controller. Calculation is based on standard servo parameters available in SERVO_PARAMETERS structure
FRD*	openLoop	Calculated FRD of the openLoop based on plant and controller FRDs
FRD*	closedLoop	Calculated FRD of the closedLoop based on plant and controller FRD's. Closed loop will take into account SLAFF influence when it is relevant
FRD*	sensitivity	Calculated FRD of the sensitivity based on plant and controller FRD's
FRD*	coherence	Measured FRD of the coherence amplitude. Imaginary part of the coherence amplitude is 0
FRF_LOOP_TYPE	loopType	ACS specific definition of servo loop that defines the type of Plant that will be measured

FRF_ STABILITY_ MARGINS*	stabilityMargins	Stability margins of the resulting closed loop. In case of the openLoop measurement stability margins are calculated for **PositionVelocity** loop
double	excitationAmplitude	Excitation amplitude used during measurement
double*	inRaw	Raw excitation signal as measured during FRF calculation. Used only if **recalculate** = true
double*	outRaw	Raw measured signal as measured during FRF calculation. Used only if **recalculate** = true
int	lengthRaw	Length of the raw signals. Used only if **recalculate** = true

7.8.3 FRF_DURATION_CALCULATION_PARAMETERS

Description

The structure contains the results of the duration calculation.

Syntax

```
struct
{
 double startFreqHz;
double endFreqHz;
int freqPerDec;
double highResolutionStart;
int highResolutionFreqPerDec;
FRF_FREQUENCY_DISTRIBUTION_TYPE FrequencyDistributionType;
int frequencyHzResolutionForLinear;
} FRF_DURATION_CALCULATION_PARAMETERS;
```

Arguments

double	startFreqHz	Defines start frequency of the resulted plant in Hz *Ignored if frequency distribution type is set to Linear
double	endFreqHz	Defines end frequency of the resulted plant in Hz *Ignored if frequency

		distribution type is set to Linear
int	freqPerDec	Defines number of points per decade for standard resolution *Ignored if frequency distribution type is set to Linear
double	highResolutionStart	Frequency where high resolution starts. High resolution feature is active only if **highResolutionStart** is within (**startFreqHz**-**endFreqHz**) range *Ignored if frequency distribution type is set to Linear
int	highResolutionFreqPerDec	Defines number of frequencies per decade for high resolution range. High resolution feature is active only if **highResolutionStart** is within (**startFreqHz**-**endFreqHz**) range *Ignored if frequency distribution type is set to Linear
FRF_FREQUENCY_DISTRIBUTION_TYPE	FrequencyDistributionType	Describes distribution of points of the resulted Plant. *Ignored if frequency distribution type is set to Linear
int	frequencyHzResolutionForLinear	Defines required frequency resolution if frequency distribution type is set to Linear, otherwise ignored.

7.8.4 FRD

Description

The structure holds FRD data.

Syntax

```
struct
{
 double* real;
double* imag;
double* frequencyHz;
size_t length;
} FRD;
```

Arguments

double*	real	Real part of FRD
double*	imag	Imaginary part of FRD
double*	frequencyHz	Frequencies in Hz units
size_t	length	Length of real, imaginary and frequencyHz arrays in C library

7.8.5 FRF_STABILITY_MARGINS

Description

The structure .

Syntax

```
struct
{
 double* gainMarginArray;
double* gainMarginArrayFrequencyHz;
size_t gainMarginArrayLength;
double gainMarginWorst;
double gainMarginWorstFrequencyHz;
double* phaseMarginArray;
double* phaseMarginArrayFrequencyHz;
size_t phaseMarginArrayLength;
double phaseMarginWorst;
double phaseMarginWorstFrequencyHz;
double modulusMargin;
double modulusMarginFrequencyHz;
double bandwidth;
} FRF_STABILITY_MARGINS;
```

Arguments

double*	gainMarginArray	Array of gain margin values
double*	gainMarginArrayFrequencyHz	Array of gain margin frequency values in Hz.
size_t	gainMarginArrayLength	Length of the **gainMarginArray** and **gainMarginArrayFrequencyHz** arrays
double	gainMarginWorst	Most critical gain margin
double	gainMarginWorstFrequencyHz	Most critical stability margin frequency in Hz.
double*	phaseMarginArray	Array of phase margin values
double*	phaseMarginArrayFrequencyHz	Array of phase margin frequency values in Hz.
size_t	phaseMarginArrayLength	Length of the phaseMarginArray and phaseMarginArrayFrequencyHz length
double	phaseMarginWorst	Most critical phase margin
double	phaseMarginWorstFrequencyHz	Most critical phase margin frequency in Hz.
double	modulusMargin	Modulus margin
double	modulusMarginFrequencyHz	Modulus margin frequency in Hz.
double	bandwidth	Bandwidth Hz. Defined as first 0dB cross of the open loop

7.8.6 JITTER_ANALYSIS_INPUT

Description

The structure holds input values for jitter analysis.

Syntax

```
struct
{
 double* jitter;
size_t jitterLength;
size_t samplingFrequencyHz;
double desiredFrequencyResolutionHz;
```

```
double* frequencyBandsHz;
size_t frequencyBandsHzLength;
double jitterFrequencyBandsCumulativeAmplitudeRMSthreshold;
FRF_WINDOW_TYPE windowType;
} JITTER_ANALYSIS_INPUT;
```

Arguments

double*	jitter	Jitter values. In most cases position error should be used for the analysis.
size_t	jitterLength	Length of jitter array in C library
size_t	samplingFrequencyHz	Sampling frequency of jitter signal
double	desiredFrequencyResolutionHz	Desired frequency resolution of the jitter analysis result in frequency domain
double*	frequencyBandsHz	Frequency bands for jitter analysis For example, 0,1000,2000,5000 defines three frequency bands 1. 0-1000Hz 2. 1000-2000Hz 3. 2000-5000Hz
size_t	frequencyBandsHzLength	Length of the frequencyBandsHz array
double*	jitterFrequencyBandsCumulativeAmplitudeRMSthreshold	Threshold for jitter level for frequency bands defined by **frequencyBandsHz** parameter. The length of this parameter should be **frequencyBandsHzLength** - 1

FRF_ WINDOW_ TYPE	windowType	Defines the window type for filtering the signals before computing the FFT

7.8.7 JITTER_ANALYSIS_OUTPUT

Description

The structure holds the results of the jitter analysis.

Syntax

```
struct
{
 double* jitterAmplitudeRMS;
double* jitterCumulativeAmplitudeRMS;
double* frequencyHz;
size_t frequencyLength;
double* jitterFrequencyBandsCumulativeAmplitudeRMS;
double* frequencyBandsHz;
size_t frequencyBandsHzLength;
int jitterFrequencyBandsResultBool;
double jitterRMS;
double jitterAmplitudePeak2Peak;
} JITTER_ANALYSIS_OUTPUT;
```

Arguments

double*	jitterAmplitudeRMS	Jitter values in frequency domain representing rms value of the jitter at unit of the Jitter variable in Jitter analysis input.
double*	jitterCumulativeAmplitudeRMS	Cumulative jitter rms values as a function of frequencies. Rapid change in this graph implies a problematic area in frequency domain
double*	frequencyHz	Frequency array for **jitterAmplitudeRMS** and **jitterCumulativeAmplitudeRMS**
size_t	frequencyLength	Length of **frequencyHz**, **jitterAmplitudeRMS** and **jitterCumulativeAmplitudeRMS**

double*	jitterFrequencyBandsCumulativeAmplitudeRMS	Jitter RMS values in frequency bands defined by the **frequencyBandsHz** parameter of the jitter analysis input parameter The length of this array is frequencyBandsHzLength - 1
double*	frequencyBandsHz	Frequency bands as defined in the **frequencyBandsHz** parameter of the jitter analysis input parameter
size_t	frequencyBandsHzLength	Length of the **frequencyBandsHz** parameter
int	jitterFrequencyBandsResultBool	true if all values in **jitterFrequencyBandsCumulativeAmplitudeRMS** are below **JitterFrequencyBandsCumulativeAmplitudeRMSthreshold**
double	jitterRMS	RMS value of jitter in time domain
double	jitterAmplitudePeak2Peak	Peak to peak value of jitter in time domain

7.8.8 SERVO_PARAMETERS

Description

The structure holds the axis servo parameters read from the controller.

Syntax

```
struct
{
 double SLIKP;
 double SLIKP;
 double SLILI;
double SLPKP;
 double SLPKI;
double SLPLI;
 double SLVKP;
 double SLVKI;
 double SLVLI;
 double SLVSOF;
 double SLVSOFD;
 double SLVNFRQ;
 double SLVNWID;
 double SLVNATT;
 double SLVB0NF;
 double SLVB0DF;
 double SLVB0ND;
 double SLVB0DD;
 double SLVB1NF;
 double SLVB1DF;
 double SLVB1ND;
 double SLVB1DD;
 double XVEL;
 double EFAC;
 double SLVRAT;
 double SLAFF;
 INT32 MFLAGS;
 INT32 MFLAGSX;
} SERVO_PARAMETERS;
```

Arguments

double	SLIKP	value read from controller
double	SLIKP	value read from controller
double	SLILI	value read from controller
double	SLPKP	value read from controller
double	SLPKP	value read from controller
double	SLPKI	value read from controller
double	SLPLI	value read from controller

double	SLVKP	value read from controller
double	SLVLI	value read from controller
double	SLVSOF	value read from controller
double	SLVSOFD	value read from controller
double	SLVNFRQ	value read from controller
double	SLVNWID	value read from controller
double	SLVNATT	value read from controller
double	SLVB0NF	value read from controller
double	SLVB0DF	value read from controller
double	SLVB0ND	value read from controller
double	SLVB0DD	value read from controller
double	SLVB1NF	value read from controller
double	SLVB1DF	value read from controller
double	SLVB1ND	value read from controller
double	SLVB1DD	value read from controller
double	XVEL	value read from controller
double	EFAC	value read from controller
double	SLVRAT	value read from controller
double	SLAFF	value read from controller
INT32	MFLAGS	value read from controller
INT32	MFLAGSX	value read from controller

7.8.9 FRF_CROSS_COUPLING_INPUT

Description

Sets the input parameters for measurement of cross-coupling effects.

Name	Type	Default and Range	Description
axes	Int *	Not defined	Axes involved in the cross coupling measurement.
axesLength	Int	0-127	Length of Axes array. Determines the number of involved axes.
crossCouplingType	FRF_CROSS_ COUPLING_TYPE	Complete CompleteOpen	Complete – measures the cross-coupling between all axes in closed loop. CompleteOpen – measures the cross- coupling between all axes in open loop.
excitationType	FRF_EXCITATION_ TYPE	WhiteNoise ChirpPeriodic UserDefined	Defines the excitation signal type White noise: generated for duration specified by durationSec and numberOfRepetitions. Signal is fully uncorrelated (not pseudo random noise). Standard deviation is determined by the excitationAmplitudePercentIp ChirpPeriodic: frequency range defined by the startFreqHz and endFreqHz is covered during the time period defined by the durationSec. The signal is repeated as defined in numberOfRepetitions. UserDefined: The signal is repeated as defined in

Name	Type	Default and Range	Description
			numberOfRepetitions.
chirpType	FRF_CHIRP_TYPE	LogarithmicChirp LinearChirp	Defines the type of Chirp signal. Applicable only if excitationType is ChirpPeriodic
windowType	FRF_WINDOW_TYPE	Hanning Hamming Rectangular	Defines the window type for filtering the signals before computing the FFT
overlap	FRF_OVERLAP	NoOverlap HalfSignal	Defines the amount of signals overlap for Welch averaging
Frequency DistributionType	FRF_FREQUENCY_DISTRIBUTION_TYPE	Logarithmic Linear	Describes distribution of points of the resulted Plant. Has two options Logarithmic: in this case user may specify startFreqHz, endFreqHz, freqPerDec, HighResolutionStart and highResolutionFreqPerDec. If HighResolutionStart falls between startFreqHz and endFreqHz and highResolutionFreqPerDec > freqPerDec then frequency range will have two regions with different frequency densities. Otherwise only the startFreqHz, endFreqHz, freqPerDec will determine the frequency range and density. Linear: frequency range is determined by the startFreqHz and endFreqHz, while frequency resolution is by duration

Name	Type	Default and Range	Description
			of the single measurement
startFreqHz	double	30, 1-5000	Defines start frequency of the resulting plant in Hz
endFreqHz	double	3000, 1-5000	Defines end frequency of the resulting plant in Hz
freqPerDec	int	50, 10-1000	Defines number of points per decade for standard resolution
highResolutionStart	double	500	Frequency where high resolution starts. The high resolution feature is active only if highResolutionStart is within the range from startFreqHz to endFreqHz.
highResolutionFreqPerDec	int	500	Defines number of frequencies per decade for high resolution range. The high resolution feature is active only if highResolutionStart is within the range from startFreqHz to endFreqHz.
excitationAmplitudePercentIp	double*	1, 1e-4 – 50	Excitation amplitude of the signal defined in % of the drive's peak current. It is defined for each axis involved in the cross coupling measurement. In the case of white noise this parameter determines the standard deviation of the noise.

Name	Type	Default and Range	Description
durationSec	double	5, 1e6-100	Duration of a single excitation in seconds. For periodic chirp it is the time that take to go through all frequencies once. This parameters ignored if selected excitationType = UserDefined.
numberOfRepetitions	int	1-100	Number of repetitions of the excitation signal. In case of excitationType = ChirpPeriodic or UserDefined the signal is repeated according to numberOfRepetitions value. In case of white noise overall excitation duration will be excitation duration multiplied by the numberOfRepetitions.

7.8.10 FRF_CROSS_COUPLING_OUTPUT

Description

Reads the results of measurement of cross-coupling effects for use in FRF calculations.

Name	Type	Description
plant	FRD*	Measured frequency response data (FRD) of the Plant matrix
controller	FRD*	Calculated FRD of the controller matrix. Calculation is based on standard servo parameters available in SERVO_PARAMETERS structure. Only the main diagonal has valid controller frequency response. Rest is euther zero or NULL.
characteristicPolynomial	FRD	Calculated FRD of the characteristic polynomial based on plant and controller FRD's
closedLoop	FRD*	Calculated FRD of the ClosedLoop matrix based on plant and controller FRD's.
sensitivity	FRD*	Measured FRD of the sensitivity matrix.
coherencePS	FRD*	Measured FRD of the coherence amplitude of the precess sensitivity matrix. Imaginary part of the coherence amplitude is 0
coherencePS	FRD*	Measured FRD of the coherence amplitude of the sensitivity matrix. Imaginary part of the coherence amplitude is 0
RGA	FRD*	Relative gain array matrix

Name	Type	Description
crossCouplingType	FRF_LOOP_TYPE	Determines the cross coupling measurement type
stabilityMargins	FRF_STABILITY_MARGINS*	Stability margins calculated based on characteristic polynomial frequency respose
excitationAmplitude	double*	Excitation amplitude used during measurement

8. Enums

This chapter details the enums that are available for SPiiPlus C programming.

8.1 ACSC_LOG_DETALIZATION_LEVEL

Description

This enum is used for setting log file parameters in the acsc_SetLogFileOptions function.

Syntax

```
typedef enum
{
 Minimum,
 Medium,
 Maximum,
} ACSC_LOG_DETALIZATION_LEVEL;
```

Arguments

Minimum	Value 0: Minumum information
Medium	Value 1: Medium information
Maximum	Value 2: Maximum information

8.2 ACSC_LOG_DATA_PRESENTATION

Description

This enum is used for setting log file parameters in the acsc_SetLogFileOptions function.

Syntax

```
typedef enum
{
 Compact,
 Formatted,
 Full
} ACSC_LOG_DATA_PRESENTATION;
```

Arguments

Compact	Value 0: No more than the first ten bytes of the data strings will be logged. Non-printing characters will be represented in Hex ASCII code.
Formatted	Value 1: All the binary data will be logged. Non-printing characters will be represented in Hex ASCII code.
Full	Value 2: All the binary data will be logged as is.

8.3 ACSC_APPSL_FILETYPE

Description

This enum is used by Application Load/Save functions for defining the application file type.

Syntax

```
typedef enum
{
 ACSC_ADJ,
 ACSC_SP,
 ACSC_ACSPL,
 ACSC_PAR,
 ACSC_USER
}  ACSC_APPSL_FILETYPE;
```

Arguments

ACSC_ADJ	Value 0: File type is an Adjuster.
ACSC_SP	Value 1: File type is an SP application.
ACSC_ACSPL	Value 2: File type is a Program Buffer.
ACSC_PAR	Value 3: File type is a Parameters file.
ACSC_USER	Value 4: File type is a User file

8.4 ACSC_CONNECTION_TYPE

Description

This enum is used for setting communication type. Used in the acsc_GetConnectionInfo function

Syntax

```
typedef enum
{
 ACSC_NOT_CONNECTED = 0,
 ACSC_SERIAL = 1,
 ACSC_PCI = 2,
 ACSC_ETHERNET = 3,
 ACSC_DIRECT = 4
} ACSC_CONNECTION_TYPE;
```

Arguments

ACSC_NOT_CONNECTED	Value 0: Not Connected
ACSC_SERIAL	Value 1: Serial Communication
ACSC_PCI	Value 2: PCI Communication
ACSC_ETHERNET	Value 3: Ethernet Communication

ACSC_DIRECT	Value 4: Direct (Simulator) Communication

8.5 FRF_LOOP_TYPE

Description

Enumerates supported FRF loop types.

Syntax

typedef enum {
PositionVelocity,
Position,
Velocity,
Current,
Open
} FRF_LOOP_TYPE;

Arguments

PositionVelocity	Measures Plant and Sensitivity of the PositionVelocity loop along with measurement coherence. Measurement is executed in closed-loop mode.
Position	Measures Plant and Sensitivity of the Position loop along with measurement coherence. Measurement is executed in closed-loop mode.
Velocity	Measures Plant and Sensitivity of the Velocity loop along with measurement coherence. Measurement is executed in closed-loop mode.
Current	Measures Plant of the Current loop along with measurement coherence. Measurement is executed in closed-loop mode.
Open	- Measures Plant of the PositionVelocity loop along with measurement coherence. Measurement is executed in open-loop mode.

8.6 FRF_EXCITATION_TYPE

Description

Enumerates supported methods for distributing frequency points.

Syntax

typedef enum
{
Linear,
Logarithmic
} FRF_EXCITATION_TYPE;

Arguments

Linear	frequency points distributed linearly in the range defined by the startFreqHz and endFreqHz. Frequency resolution is determined by **durationSec** parameter
Logarithmic	– points are distributed according to **startFreqHz**, **endFreqHz**, **freqPerDec**, **highResolutionStart** and **highResolutionFreqPerDec**

8.7 FRF_CHIRP_TYPE

Description

Enumerates types of chirp input supported.

Syntax

typedef enum
{
LogarithmicChirp,
LinearChirp
} FRF_CHIRP_TYPE;

Arguments

LogarithmicChirp	Chirp frequencies are distributed logarithmically in the time range. As a result, lower frequencies will have higher power per frequency.
LinearChirp	Chirp frequencies are distributed Linearly. As a result, lower all frequencies will have the same energy.

8.8 FRF_OVERLAP

Description

Enumerates the options for signal overlapping.

Syntax

typedef enum
{
NoOverlap,

```
HalfSignal,
}FRF_OVERLAP;
```

Arguments

NoOverlap	Signals are not overlapped
HalfSignal	Signals are overlapped at half-length for better averaging in frequency domain

8.9 FRF_CROSS_COUPLING_TYPE

Description

Determines whether measurement of cross coupling is in closed or open loop configuration.

Syntax

```
typedef enum
{
 Complete,
CompleteOpen
} FRF_CROSS_COUPLING_TYPE;
```

Arguments

Complete	Measures the cross coupling in the closed loop
CompleteOpen	Measures the cross coupling in the open loop

9. Sample Programs

The following samples demonstrate the usage of the SPiiPlus C Library functions. The examples show how to write the C/C++ applications that can communicate with the SPiiPlus controller.

The samples open the communication with the controller or the simulator and perform some simple tasks, like starting of a point-to-point motion, reading a motor feedback position, downloading an ACSPL+ program to the controller program buffer, etc.

After installation of the package in the SPiiPlus C Library directory, the full source code of these samples with the projects for Visual C++ 6 and Visual Studio 2005 can be found.

9.1 Reciprocated Motion

The following two samples execute a reciprocated point-to-point motion and read a motor feedback position. The first sample downloads the ACSPL+ program to the controller's program buffer and run it with help of the SPiiPlus C functions. The second sample uses the SPiiPlus C functions to execute motion.

Both of the samples are written as Win32 console applications.

9.1.1 ACSPL+

The sample shows how to open communication with the simulator or with the controller (via serial, ethernet or PCI Bus), how to download the ACSPL+ program to controller's buffer, and how to execute it. The ACSPL+ program executes a reciprocated point-to-point motion.

File ACSPL.CPP:

```
#include <conio.h>
#include <stdio.h>
#include "windows.h"
#include "C:\Program Files\ACS Motion Control\SPiiPlus 6.70\ACSC\C_
CPP\acsc.h"
HANDLE hComm;// communication handle
void ErrorsHandler(const char* ErrorMessage, BOOL fCloseComm)
{
printf (ErrorMessage);
printf ("press any key to exit.\n");
if (fCloseComm) acsc_CloseComm(hComm);
{
_getch();
};
int main(int argc, char *argv[])
{
double FPOS;
// ACSPL+ program which we download to controller's buffer
// The program performs a reciprocated motion from position 0 to 4000
// and then back
char* prog = " enable X \r\n\
St: \r\n\
ptp 0, 4000 \r\n\
ptp 0, 0 \r\n\
```

```
goto St \r\n\
stop \r\n";
printf ("ACS Motion Control Copyright (C) 2011. All Rights \
Reserved.\n");
printf ("Application executes reciprocated point-to-point motion\n");
/*****************************************************************/
// Open communication with simulator
printf ("Application opens communication with the simulator, \
downloads\n");
printf ("program to controller's and executes it using SPiiPlus C Library
\
functions\n\n");
printf ("Wait for opening of communication with the simulator...\n");
hComm = acsc_OpenCommSimulator();
if (hComm == ACSC_INVALID)
{
ErrorsHandler("error while opening communication.\n", FALSE);
return -1;
}
printf ("Communication with the simulator was established \
successfully!\n");
/*****************************************************************/
/*****************************************************************
// Example of opening communication with the controller via COM1
printf ("Application opens communication with the controller via \
COM1, downloads\n");
printf ("program to the controller and executes it using SPiiPlus C \
Library functions\n\n");
printf ("Wait for opening of communication with the \
controller...\n");
hComm = acsc_OpenCommSerial(1, 115200);
if (hComm == ACSC_INVALID)
{
ErrorsHandler("error while opening communication.\n", FALSE);
return -1;
}
printf ("Communication with the controller was established \
successfully!\n");
/*****************************************************************/
/*****************************************************************
// Example of opening communication with the controller via COM1
printf ("Application opens communication with the controller via \
COM1, downloads\n");
printf ("program to the controller and executes it using SPiiPlus C \
Library functions\n\n");
printf ("Wait for opening of communication with the \
controller...\n");
hComm = acsc_OpenCommSerial(1, 115200);
if (hComm == ACSC_INVALID)
{
```

```
ErrorsHandler("error while opening communication.\n", FALSE);
return -1;
}
printf ("Communication with the controller was established \
successfully!\n");
/*****************************************************************/
/*****************************************************************
// Example of opening communication with controller via Ethernet
printf ("Application opens communication with the controller via \
Ethernet, downloads\n");
printf ("program to the controller and executes it using SPiiPlus C \
Library functions\n\n");
printf ("Wait for opening of communication with the \
controller...\n");
// 10.0.0.100 - default IP address of the controller
// for the point-to-point connection to the controller
hComm = acsc_OpenCommEthernet("10.0.0.100", ACSC_SOCKET_DGRAM_PORT);
// for the connection to the controller via local network or Internet
//hComm = acsc_OpenCommEthernet("10.0.0.100", ACSC_SOCKET_STREAM_PORT);
if (hComm == ACSC_INVALID)
{
ErrorsHandler("error while opening communication.\n", FALSE);
return -1;
}
printf ("Communication with the controller was established \
successfully!\n");
/*****************************************************************/
/*****************************************************************
// Open communication with the controller via PCI bus
// (for the SPiiPlus PCI-8 series only)
printf ("Application opens communication with the controller and\n");
printf ("sends some commands to the controller using SPiiPlus C Library
\
functions\n\n");
printf ("Wait for opening of communication with the \
controller...\n");
hComm = acsc_OpenCommPCI(ACSC_NONE);
if (hComm == ACSC_INVALID)
{
ErrorsHandler("error while opening communication.\n", FALSE);
return -1;
}
printf ("Communication with the controller was established \
successfully!\n");
/*****************************************************************/
printf ("Press any key to run motion.\n");
printf ("Then press any key to exit.\n");
_getch();
// Stop a program in the buffer 0
if (!acsc_StopBuffer(hComm, 0, NULL))
```

```
{
ErrorsHandler("stop program error.\n", TRUE);
return -1;
}
// Download the new program to the controller's buffer
if (!acsc_LoadBuffer(hComm, 0, prog, strlen(prog), NULL))
{
ErrorsHandler("downloading program error.\n", TRUE);
return -1;
}
printf ("Program downloaded\n");
// Execute the program in the buffer 0
if (!acsc_RunBuffer(hComm, 0, NULL, NULL))
{
ErrorsHandler("run program error.\n", TRUE);
return -1;
}
printf ("Motion is in progress...\n");
printf ("Feedback position:\n");
while (!_kbhit())
{
// read the feedback position of axis 0
if (acsc_GetFPosition(hComm, ACSC_AXIS_0, &FPOS, NULL))
{
printf ("%f\r", FPOS);
}
Sleep(500);
}
// Stop the program in the buffer 0
if (!acsc_StopBuffer(hComm, 0, NULL))
{
ErrorsHandler("stop program error.\n", TRUE);
return -1;
}
// Close the communication
acsc_CloseComm(hComm);
return 0;
}
```

9.1.2 Immediate

The sample shows how to open communication with the Simulator or with the controller (via serial, ethernet or PCI Bus) and how to execute a reciprocated point-to-point motion only by calling appropriate SPiiPlus C functions without any ACSPL+ program.

File IMMEDIATE.CPP:

```
#include <conio.h>
#include <stdio.h>
#include "windows.h"
#include "C:\Program Files\ACS Motion Control\SPiiPlus 6.70\ACSC\C_
```

```
CPP\acsc.h"
HANDLE hComm;           // communication handle
void ErrorsHandler(const char* ErrorMessage, BOOL fCloseComm)
{
        printf (ErrorMessage);
        printf ("press any key to exit.\n");
        if (fCloseComm) acsc_CloseComm(hComm);
        _getch();
};
int main(int argc, char *argv[])
{
        double FPOS;
        int State;
        printf ("ACS Motion Control. Copyright (C) 2011. All Rights \
Reserved.\n");
        printf ("Application executes reciprocated point-to-point
motion\n");
        /*****************************************************************/
        // Open communication with the simulator
        printf ("Application opens communication with the simulator and\n");
        printf ("sends some commands to the simulator using SPiiPlus C Library \
functions\n\n");
        printf ("Wait for opening of communication with the simulator...\n");
        hComm = acsc_OpenCommSimulator();
        if (hComm == ACSC_INVALID)
        {
                ErrorsHandler("error while opening communication.\n", FALSE);
                return -1;
        }
        printf ("Communication with the simulator was established \
        successfully!\n");
        /*****************************************************************/
        /*****************************************************************
        // Example of opening communication with the controller via COM1
       printf ("Application opens communication with the controller via \
        serial link and\n");
        printf ("sends some commands to the controller using SPiiPlus C Library
        \functions\n\n");
        printf ("Wait for opening of communication with the \
        controller...\n");
       hComm = acsc_OpenCommSerial(1, 115200);
        if (hComm == ACSC_INVALID)
        {
                ErrorsHandler("error while opening communication.\n", FALSE);
                return -1;
        }
        printf ("Communication with the controller was established \
        successfully!\n");
        /*****************************************************************/
        /*****************************************************************
```

```
        // Example of opening communication with the controller via Ethernet
        printf ("Application opens communication with the controller via \
        ethernet and\n");
        printf ("sends some commands to the controller using SPiiPlus C Library
        \functions\n\n");
        printf ("Wait for opening of communication with the \
        controller...\n");
        // 10.0.0.100 - default IP address of the controller
        // for the point to point connection to the controller
        hComm = acsc_OpenCommEthernet("10.0.0.100", ACSC_SOCKET_DGRAM_PORT);
        // for the connection to the controller via local network or Internet
//      hComm = acsc_OpenCommEthernet("10.0.0.100", ACSC_SOCKET_STREAM_PORT);
        if (hComm == ACSC_INVALID)
        {
                ErrorsHandler("error while opening communication.\n", FALSE);
                return -1;
        }
        printf ("Communication with the controller was established \
        successfully!\n");
        /*****************************************************************/
        /*****************************************************************
        // Open communication with the controller via PCI bus
        // (for the SPiiPlus PCI-8 series only)
        printf ("Application opens communication with the controller and\n");
        printf ("sends some commands to the controller using SPiiPlus C Library
\
functions\n\n");
        printf ("Wait for opening of communication with the \
        controller...\n");
        hComm = acsc_OpenCommPCI(ACSC_NONE);
        if (hComm == ACSC_INVALID)
        {
                ErrorsHandler("error while opening communication.\n", FALSE);
                return -1;
        }
        printf ("Communication with the controller was established \
       successfully!\n");
        /*****************************************************************/
        printf ("Press any key to run motion.\n");
        printf ("Then press any key to exit.\n");
        _getch();
        // Enable the motor 0
        if (!acsc_Enable(hComm, ACSC_AXIS_0, NULL))
        {
                ErrorsHandler("transaction error.\n", TRUE);
                return -1;
        }
        printf ("Motor enabled\n");
while (!_kbhit())
        {
```

```
                // execute point-to-point motion to position 4000
                if (!acsc_ToPoint(hComm, 0, ACSC_AXIS_0, 4000, NULL))
                {
                        ErrorsHandler("PTP motion error.\n", TRUE);
                        return -1;
                }
                printf ("Moving to the position 4000...\n");
                // execute backward point-to-point motion to position 0
                if (!acsc_ToPoint(hComm, 0, ACSC_AXIS_0, 0, NULL))
                {
                        ErrorsHandler("PTP motion error.\n", TRUE);
                        return -1;
                }
                printf ("Moving back to the position 0...\n");
                // Check if both of motions finished
                do
                {
                        if (acsc_GetFPosition(hComm, ACSC_AXIS_0, &FPOS, NULL))
                        {
                                printf ("%f\r", FPOS);
                        }
                        // Read the motor 0 state. Fifth bit shows motion
                        // process
                        if (!acsc_GetMotorState(hComm, ACSC_AXIS_0, &State,NULL))
                       {
                        ErrorsHandler("get motor state error.\n", TRUE);
                                return -1;
                        }
                        Sleep(500);
                } while (State & ACSC_MST_MOVE);
        }
        acsc_CloseComm(hComm);
return 0;
}
```

5 HaTnufa St.
Yokne'am Illit 2066717
Israel
Tel: (+972) (4) 654 6440 Fax: (+972) (4) 654 6443