Integration Toolkit Guide

Velox

PN 167-754, Revision 3, July 13, 2018

FormFactor™

Important Notice

While the information contained herein is believed to be accurate as of the date hereof, no express or implied representations or warranties are made with respect to its accuracy or completeness. FormFactor, Inc., and its subsidiaries disclaim liability for any inaccuracies or omissions. All information is subject to change without notice.

Users are required to read and follow carefully all safety, compliance and use instructions. Users assume all loss and liability arising from the use of products in any manner not expressly authorized. The conditions and methods of use of products and information referred to herein are the entire responsibility of the user and, to the maximum extent permitted by applicable law, FormFactor, Inc., and its subsidiaries shall not be liable for any damages, losses, costs or expenses arising out of, or related to, the use thereof.

No license, express or implied, by estoppel or otherwise, under any intellectual property right is granted in connection herewith. Users shall take all actions required to avoid intellectual property infringement.

Contents

Welcome

Welcome to the *Velox Integration Toolkit*. The information provided in this guide describes the toolkit installation, implementation, communication process, functions, and Message Server interface, as well as migration from legacy tools.

NOTE

Only FormFactor supplied controllers are supported. Adding third-party software or drivers, or modifying the software installed by FormFactor may void your warranty

Notational Conventions

This manual uses the following conventions:

Syntax strings appear in this font:

```
vxit_get_command_number
```

NOTE

Note is used to indicate important information about the product that is not hazard related.

CAUTION

Caution is used to indicate the presence of a hazard which will or can cause minor personal injury or property damage if the warning is ignored.

WARNING

Warning is used to indicate the presence of a hazard which can cause substantial personal injury or property damage if the warning is ignored.

DANGER

Danger is used to indicate the presence of a hazard which will cause severe personal injury, death or substantial property damage if the warning is ignored.

For More Information

More information is available from these sources:

- *Velox User Guide* or online Help
- *Velox Remote Interface Guide*
- www.formfactor.com

Technical Support

For immediate sales support or customer service assistance, please contact the Sales Administration Department at 1-800-550-3279 or sales@cmicro.com. Requests for sales, service, and technical support information receive prompt response.

NOTE

When sending email for technical support, please include information about both the hardware and software, with a detailed description of the problem, including how to reproduce it.

To receive a faster solution, try to recreate the problem to provide us with an exact sequence of events. Please have the following information available, if possible:

- Name, version number, and file date of the application in use
- Drive information, including sizes, hard drive controller card brand, partition sizes and partitioning software
- Additional hardware such as specialty video cards, EMS boards, or turbo cards
- Memory-resident programs in use when the problem occurred. Problems can occur when memory-resident programs are not loaded in the correct order
- Exact wording of any error messages
- Notes about any steps you took in trying to solve the problem
- Windows version number and manufacturer

Chapter

1 Getting Started

The Velox Integration Toolkit enables custom applications to communicate with Velox software running on the same or on a remote Windows-based PC. This document is intended for the third-party programmer who is creating such a custom application.

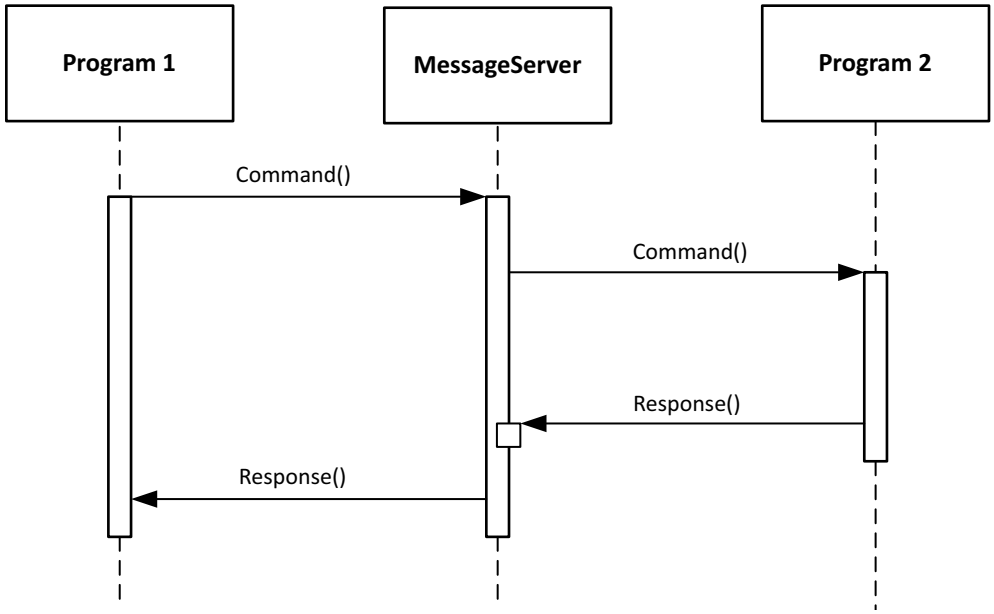
Whether the custom application runs on a separate computer or uses IPC on a single computer, it will use the same text-string-based, command-and-response communication scheme. To exchange command and response messages with Velox software via IPC, the custom application will call C-language functions that are exported by the central VxIntegrationToolkit.dll.

Communication Process

The Velox communication process is based on a command-and-response scheme. The process occurs in two phases:

- In the first phase, a command message is sent from an application to the Message Server.
- In the second phase, a response message is sent back from the server to the originating application.

For every command message, there is a corresponding response message. This command message combined with the response message forms a conceptual unit called a command.



If an application does not send a response within a defined timeout, the Message Server will send an error response indicating a timeout error.

A command message consists of a list of parameters and a unique command identifier that implicitly specifies the meaning and type of each command parameter. A response message also consists of a list of parameters, and its associated command identifier specifies the meaning and type of each response parameter.

The caller of a command also defines a message ID in the command message. The response message will contain the same ID and matches command and response for multiple commands handled in parallel.

A notification is a special kind of a command that always generates the message ID 0. Every application receives the command message and no response message is expected.

The Message Server is the central manager for every application. The Message Server also dispatches the commands and notifications.

Command Definitions

The commands are defined in multiple files:

commands.txt	Public commands
commands.user.txt	User-defined commands
commands.internal.txt	Internal, undocumented commands which should not be used by third-party applications

The commands in commands.txt are documented in the *Velox Communications Guide*. Custom commands should be written to *commands.user.txt*. Commands.txt and commands.internal.txt must not be modified. These files will be overwritten on every installation of Velox. The Message Server merges these three files at startup. The following discussion describes this merged result.

Entries in the commands.txt have the following format:

[Group Name]

```
[Group Name]
<CmdNum> <TimeOut> <InputParams> <OutputParams> <CmdText> <DefaultCmdParams>
```

Each item represents the following information:

CmdNum	Command number in hexadecimal format
TimeOut	Timeout for this command in milliseconds
InputParams	String of characters representing input parameters types
OutputParams	String of characters representing output parameters types
CmdText	ASCII text name for this Command
DefaultCmdParameters	String of default input parameters for this command (optional)

Here is an example entry from commands.txt: `31 10000 CCC DDD ReadChuckPosition Y Z`

Command number	0x31
Command name	ReadChuckPosition
Command parameter list	CCC - 3 characters specifying unit, reference and compensation mode
Response parameter list	DDD - 3 floating point values indicating X, Y and Z

Parameter lists for the public commands are documented in the *Velox Communications Guide*. To see the parameters for ReadChuckPosition in the Communications Guide, select:

/Contents/Kernel Commands/Chuck Commands/ReadChuckPosition

Command and response parameter lists are passed as text strings, in which the individual parameters are separated by white space (usually a single-space character). Typically, the application will need to:

- Compose a command parameter string from native language data types
- Parse a response parameter string into native language data types

Command Groups

Velox command types are grouped according to the target of the command. The target for a group can be the Kernel, the Message Server, or an application that has been specifically designed to accept commands.

A third-party application can add commands to commands.user.txt. The ID must be unique and must be 0x10000 or higher to avoid collisions with built-in Velox commands.

Logging

Velox provides List logging and File logging, executed by MsgServer.exe. MsgServer.exe is a tray icon application. Double-click on the icon to make MsgServer.exe visible. MsgServer.exe logs all of the command-and-response messages which pass through the Server. This is useful for debugging the communication between applications. For each message, MsgServer.exe logs the command source, command target, message type (command, response, or notification), timestamp and command ID.

The MsgServer.exe GUI provides an easy, intuitive logging configuration. The following logging modes can be configured:

None	No logging is performed
List	List all commands and their responses in the GUI
File	Write all commands and their responses to a log file
List&File	List all commands and their responses in the GUI and write them to a log file

The MsgServer performs logfile rotation: if the logfile MessageServer.log reaches a predefined size (10MB), it will be renamed to MessageServer_1.log and a new MessageServer.log is created. The former MessageServer_1.log will be renamed to MessageServer_2.log and so on up to MessageServer_4.log.

Error Codes

When a command fails, it returns a nonzero error code. A complete list of errors is kept in the Errors.txt file, which contains a translation table of error codes to error messages in the following format:

```
<ErrorNumber>:<ErrorMessage>
```

ErrorNumber	Decimal error number
ErrorMessage	Textual description of the error

Errors.txt can be extended with custom errors to allow descriptive error messages for user-defined commands. When Errors.txt is edited, the Message Server must be restarted to take effect.

Installation

Note

Before installing the toolkit, MATLAB and LabVIEW applications must be closed.

To install the Velox Integration Toolkit:

1. Start the setup.exe from the CD and follow the instructions of the installer.
2. Select the custom option to choose which components should be installed.

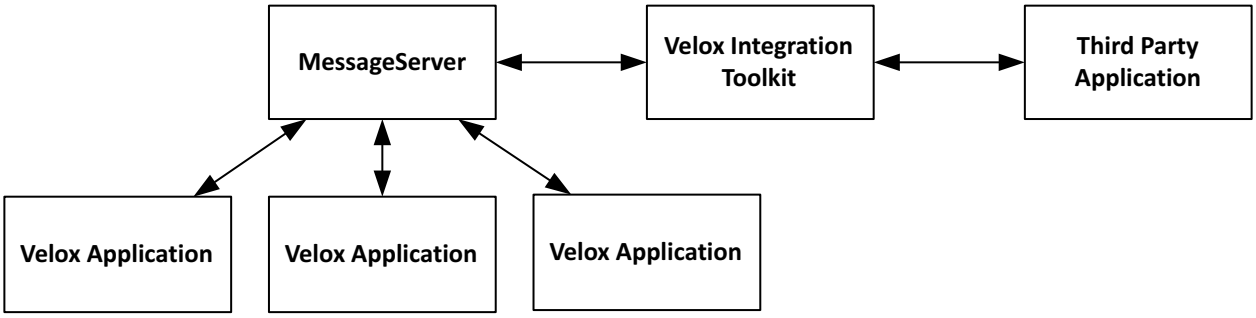
Use the Start Menu under Cascade Microtech\\VeloxIntegrationToolkit to access the installed files.

- All files except LabVIEW are installed into the default folder C:\Program Files (x86)\Cascade Microtech\VeloxIntegrationToolkit. For more information on LabVIEW, see LabVIEW on page 23.
- All provided files under the installed \\VeloxIntegrationToolkit\\Examples folder as well as the VI's of the LabVIEW \\CascadeProbestation\\Examples folder can be modified for user application.
- If MATLAB is installed, the installer automatically updates the matlabrc.m file and inserts the path variables for the Velox Integration Toolkit library .dll files. For more information on MATLAB, see MATLAB on page 25.
- All examples and .dll files of the Velox Integration Toolkit are available on the installation CD.

Chapter

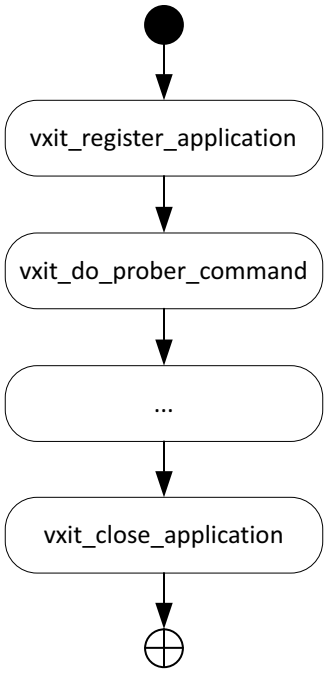
2 Implementation

The Velox Integration Toolkit operates as a bridge between the Message Server and the third-party application. It provides a simple API, which can be used for communication. It is possible, but not recommended, to communicate with the Message Server directly. See Message Server Interface for details.



The general workflow is always:

1. Register application.
2. Execute commands, handle responses, etc.
3. Close application.



A minimal example in C without proper error handling:

```
char responseBuffer[4096];
vxit_register_application("TestApplication", "TestApplication", 0, 0);
vxit_do_prober_command("ReportKernelVersion K", responseBuffer);
vxit_close_application();
```

The application must always be closed at the end. Otherwise, the application can't shut down properly.

In C/C++, you can compose parameters using the standard library function sprintf(). For instance, this code fragment composes a command parameter string for the MoveChuckIndex command (0x35). It will move the chuck 2 index steps in X and 3 index steps in Y, relative to the current position, at 50% of full speed:

```
char cmd[64]; // leave ample room for command param
int indexStepsX=2, indexStepsY=3;
sprintf(cmd, "MoveChuckIndex %i %i R %.3f", indexStepsX, indexStepsY, 50.0);
vxit_do_prober_command(cmd);
```

Parsing in C/C++ can be accomplished using the standard library function sscanf(). For instance, this code fragment will parse the response from a ReadChuckPosition command into three floating point variables:

```
char rsp[4096]; // leave ample room for response string
double x, y, z;
vxit_do_prober_command("ReadChuckPosition Y C", rsp);
sscanf(rsp, "%lf %lf %lf", &x, &y, &z);
```

Note that commands can always fail (for example, when the application in question isn't started). Refer to the examples provided for proper error handling.

32-bit and 64-bit Applications

The Velox Integration Toolkit supports 32-bit and 64-bit applications. The signature of the functions are the same for both architectures. The library names are:

- VxIntegrationToolkit.dll for 32-bit
- VxIntegrationToolkit64.dll for 64-bit

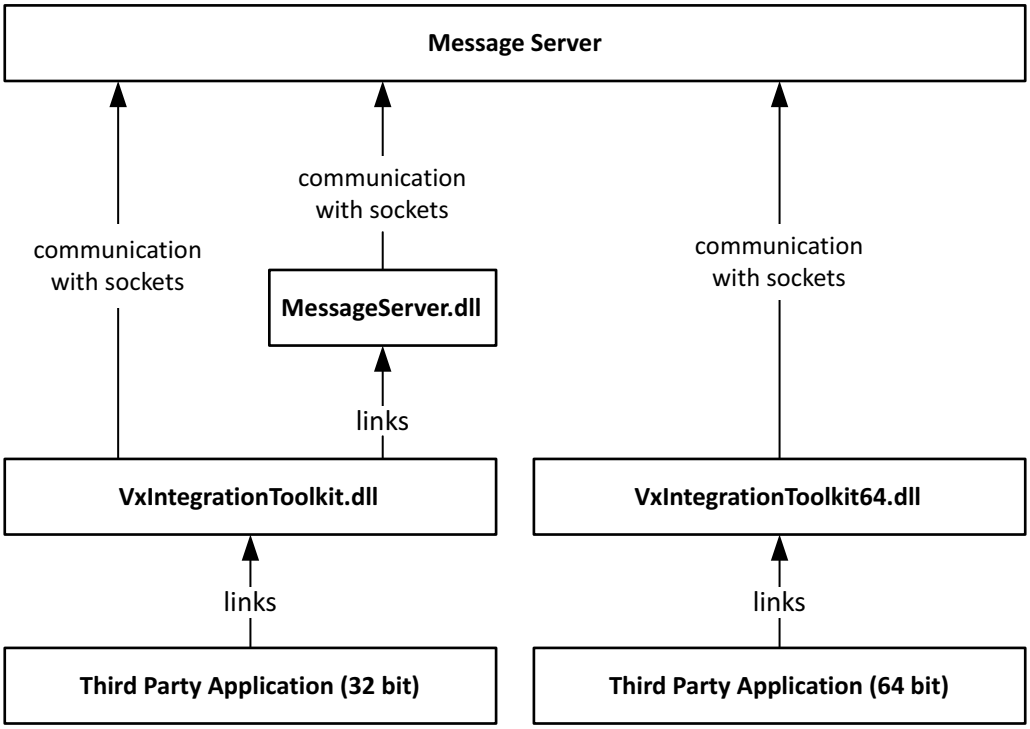
The name and behavior of the functions is the same for both.

Connection to Message Server

The Velox Integration Toolkit can connect to the Message Server either by using a Velox internal library (32-bit only) or communicating directly with sockets (32-bit and 64-bit).

- When vxit_register_application is called, the communication will be handled by the internal library.
- When vxit_register_application_socket is called, the Message Server communicates directly with the sockets.

All methods behave the same in both modes.



Internal library communication

Using the internal message-server-implementation (using vxit_register_application()) for communication is only supported on 32-bit. The internal library Vx.MessageServe.dll must be in the same directory as VeloxIntegrationToolkit.dll and the final executable. Also, a correctly-configured Velox installation is required. The configuration of Velox will be reused and no manual configuration of host and ports is required.

Communicating via socket

The communication via socket is supported on both architectures. No additional libraries are required, but a valid host and port must be entered. To connect to a locally-running instance of Message Server, enter localhost as host and 1412 as port (this is the default configuration and can be altered in the Velox Setup tool).

Chapter

3 Library Functions

The Integration Toolkit interface is encapsulated by the functions exported from the VxIntegrationToolkit.dll. All functions implement the following conventions:

- A function returns an integer value.
- A negative return value indicates an error in using the API (for example, passing unexpected NULL-pointers).
- All functions use the stdcall convention.

Function Descriptions

vxit_register_application

This function can be called only by 32-bit applications. On 64-bit, this function will return an error.

Signature		int_stdcall vxit_register_application(const char *appName, const char *section, int singleInstance, int handleNotifications)
Input	appName	Application name
	section	Section of commands.txt used
	singleInstance	If true, only one instance of this program can be registered at a time. If this flag is set and a second application with the same name tries to register, this function will return a negative error code.
	handleNotifications	If true, this application will receive notifications. If not, notifications can be polled with vxit_get_command.)
Output		None
Return		Zero on success, a negative error code on failure

Any application that will communicate with the Message Server via the Integration Toolkit must first call this function to register itself before calling any other function of the API. Every application must also call vxit_close_application on the cleanup process.

The application name is used to identify the program in the Message Server log. Also, other applications can test if the application is set up with the remote command IsAppRegistered.

The commands handled by this application are defined by the commands.txt and selected with the section.

To change the name and/or the section of an already registered application, vxit_close_application must be called first.

Note

If the application should not handle notifications, the flag handleNotificaions must not be set. Otherwise, the notifications can be polled with vxit_get_command.

To change the name and/or the section of an already registered application, vxit_close_application must be called first.

vxit_register_application_socket

This function can be called by both 32-bit and 64-bit applications.

Signature		int_stdcall vxit_register_application_socket(const char *appName, const char *section, int singleInstance, int handleNotifications, const char *host, int port);
Input	appName	Application name
	section	Section of commands.txt used
	singleInstance	If true, only one instance of this program can be registered at a time
	handleNotifications	If true, this application will receive notifications
Output		None
Return		Zero on success, a negative error code on failure

This method behaves the same as vxit_register_application, but communication is directly with sockets and not using the internal library. The port default is 1412 for Velox but can be configured by the Velox Setup tool of the Velox software suite.

vxit_close_application

Signature	int_stdcall vxit_close_application();
Input	None
Output	None
Return	Return 0 on success

This method ends the application cycle and must be called after vxit_register_application and before application termination. An application can close and reopen communication with the Message Server multiple times.

vxit_do_prober_command

Signature		int_stdcall vxit_do_prober_command(const char *commandString, char *responseString);
Input	commandString	Command to execute
Output	responseString	Response to the command or an error description
Return		Zero on success, a positive error code or a negative number indicating an error in use

This function provides the simplest interface to the IntegrationToolkit. It implements synchronous command dispatch: the calling application will block until the response message becomes available.

The CommandString input parameter is the command name and its parameter, separated by a space.

The ResponseString output parameter length must accommodate the response parameter list string. If the command results in an error, ResponseString will contain a description of the error. As the error description can be much longer than the expected result, use a large buffer even if the expected answer is short. A buffer length of 512 should be considered as a minimum, 4096 is recommended.

The return code is defined by the called command, but in general, a value of zero indicates success, a value greater than zero indicates a specific error code of the called application, and a value less than zero indicates an error using the API.

vxit_send_command

Signature		int_stdcall vxit_send_command(const char *commandString);
Input	commandString	Command to execute
Output		None
Return		A positive message ID on success

This method is the first part of an asynchronous communication. The command is sent to the Message Server, then the function returns immediately. The response to this command can be received with vxit_get_response.

vxit_get_response

Signature		int_stdcall vxit_get_response(int *errorCode, char *responseString);
Input		None
Output	error code	Zero on success or a positive error code
	responseString	The response of the command or an error description
Return		A positive message ID or zero, if there are no pending commands

This function gets a pending response message from the Message Server, if one is available. The MessageID return value allows the application to match the response with a previously-dispatched command. The ResponseString output parameter must have enough length to accommodate the response parameter list string. As with vxit_do_prober_command, a buffer size of 512-4096 bytes is recommended. The ErrorCode returns the status of the command, where zero indicates success.

This function can be used in blocking or nonblocking mode. See vxit_set_block_mode for details.

vxit_get_command

Signature		int_stdcall vxit_get_command(int* messageId, char *commandString);
Input		None
Output	messageId	ID used send response to this command, will be zero for notifications
	parameterString	The parameters of this command
Return		The command- or notifcation-ID or zero if no commands are pending

This function is used to implement command handler or notification handler. The message ID is used to combine a command with the correct response to send with vxit_send_response. The application should respond within the defined timeout. Otherwise, the caller will see a timeout error message. The message ID will be zero for notifications as no response is expected. The parameterString does not include the command name but only the send parameter.

The called command or notification is given by the numeric return value. The mapping from the command name as string to the command number is defined by the commands.txt. To query this numeric value programmatically, vxit_get_command_number can be called.

This function can be used in blocking or nonblocking mode. See vxit_set_block_mode for details.

vxit_send_response

Signature		int_stdcall vxit_send_response(int messageId, int errorCode, const char *responseString);
Input	messageId	Message ID of the command
	errorCode	Error code of the command, should be zero on success
	responseString	Response to the command
Output		None
Return		Return 0 on success

After an application handles a command received from vxit_get_command, it must send a response message by calling this function.

- For the messageID parameter, the application should pass the same value that was received from the originating vxit_get_command call.
- For responseString, the application should pass the response parameter list string.
- For the errorCode parameter, the application should pass zero to indicate success, or an appropriate error code to indicate failure.

vxit_get_error_description

Signature		int_stdcall vxit_send_command(const char *commandString);
Input	errorCode	Error number to get description for
Output		The description of the error
Return		Return 0 on success

This method is used to query a description to an error code. On regular error codes greater than zero, a more detailed description will be present in the response string. On errors using the API with values less than zero, this method can be used to get a description as a debugging aid.

vxit_get_command_number

Signature		int_stdcall vxit_send_command(const char *commandString);
Input	errorCode	The command in question
Output		None
Return		A value greater than zero on success

The command vxit_get_command relies on the command number; all other commands rely on the command name. This function is used to translate the command name to the command number. The translation into the other way is spared out intentionally. If the command number is unknown, a value less than zero will be returned.

vxit_set_block_mode

Signature		int_stdcall vxit_set_block_mode(vxit_block_mode_t blockmode);
Input	blockmode	The blocking mode to use
Output		None
Return		Zero on success

vxit_get_response and vxit_get_command can be used in different blocking modes:

VXIT_BLOCK_MODE_NONE (numeric 0)	Both functions do not block
VXIT_BLOCK_MODE_CMD (numeric 1)	vxit_get_command blocks, vxit_get_response does not block
VXIT_BLOCK_MODE_RSP (numeric 2)	vxit_get_command does not block, vxit_get_response blocks
VXIT_BLOCK_MODE_BOTH (numeric 3)	Both commands block
VXIT_BLOCK_MODE_TOGETHER (numeric 4)	Both commands block. When either a command or a response is received, both commands will unblock

With the last mode, a command handler and a response handler can be implemented into a single thread. The C#/VB/C++-examples uses this approach. The block mode can be set multiple times at run time.

When vxit_close_application() is called, vxit_get_command() and vxit_get_response() will return zero immediately.

Chapter

4 Applications

C#, C++, Visual Basic

The examples provided are the same for C#, C++, and Visual Basic. The wrapper code resembles the same interface. For this reason, these three languages are discussed in the same section.

The examples are included on the installation CD. See descriptions below.

1 - Synchronous Commands
2 - Asynchronous Commands
3 - Notifications
4 - Custom Commands
5 - Wafer Map Stepping
6 - Tester Interface Example
7 - Tester Control

All examples are created in Visual Studio 2013. Visual Basic and C# are configured for the DotNet-Framework 4.5.

Wrapper Code

The API of the IntegrationToolkit consists of typical C functions. To use this functionality in object-oriented code, it is wrapped in a class, and is simplified that way. This class is provided in source. It can be modified on demand. Applications implemented in C#, C++, and Visual Basic will use this wrapper code. The C-API can also be used directly.

The Server class is the access point to communicate with the Message Server. The underlying library functions should not be called when this implementation is used. The class is implemented as a singleton. This means it is not possible to create an instance of this class via a call to a constructor. An Instance is created on startup and is accessible by the static (VB: shared) method Instance(). All methods are safe to be called in a threaded environment. The meaning of the methods:

Register	Start the communication with the Message Server. This must be called before sending any commands.
RegisterSocket	Start the communication with the Message Server via sockets. This must be called before sending any commands.
Close	End the communication with the Message Server. This method must be called at the end of the program. Otherwise, this may stop the shutdown of the application by still running threads.
SendCommand	The meaning of this method is similar to vxit_do_prober_command but the implementation is different. See the diagrams in this section for details.
SendCommandAsync	Send a command and return immediately. When the command is executed, the provided callback is called. This callback is either a function pointer (C++) or a delegate (C#, VB). The third parameter can be used to pass an additional pointer and is only provided in C++.

SetHandler	This method is used to install command handler for predefined commands, custom commands, or notifications. If a command is received, the given command handler is called. There can only be one command handler at a time for any command or notification. Setting the callback of a command to NULL/null/Nothing deletes the installed command handler.
SendResponse	Received commands should be answered in time using this method. Otherwise, the caller of the command will receive a timeout. Notifications do not need a response. A command can be answered inside the provided callback when it is installed with SetHandler or at any later time.
ThrowExceptionOnError	This method influences the behavior of the SendCommand method. If true, the Server will throw an exception if a command returns with a nonzero return code. If false, SendCommand will return normally and the returned Command object will have the nonzero value in the ErrorCode attribute. This allows try-catch-based programming. The default is false.
Instance	A static/ shared method to access the instance of the singleton. Shared access across multiple threads is possible.

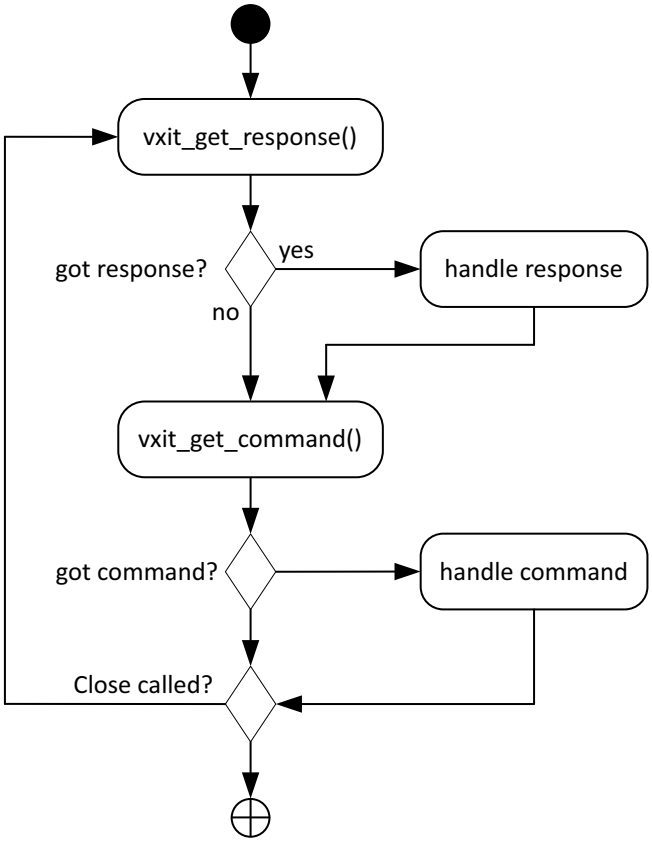
The wrapper code lets you:

- use a common interface for the language of choice
- use exceptions for error handling
- use delegates/function pointer for asynchronous commands and command handling

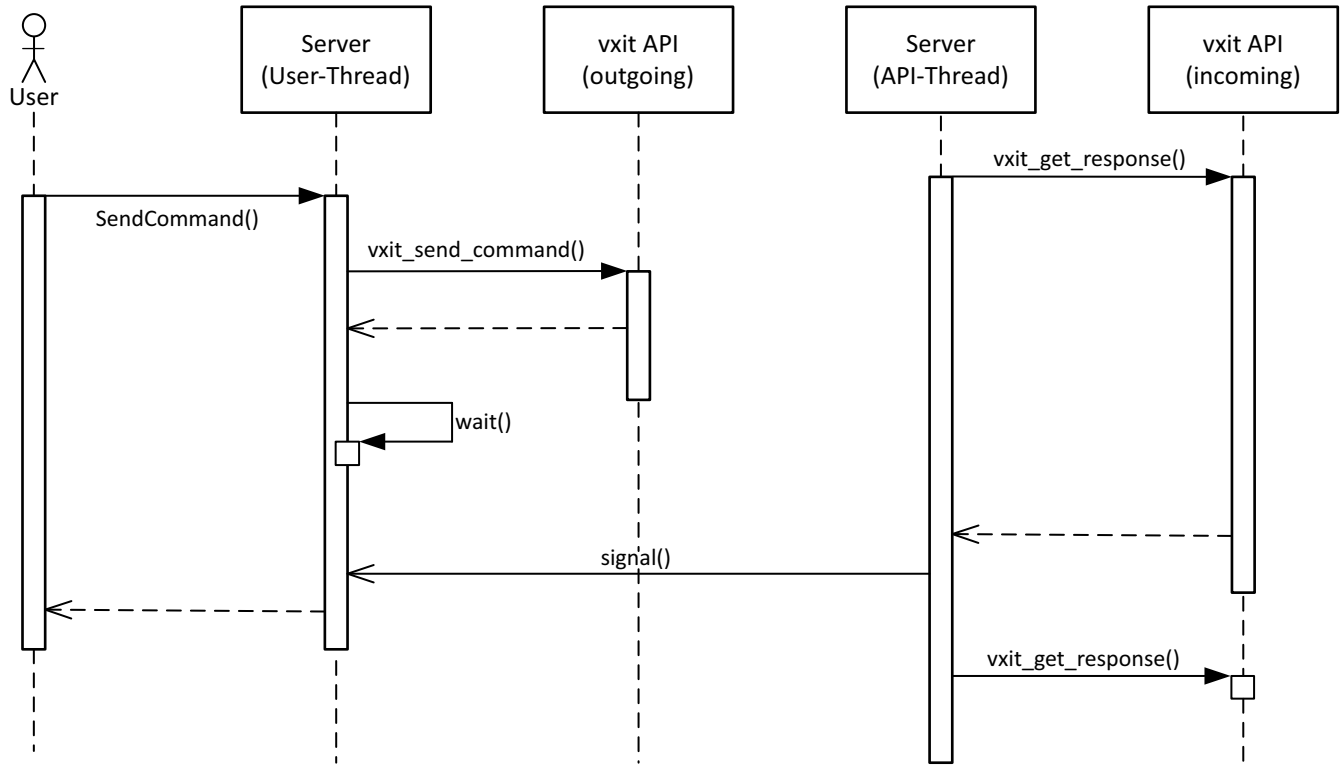
The primary difference in implementations is:

- C++ implementation uses the Win-API and STL
- C#/VB implementation uses DotNet

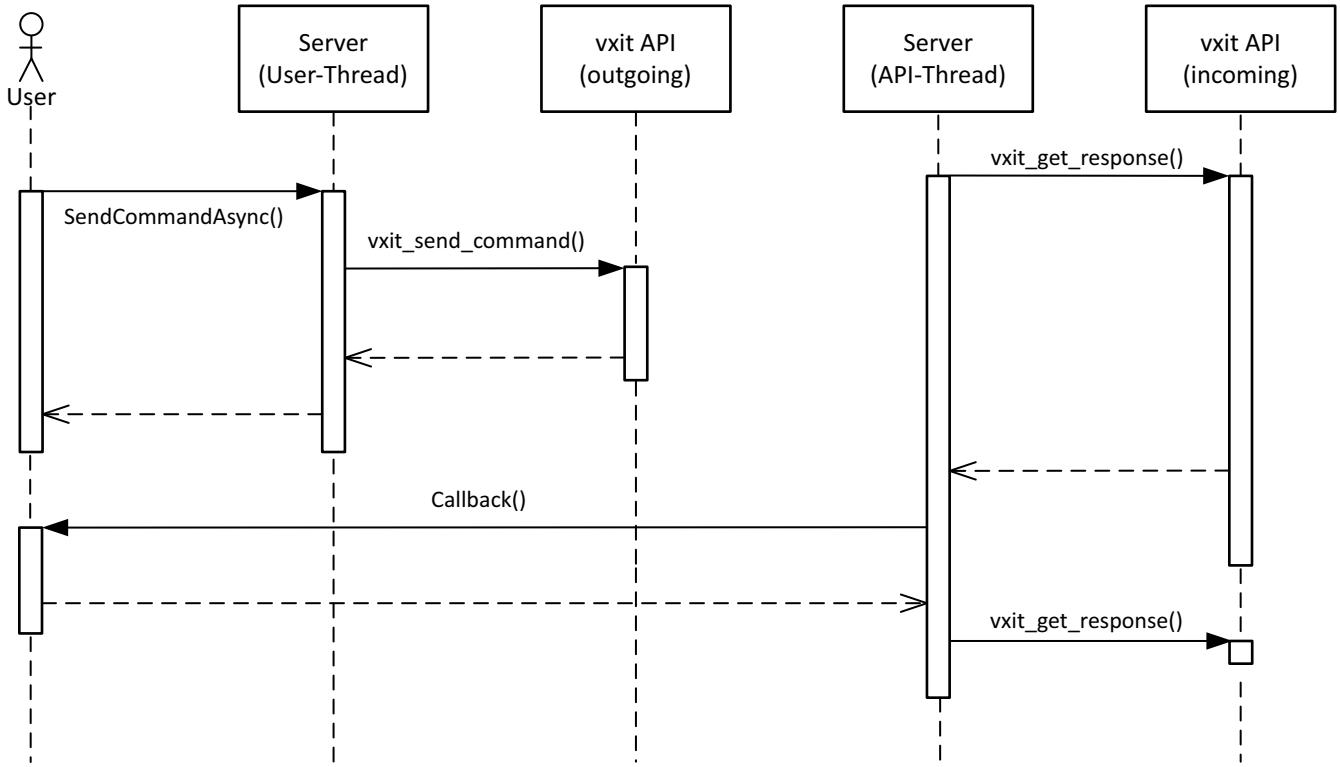
All three follow the same program flow.



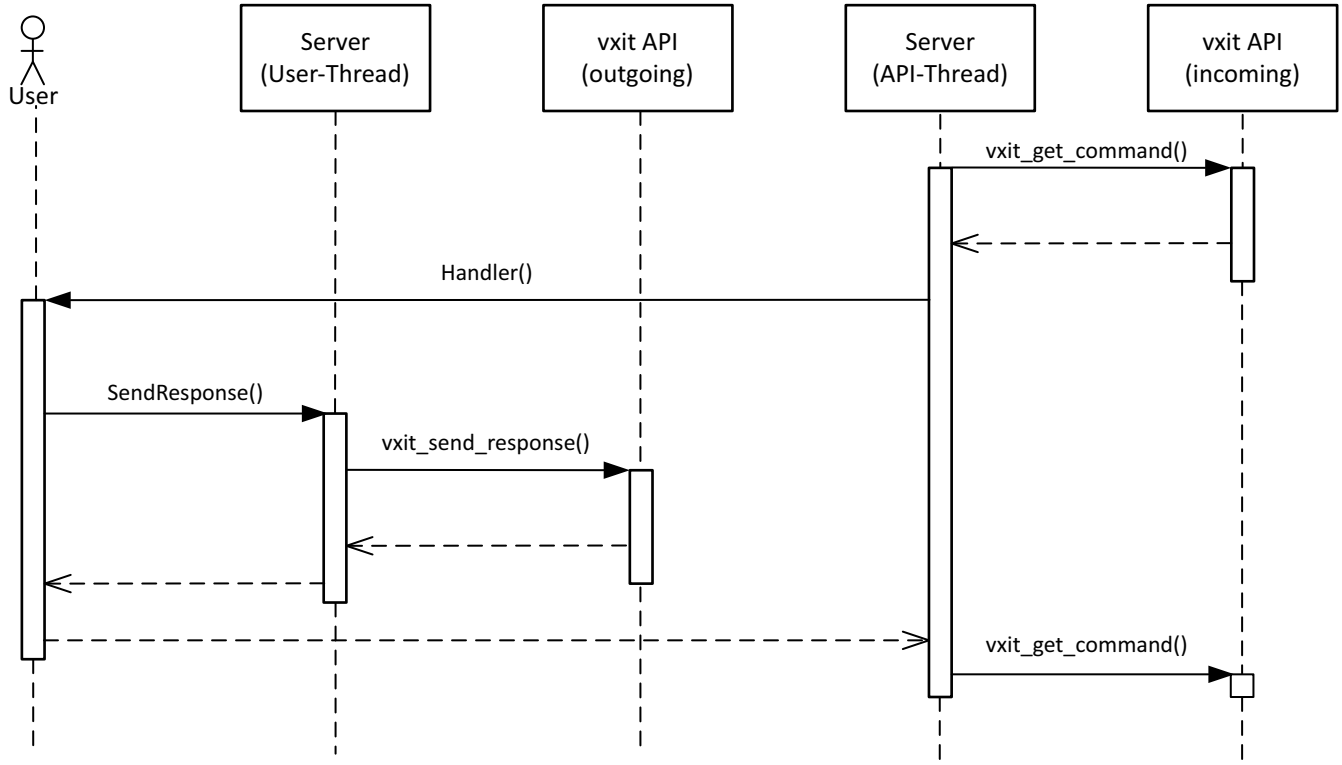
By calling the Register method, the internal server thread will be started. This thread will repeatedly call vxit_get_response() and vxit_get_command in sequence. This is done with block mode VXIT_BLOCK_MODE_TOGETHER. If either a command or a response is received, both methods will unblock. One of them will receive the event and handle it. The other one will skip the handling.



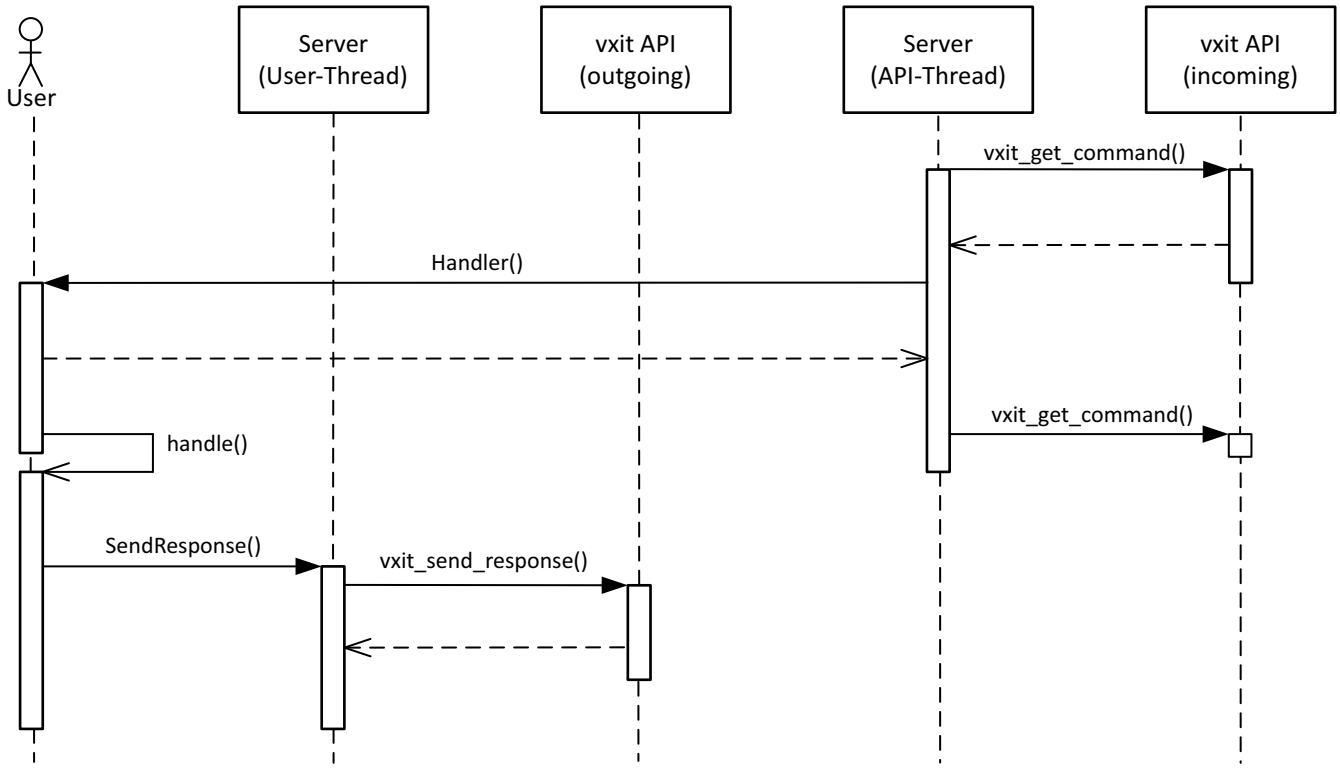
SendCommand doesn't use vxit_do_prober_command. It uses vxit_send_command and blocks until the correct response is received. This change is transparent for the user of the provided Server interface. There can be multiple calls in parallel to SendCommand invoked by multiple threads at any time.



Calls to SendCommandAsync return immediately. As soon as a response is received, the callback is invoked. This callback is invoked within the server thread. As long as this callback lasts, no further commands or responses can be processed. Events or other threading primitives should be considered for any time-consuming operation.



When a command is received where a handler is installed, the handler will be invoked. It is possible to send the response within the command handler. As this operation runs inside the server thread, no further commands or responses can be processed as long as the command handling lasts.



A response can also be sent after the callback has returned. This way, further commands or responses can be processed while the command is handled. This also includes the currently-handled command; the handler can be invoked before the former call is returned.

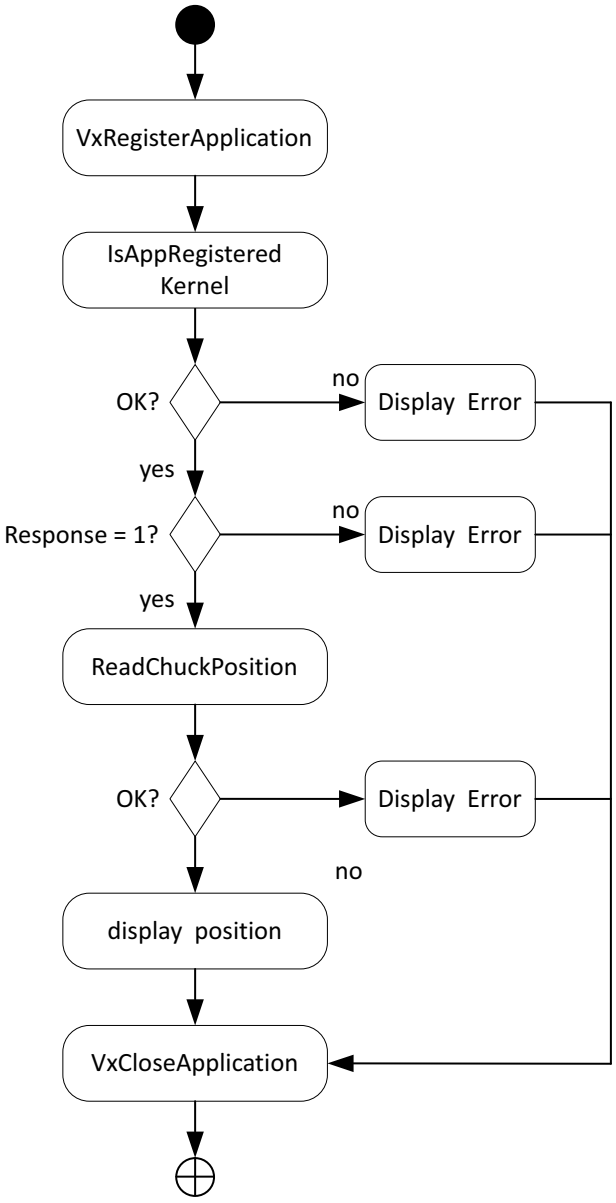
Synchronous Commands

The first example compares the direct use of the VxIntegrationToolkit.dll with the use of the wrapper code. The same functionality is implemented twice.

- Always call vxit_register_application after vxit_close_application.
- Every command has a return code. This return code should be 0 on success.
- Some commands have a return value. The format of this return value is defined by the command.

  For example:

  – IsAppRegistered always returns a single integer: 0 or 1
  – ReadChuckPosition always returns 3 floating point values separated by a space.



Asynchronous Commands

The second example shows how commands can be sent asynchronously. A delegate/callback is passed to the wrapper code along with the command to execute. The function returns immediately. When the command is finished and the response is received, the delegate/callback is called. In this example, the chuck moves to 5 different positions. The next step is triggered in the delegate/callback of the former one.

Notifications

The wrapper code simplifies the use of Notifications. Two handlers for notifications are installed in the beginning: RegisterProberAppChange and AlertNotification (KernelnotifiCations). After that, the main loop waits for user interaction to close the application. Whenever a program is opened or closed, or the chuck is moved, the corresponding callback is called and the information is printed onto the screen.

Custom Commands

This example demonstrates the implementation of custom commands and their handling. A custom command TestAppHello returns Hello <Input>. Except when the input is “John Doe.” Then, an error message will be returned.

Wafer Map Stepping

This example shows a typical use case for the Velox Integration Toolkit. Step through a configured Wafer Map and perform the measurement. In C# it is implemented in the following variations:

- Simple stepping includes Wafer Maps with subdies.
- For temperature handling, set custom temperature before stepping.
- For temperature loop, cycle through multiple temperatures before stepping.

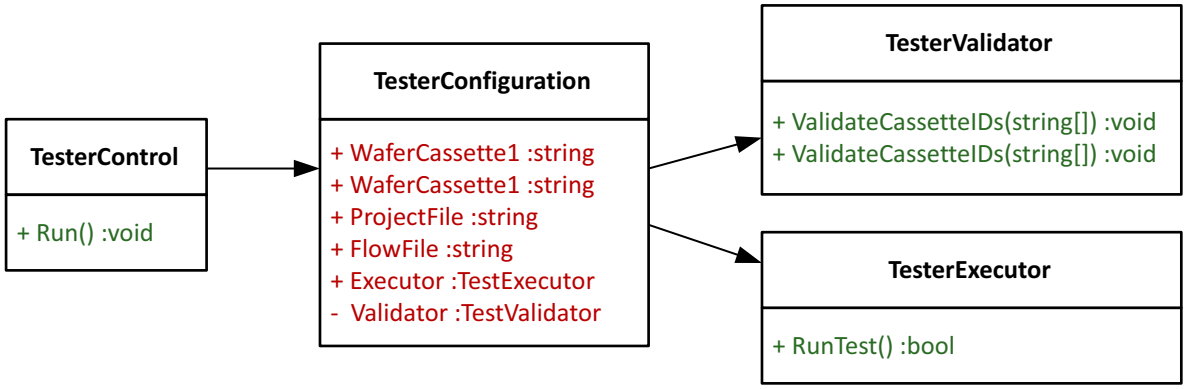
The required applications must be opened (for example, Wafer Map for stepping or Spectrum for alignment) and the system must be configured accordingly (for example, the system must have a ThermoChuck to allow temerature handling).

Tester Interface Example

This example consists of a basic implementation of the tester interface. This example can be used to implement a custom tester compatible with the VeloxPro Tester Interface. For more details, see Tester Interface for details.

Tester Control

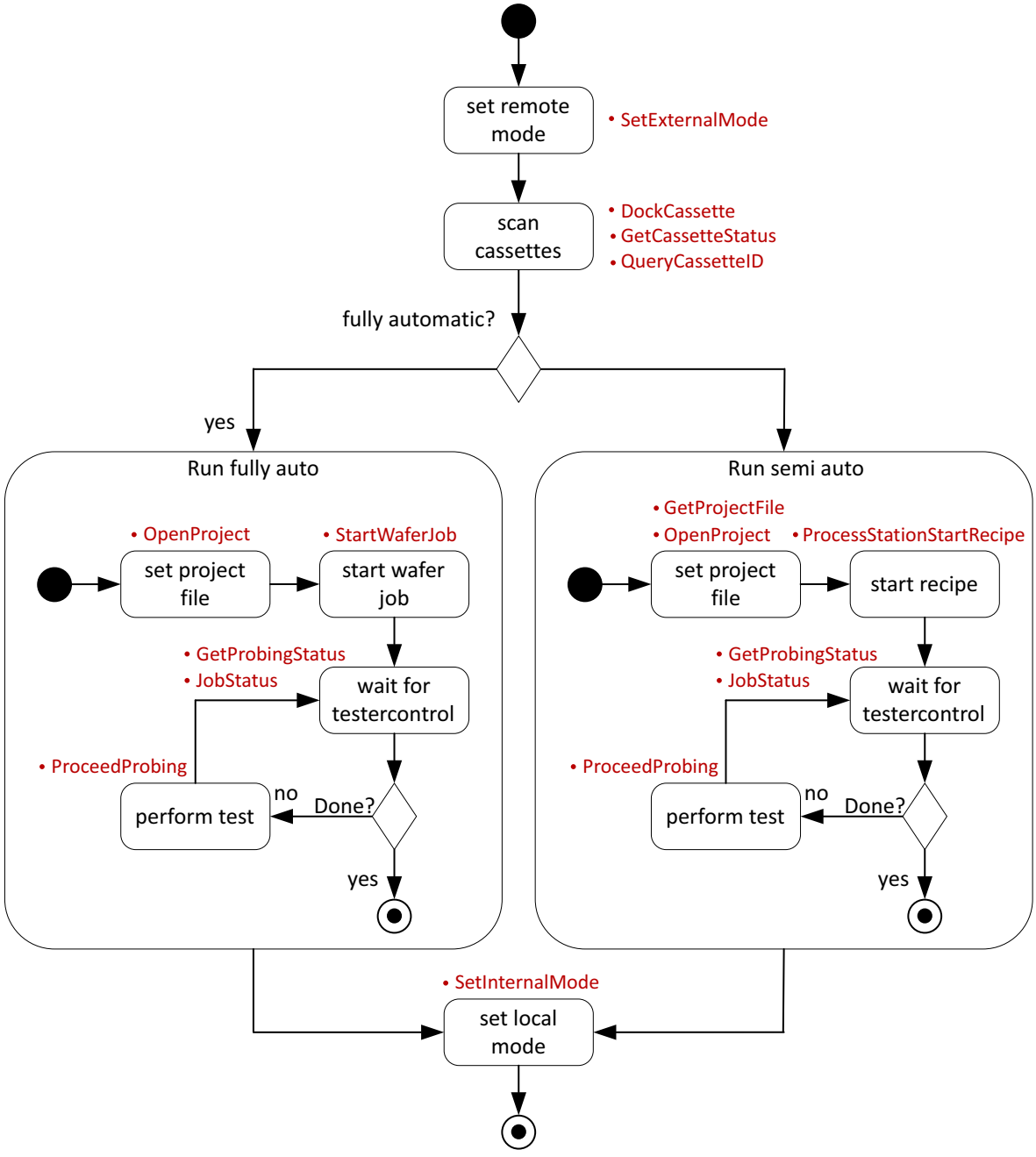
This example demonstrates how to control VeloxPro as master. Handling, which is normally done by VeloxPro, is now done in the application. This example is more complicated than other examples.



The TesterConfiguration class is used for simple setup of the test. This includes references to objects of the classes:

TestExecutor	Executed once per wafer to run the test. If the method returns false, testing will be aborted.
TestValidator	Executed in the setup phase. Cassette IDs and Wafer IDs can be verified.

Classes derived from the two above can be used to configure the desired behavior. The execution of the test is performed in multiple steps with simplifications in error handling:



Tester Control Implementation

This implementation can be used as a template for custom implementations.

NOTE

The SimpleTestExecutor class, near the main function in the examples, implements the TestExecutor interface. The RunTest method just returns true. A real implementation is similar to the Wafer Stepping example. Wafer operations such as chuck heating or alignment should be done using this method.

The basic functionality of the Tester Control Implementation is similar to this pseudo code snippet:

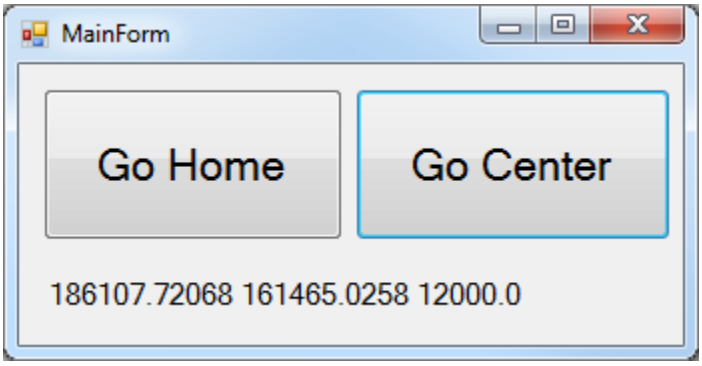
```
def TesterControl():
   SetExternalMode R
   DockCassette 1 1

   # get information about each present wafer
   # format:
   # <Cassette> <Slot> <Status> <IdStat> <WaferId>; <Cassette> <Slot> …
   # status can be Empty, Present, Testing
   cassetteStatus = GetCassetteStatus 1
   verifyCassetteStatus(cassetteStatus)

   # create a list of wafers to test
   myWafersTest = ""
   for waferStatus in cassetteStatus:
      if waferStatus.Status == "Present" or \
            waferStatus.Status == "Testing":
         myWafersTest = myWafersTest + " " + waferStatus.WaferId

   # verify cassette id - optional
   id = QueryCassetteID 1
   verifyCassetteID(id)

   # set project file; do not set, if the right one
   # is already configured (doing so is not wrong but takes longer)
   projectFile = GetProjectFile
   if (projectFile != myProjectFile):
      OpenProject myProjectFile myWafersTest
      Sleep(10s)

   # start the wafer job
   myJobId = StartWaferJob myFlowFile
   while True:
      # verify if our job is running
      jobStatus = JobStatus myJobId
      if jobStatus == "Aborted" or jobStatus == "InvalidJob" \
            or jobStatus == "Done":
         break

      # test if we can proceed with testing.
      # If so, start testing and proceed with probing
      probingStatus = GetProbingStatus
      if probingStatus == "AtFirstDie":
         performTest()
         ProceedProbing
      elif probingStatus.StartsWith("Error"):
         break:
      else:
         Sleep(1s)
   SetExternalMode L
```

Actual remote commands start with an upper case letter and are set in bold. If any command fails and returns an error code, the execution must stop (i.e. by using exceptions). A second cassette can be added by calling DockCassette 2 1 and GetCasseteStatus 2 and adding the wafer ids to the list of wafers to test.

C# User Interface Example

You can create a simple interface (implemented for C# only) to use with the Velox Integration Toolkit.

Here is an example that shows buttons for ‘Go Home’ and ‘Go Center.’ After the chuck moves, the chuck position is displayed beneath the buttons.



The toolkit includes example code, which can be found at C:\Program Files (x86)\Cascade Microtech\VeloxIntegrationToolkit\Examples.

- The code associated with the Go Home button is written to output the most detailed information when errors occur.
- The code associated with the Go Center button gives an example of a less detailed output. It provides minimal error checking, information about registering and closing the connection, and reading the chuck’s position. The example code is shown below.

```
var buffer = new StringBuilder(2048);
if (API.RegisterApplication("TestApp", "TestApp", 0, 0) != 0) return false;
if (API.DoProberCommand("ReadChuckPosition Y Z", buffer) != 0) return false;
lblPosition.Text = buffer.ToString();
API.CloseApplication();
return true;
```

LabVIEW

The LabVIEW part of the Velox Integration Toolkit comes as a plug-and-play instruments driver. This helps the LabVIEW developer get easy access to the Cascade Microtech probe station functionality. After installing, a CascadeProbestation node is available under the “Intruments I/O → Instr Drivers” node. See the VI Tree.vi to get an overview of the functional organization. The driver contains a subset of the most common Velox commands to control the probe station.

All VIs are created with LabView2014 SP1. The project is saved in a format compatible with LabVIEW 2009. All later versions are supported.

LabVIEW 32-bit and 64-bit

All interface VIs support both versions and reference without code changes to the correct library dll. The library names are stored in the global VeloxToolkitDLL.vi. The 64bit version of the Velox Integration Toolkit supports only a direct socket connection to the MessageServer. Therefore, the example VIs must be modified to use this connection. Please see the block diagrams of the example VIs for the Initialize.vi call. The flag "Use Socket Connection" must be set to TRUE to run the example under LabVIEW 64bit.

Referencing the Velox Integration Toolkit Library

All Interface VIs reference the library .dll files stored in the \\CascadeProbestation\\Public\\Interface folder.

The LabVIEW project contains two global variable VIs (VeloxToolkitDLL.vi, VeloxToolkitDLLPath.vi) and the GetDllPathFromReg.vi which creates the correct referencing path based on the LabVIEW installation. The GetDllPathFromReg.vi is called from the RegisterProberApp.vi and RegisterProberAppSocket.vi and sets the path in the global VeloxToolkitDLLPath.vi. This is done at the beginning of each application start by calling the Initialize.vi.

All other interface VIs use the global VeloxToolkitDLL.vi and VeloxToolkitDLLPath.vi as referencing path.

With multiple installations, it may be necessary to modify the path creating VIs (GetDllPathFromReg.vi, VeloxToolkitDLL.vi, VeloxToolkitDLLPath.vi). These VIs are not protected by the Velox Integration Toolkit license.

Interface VIs

The interface VIs encapsulate all function calls to the VeloxIntegrationToolkit.dll.

RegisterProberApp.vi	vxit_register_application
RegisterProberAppSocket.vi	vxit_register_application_socket
UnregisterProberApp.vi	vxit_close_application
DoProberCommand.vi	vxit_do_prober_command
SendCommand.vi	vxit_send_command
GetResponse.vi	vxit_get_response
GetCommand.vi	vxit_get_command
SendResponse.vi	vxit_send_response
GetErrorDescription.vi	vxit_get_error_description
GetCommandNumber.vi	vxit_get_command_number
SetBlockMode.vi	vxit_set_block_mode

NOTE

The project is saved in a format compatible with LabVIEW 2009. All later versions are supported.

Example VIs

The example VIs use the LabVIEW state machine concept and show the basic concept of communication with the probe station.

The package includes examples for:

- SimpleWaferStepping
- SettingTemperature
- VeloxProWaferStepping
- TesterInterface
- Notification

For detailed descriptions, see the Block Diagrams of each VI example. The examples are included on the installation CD.

Notification

The Notification.vi example shows the notification handling in LabVIEW applications. To receive notifications from the kernel, the application has to set the flag EnableNotification to true for the Initialize.vi (refer to Block Diagram). The example handles incoming notifications for chuck movements(X/Y/Z) and OpenProject calls.

Tester Interface Example

The TesterInterface.vi shows the basic implementation of the tester interface. This example can be used to implement a custom tester, compatible with the Velox Pro Tester Interface. For more details, see Tester Interface.

MATLAB

The Velox Integration Toolkit for MATLAB consists of wrapper functions to call the library functions directly.

function result = VxRegisterProberApplicationSocket(appName, section, singleInstance, handleNotification, host, port)	vxit_register_application
function result = VxRegisterProberApplicationSocket(appName, section, singleInstance, handleNotification, host, port)	vxit_register_application_socket
function rc = VxCloseApplication()	vxit_close_application
function [err, response] = VxDoProberCommand(command)	vxit_do_prober_command
function rc = VxSendCommand(command)	vxit_send_command
function [rc, errorCode, response] = VxGetResponse()	vxit_get_response
function [rc, messageId, parameter] = VxGetCommand()	vxit_get_command
function rc = VxSendResponse(messageId, errorCode, response)	vxit_send_response
function [rc, description] = VxGetErrorDescription(errorCode)	vxit_get_error_description
function rc = VxGetCommandNumber(command)	vxit_get_command_number
function rc = VxSetBlockMode(mode)	vxit_set_block_mode

MATLAB does not support threads natively. We recommend that you use the nonblocking mode (VxSetBlockMode(0)) and periodically call VxGetCommand()/ VxGetResponse() for command handling or asynchronous command calls. The Velox Integration Toolkit for MATLAB requires a MATLAB-compatible compiler. Please refer to the MATLAB documentation for set up details. The Velox Integration Toolkit is compatible with MATLAB R2009b and above.

The examples are included on the installation CD. See the descriptions below.

1 - Synchronous Commands
2 - Asynchronous Commands
3 - Notifications
4 - Custom Commands
5 - Wafer Map Stepping
6 - Tester Interface Template

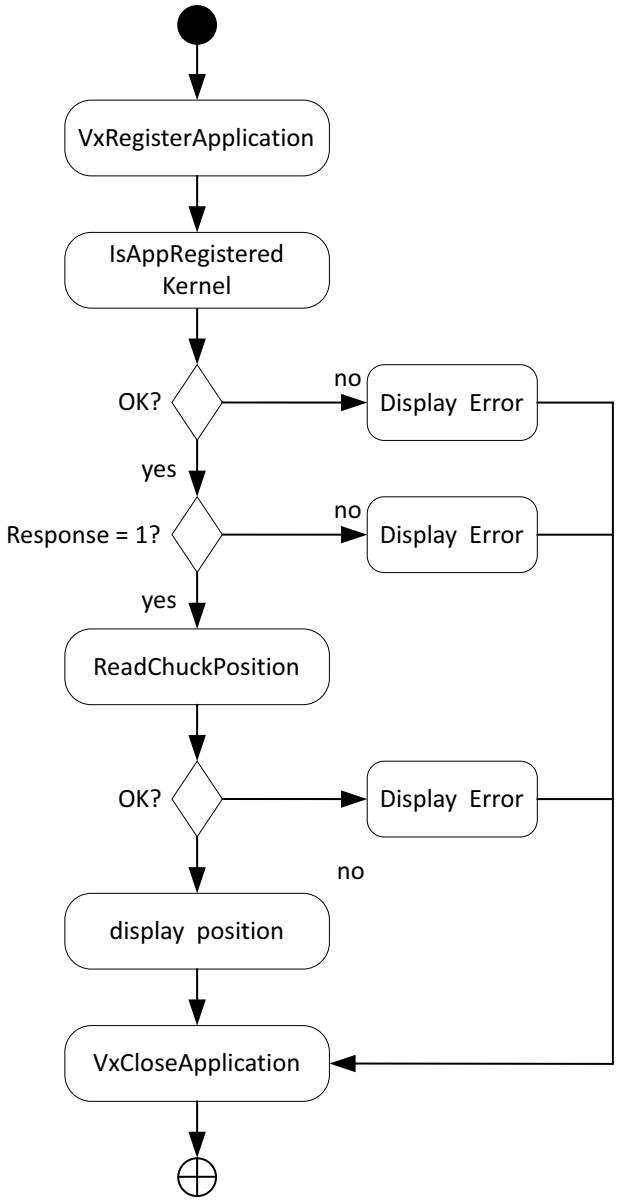
Synchronous Commands

The first example shows the general use of the Velox Integration Toolkit with synchronous commands. The key concepts are:

- Always call VxCloseApplication after VxRegisterProberApplication.

- Every command has a return code. This return code should be 0 on success.
- Some commands have a return value. The format of this return value is defined by the command.

  For example:
  - IsAppRegistered always returns a single integer: 0 or 1
  - ReadChuckPosition always returns 3 floating point values separated by a space.



Asynchronous Commands

VxSendCommand sends a command to the Message Server and returns a message ID as an identifier. VxGetResponse returns 0 if there are no responses, or a message ID. Commands and responses can be matched by those IDs. The second example shows this by invoking several chuck movements in sequence. In parallel, it will send EchoData Hello_World continuously.

Whenever one EchoData call returns, another one will be sent. The example script shows the number of successful EchoData calls per move to demonstrate this behavior.

Notifications

Notifications are basically commands for which no response is expected. To receive notifications, a script must:

- Set handleNotifications to one at vxit_register_application
- Poll VxGetCommand periodically

The example handles two notifications: RegisterProberAppChange and the AlertNotification and displays them accordingly (AlertNotification are notifications sent by the Kernel). To end the process, press Ctrl+C.

Custom Commands

You can extend the Message Server with custom commands. Please refer to Command Definitions for details. To handle custom commands, an appropriate command handler must be installed. The fourth example shows a minimalistic example. A custom command TestAppHello returns Hello <Input> . Except when the input is “John Doe.” Than an error message will be returned.

Wafer Map Stepping

This example shows a typical use case for the Velox Integration Toolkit: step through a configured Wafer Map and perform the measurement.

The Wafer Map application must be started and correctly configured. This is checked at the beginning with the remote command IsAppRegistered.

This example also shows handling of specific error codes. For example, StepNextDie returns the End of Wafer error (numeric 703) after the last die. (This is not an error but indicates that stepping is finished.)

Tester Interface Template

This template consists of a basic implementation of the tester interface. It can be used to implement a custom tester compatible with the Velox Pro Tester Interface. For more details, see Tester Interface.

Chapter

5 Tester Interface

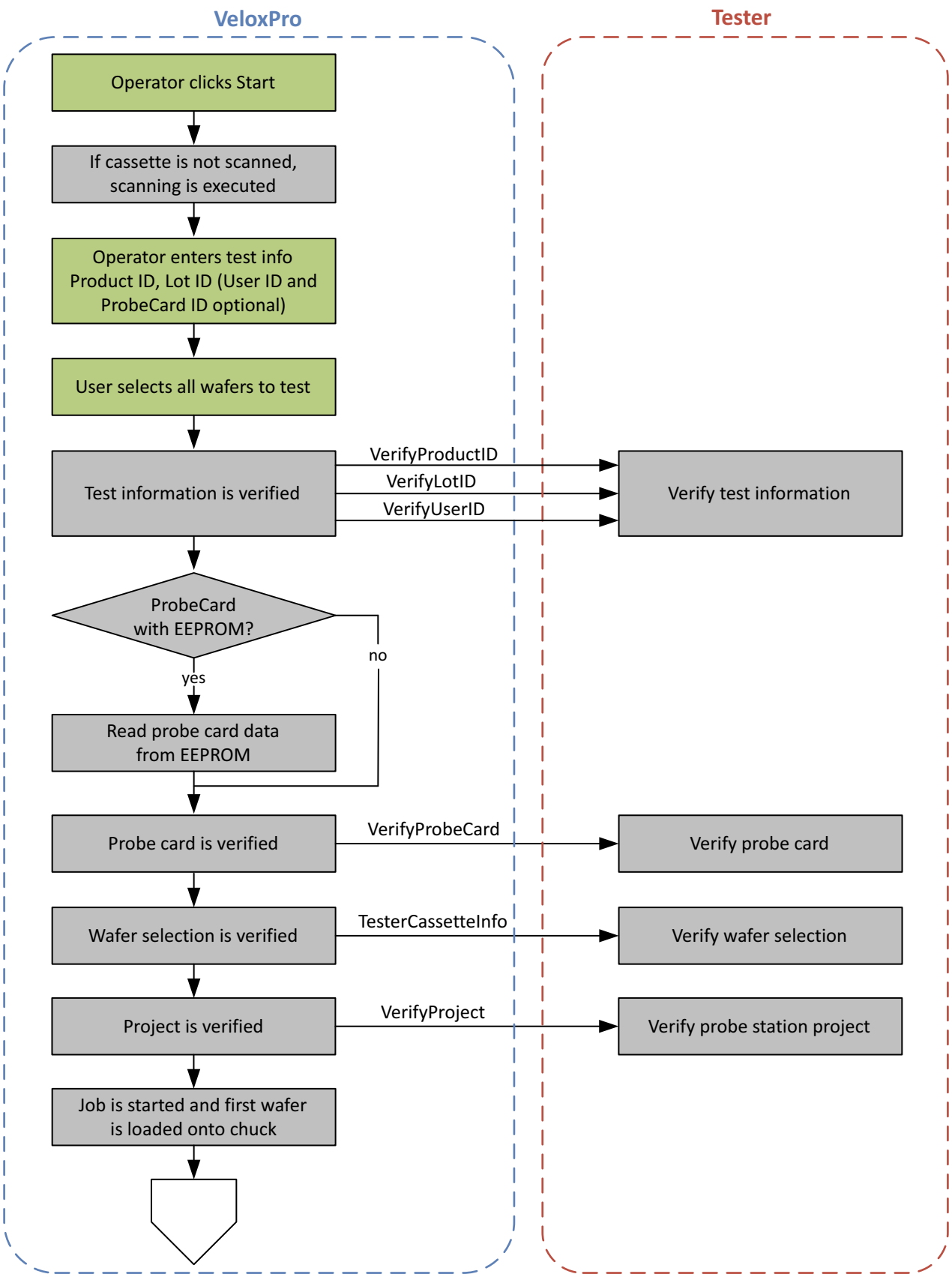
The Tester acts as slave, which means it has no control of the prober and is triggered by VeloxPro through Velox remote commands. Since the Tester application is running on the same computer, it is not necessary to use an interface such as GPIB.

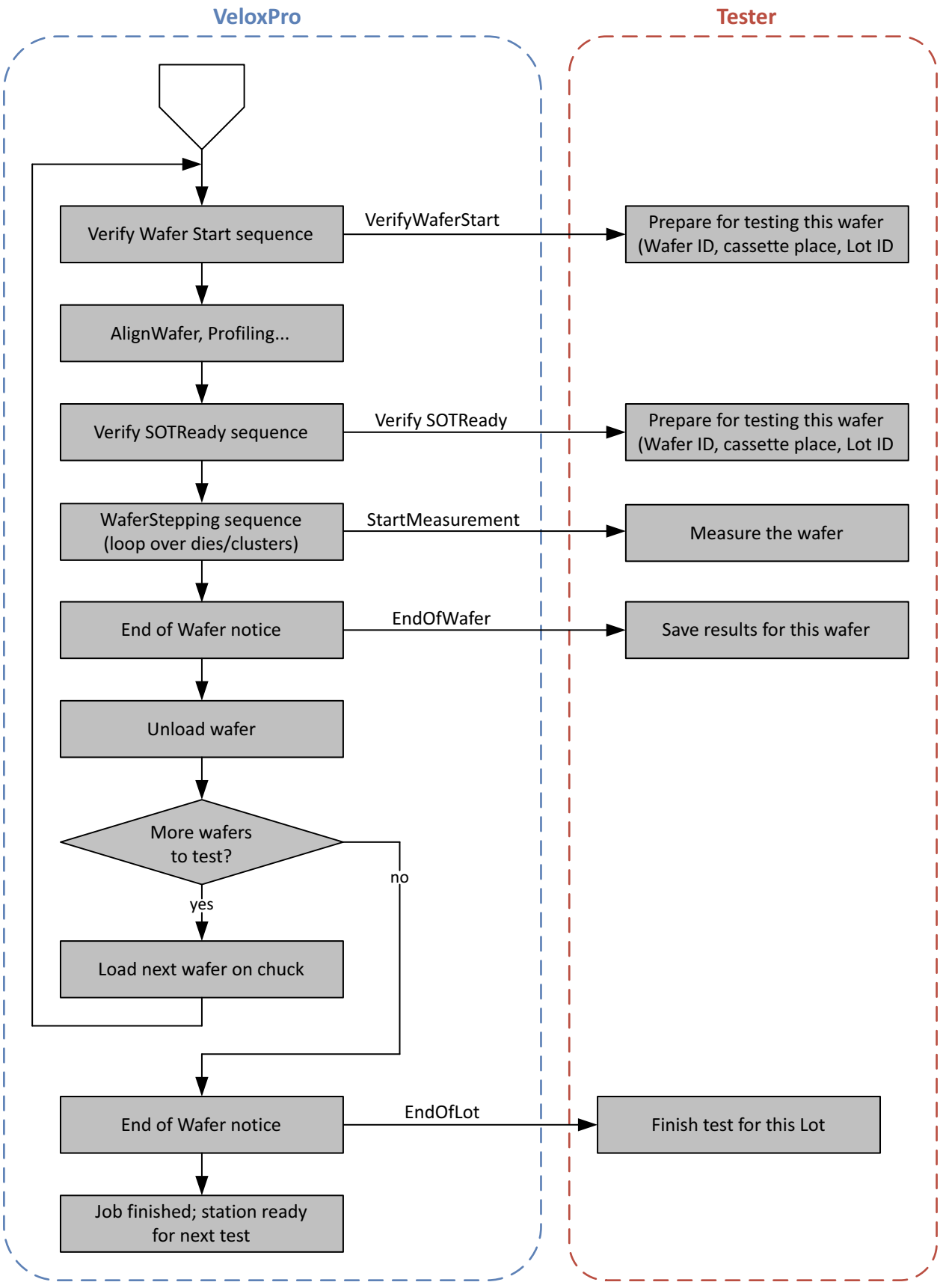
The Tester Interface is the simplest way to integrate a custom application into an automated testing project. The custom application implements a set of commands. These commands are called by VeloxPro at the appropriate time.

For details, see:

Tester Interface Commands

NewTesterProject	Sent during the VeloxPro recipe sequence VerifyProject. It uses the Lot and Tile ID that were entered into VeloxPro earlier using the WaferDialog or through the DataMatrix reader.
StartMeasurement	Sent during the VeloxPro recipe sequence WaferStepping. It is triggered for each die when the system is in Contact.
EndOfWafer	Sent during the VeloxPro recipe sequence EndOfWafer and at the end of the WaferStepping sequence.
EndOfLot	Sent after the testing of the last Lot has finished.
VerifyLotID	Sent on Job/Recipe start after the data was entered in the information dialog. It is only sent if the option Verify Lot ID was activated.
VerifySubstrateID	Sent on Job/Recipe start after the data was entered in the information dialog. It is only sent if the option Verify Product ID was activated.
VerifySubstrateID	Sent in the WaferStepping sequence when the chuck was positioned on the first die, just before the actual testing begins.
VerifyProbecard	Sent after entering the the probe card ID and only if the option Enter/Verify Probe Card ID is enabled.
VerifyUserID	Sent after entering the User ID and only if the option Enter/Verify User ID is enabled.
VerifyProject	Sent after selecting the project for the test and only if the option Verify Project is enabled.
VerifySOTReady	Sent in the VeloxPro recipe sequence Verify SOTReady.
VerifyWaferStart	Sent in the VeloxPro recipe sequence Verify Wafer Start.
TesterCassetteInfo	Sent after selecting the wafers to test if the option Verify Wafers to Test is enabled.
TesterAbort	Sent after the job execution was aborted.
TesterAbortWafer	Sent after the test for a wafer was aborted (e.g. alignment failed - skip wafer was used for error handling).


VeloxPro
Tester
Operator clicks Start
If cassette is not scanned, scanning is executed
Operator enters test info Product ID, Lot ID (User ID and ProbeCard ID optional)
User selects all wafers to test
Test information is verified
VerifyProductID
VerifyLotID
VerifyUserID
Verify test information
ProbeCard with EEPROM?
no
yes
Read probe card data from EEPROM
Probe card is verified
VerifyProbeCard
Verify probe card
Wafer selection is verified
TesterCassetteInfo
Verify wafer selection
Project is verified
VerifyProject
Verify probe station project
Job is started and first wafer is loaded onto chuck


VeloxPro
Tester
Verify Wafer Start sequence
VerifyWaferStart
Prepare for testing this wafer (Wafer ID, cassette place, Lot ID
AlignWafer, Profiling...
Verify SOTReady sequence
Verify SOTReady
Prepare for testing this wafer (Wafer ID, cassette place, Lot ID
WaferStepping sequence (loop over dies/clusters)
StartMeasurement
Measure the wafer
End of Wafer notice
EndOfWafer
Save results for this wafer
Unload wafer
More wafers to test?
no
yes
Load next wafer on chuck
End of Wafer notice
EndOfLot
Finish test for this Lot
Job finished; station ready for next test

Command Definitions

NewTesterProject

Description

This command informs the tester that a new Lot/Tile is to be tested. The tester responds with the correct project name for the test. The prober needs to verify that the correct project is loaded. If no valid project is available, the command is returned with an error.

Parameter Input

LotID

The Lot ID entered by the operator.

TileID

The Tile ID entered by the operator.

Parameter Output

Project Name

Project name for this Lot.

Examples

```
NewTesterProject Lot123 Tile456
C:\PBench\Lot123.spp
```

Invalid Lot + Spaces in Lot and Tile ID

```
NewTesterProject "Lot 456" "Tile 789"
0
```

StartMeasurement

Description

This command tells the tester to begin measuring (needles are in correct position and in contact). The command receives a response when the measurement has finished. The response string holds the binning information separated by a comma. For deactivated dies, a bin value of “0” should be returned.

Parameter Input

Die Stepping

Die Column

X Position of the current die that is to be tested.

Die Row

Y Position of the current die that is to be tested.

Cluster Stepping

Total number of dies in Cluster

Number of subsites in the current cluster.

Active number of dies in Cluster

Number of active dies in current cluster.

Active Dies

When Clusters are enabled this string contains a list of the die-positions (column, row) in the cluster and if they are activated.

Parameter Output

Bin Numbers

Measurement result as comma-separated strings. One value for each test site.

Examples

Start measurement for die -5, 12 without using clusters.

```
StartMeasurement -5 12 0
14
```

Start measurement with clusters enabled. The cluster has 4 dies but only 2 activated.

```
StartMeasurement 4 2 2,4,1;3,4,0;2,5,0;3,5,1
52,0,0,53
```

EndOfWafer

Description

This command indicates that the test of the current substrate has finished. It enables the tester application to save the measurement data and/or prepare for the next substrate.

Example

```
EndOfWafer
```

EndOfLot

Description

This command indicates that the test of the current lot has finished. It enables the tester application to save the measurement data and/or prepare for the next lot.

Example

```
EndOfLot
```

VerifyLotID

Description

This command verifies the LotID. The tester can respond with an error in case the LotID is not allowed for testing.

Parameter Input

Lot

The Lot ID entered by the operator.

Example

```
VerifyLotID Lot123
```

VerifyProductID

Description

This command verifies the ProductID. The tester can respond with an error in case the ProductID is not allowed for testing.

Parameter Input

Product

The Product ID entered by the operator.

Example

```
VerifyProductID Product123
```

VerifySubstrateID

Description
This command verifies the SubstrateID. The tester can respond with an error in case the SubstrateID is not allowed for testing.

Parameter Input

SubstrateID
The Substrate ID, either entered by the operator, or read automatically using the IDReader or the Cognex ID tools.

Example
```
VerifySubstrateID Substrate123
```

VerifyProbecard

Description
This command verifies the ProbecardID and touchdowns that are used for testing. The tester can respond with an error in case the ProbecardID is not allowed for testing.

Parameter Input

ProbecardID
The probe card ID automatically read from the EEProm.

Touchdown count
The number of touchdowns of this probe card.

Example
```
VerifySubstrateID Substrate123
```

VerifyUserID

Description
This command verifies the UserID. The tester can respond with an error in case the UserID is not allowed for testing.

Parameter Input

UserID
The User ID entered in the VeloxPro product setup page.

Example
```
VerifyUserID User123
```

VerifyProject

Description
This command is sent after a project file is selected in the VeloxPro product setup page. The command sends the name of the project to tester for verification.

Parameter Input

Project name
Name of the project selected for the test.

Example
```
VerifyProject C:\Users\Public\Documents\Velox\Projects\Test.spp
```

VerifySOTReady

Description

This command tells the tester that a wafer is ready for testing which will immediately start. This command is sent in the recipe sequence Verify SOTReady.

Example

```
VerifySOTReady
```

VerifyWaferStart

Description

This command is sent in the VeloxPro recipe sequence Verify Wafer Start and informs the tester about the current wafer on the chuck.

Parameter Input

Wafer ID

The Wafer ID, either entered by the operator, or read automatically using the IDReader or the Cognex ID tools.

Slot number

For fully automatic systems, this is the slot number where the wafer originated. For semi-automatic systems, this is always -1.

Lot ID

The Lot ID entered by the operator.

Example

```
VerifyWaferStart Wafer01 10 Lot01
```

TesterCassetteInfo

Description

This command is sent after the wafers have been selected for the test. It contains a string which represents the state of each wafer:

0 = empty
1 = full (and selected)
2 = error (e.g., double slotted)
3 = unknown
4 = deselected

Parameter Input

Cassette Info

String with each character representing one wafer slot.

Parameter Output

Cassette Info return

String with each character representing one wafer slot.

Example

```
TesterCassetteInfo 00000011100000000010010111
00000011100000000010010114
```

TesterAbort

Description
This command indicates that the job was aborted and testing stopped.
Example
`TesterAbort`

TesterAbortWafer

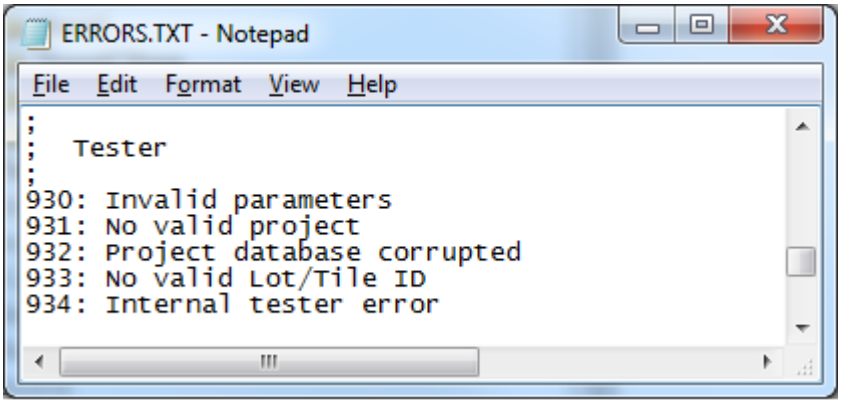
Description
This command indicates that the current wafer test was aborted and testing is stopped for the current wafer. The current job will continue.
Example
`TesterAbortWafer`

Error Codes and Messages

In case of errors, the tester application must respond to the commands with an error code and/or message.

User-defined errors must be added to the Errors.txt file in the Velox ProgramData folder of the local PC. When a command is responded to with an error code, Velox looks up the error message in this file.

Example for an Errors.txt file modification:

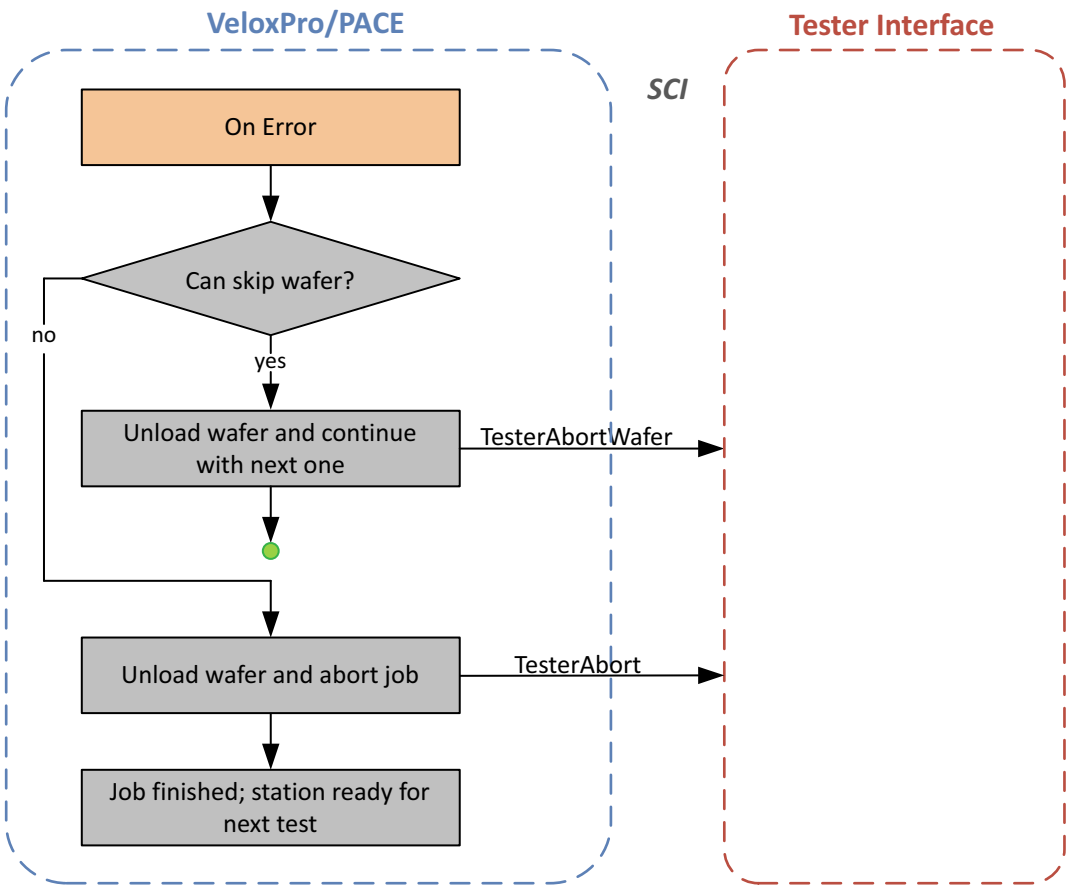


The error numbers 930-949 are reserved to be used by a custom tester.

Alternatively to the predefined messages in the Errors.txt file, the tester can provide a custom error message that must be in the standard Velox format and may contain very detailed information (such as, reason, solution, etc.)

The error string definition is as follows:

```
ERROR_NUMBER:[NAME_OF_TESTER] ErrorMessage
```


VeloxPro/PACE
Tester Interface
SCI
On Error
Can skip wafer?
no
yes
Unload wafer and continue with next one
TesterAbortWafer
Unload wafer and abort job
TesterAbort
Job finished; station ready for next test

Chapter

6 Message Server Interface

Message Server Protocol

You can communicate directly with the Message Server via sockets. The Message Server protocol is implemented in the Velox Integration Toolkit. In direct communication, handling must be implemented in the application. We recommended always using the Velox Integration Toolkit when possible.

The protocol is based on ASCII. Messages are delimited by a linebreak (\n, numerical 10). Avoid any nonprintable and any non-ASCII (>= numerical 128) character except the linebreak.

The first line in the communication process is the registration of the application:

```
Fcn=<ID>:RegisterProberApp:<Application Name> <Group Name> <Flag>
Fcn=1:RegisterProberApp:TestApplication TestApplication 0
```

The parameters:

ID	A message ID in the range 1-999 inclusive. This ID must be a 3-digit number.
Application Name	The name of the application.
Group Name	The section to use in the commands.txt.
Flag	Bit 0 — If set, this application will receive notifications. Bit 1 — If set, this application can only be started once.

The expected response:

```
Rsp=<ID>:<Return Code>:
Rsp=1:6:
```

The ID is the same as the one sent with the RegisterProberApp-Call. The return code is:

- A number greater than zero on success, or
- 0 if the application is already registered and bit one of the flags has been set.

Commands and their responses have the pattern:

```
Send:
Cmd=<ID>:<Command Name>:<Parameters>
Cmd=2:ReportKernelVersion:K

Receive:
Rsp=<ID>:<Return Code>:<Return Value>
Rsp=2:0:1.0 "PS Kernel v1.0.0 built on Mar 20 2015"
```

ID	A message ID in the 1-999 inclusive. This ID must be a 3-digit number. The response will have the same ID.
Command Name	The name of the command to call.
Parameters	The parameters of the command.
Return Code	An integer value. Usually, 0 indicates no error.
Return Value	The response of a command.

Multiple calls can be handled in parallel. Commands and responses can be matched by their ID. Handling commands and responses follow a similar pattern:

```
Receive:
Cmd=<ID>:<Command Number>:<Parameter>
Cmd=3:1:K

Send:
Rsp=<ID>:<Return Code>:<Return Value>
Rsp=3:0:1.0 "PS Kernel v1.0.0 built on Mar 20 2015"
```

This is how the kernel handles ReportKernelVersion. The Command number is sent as a hexadecimal number. Notifications have the ID 0 and no response is expected. A simple example using threads with POSIX sockets is included as sockets_example.c.

Chapter

7 Velox Python Interface (VPI)

Introduction

Velox provides an extensive API for scripting Velox applications such as WaferMap, Spectrum, etc. The interface uses SCI commands documented in the Velox Remote Interface Guide.

The Velox Scripting Console, installed with the Velox product, allows communication with a locally installed Velox probe station. Commands may be sent as simple text commands or as Python scripts. The version of Python provided in Scripting Console is Iron Python Version 2.7 from Microsoft.

The package described in this document, the Velox Python Interface (VPI), provides Python modules, making the Velox SCI command API easily accessible through a standard Python installation. Python 2.7 and Python 3.5+ are both supported. This allows communication with a local installation of Velox running on the same PC, or a remote installation communicating over network sockets.

This document describes using this Python library. You should be familiar with the Velox Integration Toolkit, supplied separately, in addition to the information presented here.

Getting Started

Using the VPI requires that a version of Python be installed. For the latest versions of Python, see www.python.org.

VPI will work with Python 2.75 or above and Python 3.5 and above. Python 3.6 or later is recommended. Python version 3.5 and above will provide the best code completion and 'Intellisense' experience for the developer.

The basic steps are:

1. Ensure that a working copy of Python 2.75+ or 3.5+ is installed. This guide assumes that you are familiar with the Python development environment.
2. Install the velox.py module using the python pip tool.
3. Use the provided samples as a starting point.

Installation

The velox.py package is not currently distributed through the standard Python Package Index, PyPI. There is an option to install via PIP from a command line.

1. Save the velox.tar.gz file to a convenient temporary location.
2. Open a command prompt at that location.
3. Type the command: `pip install velox.tar.gz`

NOTE

You must use the full file extension to avoid getting any modules named Velox from the public Python repository.

You should see several installation messages as the process proceeds.

The contents of the velox.tar.gz installation file include this document, a setup.py file, and a folder named ‘velox’. There is no need to extract the contents of the .gz file before installing.

The velox folder contains the Python module and a samples folder.

Samples

A minimal sample can be as short as three lines of code.

```
import velox
with velox.MessageServerInterface() as msgServer:
    print(velox.ReportKernelVersion())
```

All velox python programs should use the first two lines above to open a connection to the Velox Message Server. If the Velox Message Server is running on a remote PC, supply the IP address and port number as:

```
with velox.MessageServerInterface(ipaddr='x.x.x.x', targetSocket=n) as
msgServer
```

Replace ‘`x.x.x.x`’ with the IP Address of the probe station and n with the port number for the message server. The default address is ‘localhost’ and the port number is 1412. If the port number is 1412, it does not need to be specified.

Sample Code Walkthrough

The samples below are in the velox folder in a subfolder named Samples.

Example 1—Commands and Return Values – vpisample.py

```
# import velox will choose the correct version of the library
# based on the version of Python being executed
import velox

# import sys for this example, but it is not required for VPI
import sys

# print the version of Python that is running, just to see that it is
working
print('Using Python Version ' + sys.version)

# Connect to the message server using a 'with' statement.
# The application name registered with the Velox Message Server
# will be the name of our script. Keep a reference to
# the object returned by velox.MessageServerInterface
# - in this example, we call it msgServer

# the parameters to MessageServerInterface are optional.
# For a local connection, you can leave them blank. If
# connecting to a remote machine, add keyword parameters and set the
# address and socket
# velox.MessageServerInterface(ipaddr='localhost',targetSocket=1412)

with velox.MessageServerInterface() as msgServer
    # as long as the connection was successful, we can send some
commands.
    # if it was not successful, an exception is thrown.
```

```
    # try some SCI Command Examples

    # The SCI commands in the velox module have docstrings to describe
    # their function, inputs, and outputs.
    # You can print them from a Python command to see what the function
does.

    # print the docstring for the ReportKernelVersion command
    # ReportKernelVersion returns 2 values.
    # here is the description of the command

    print(velox.ReportKernelVersion.__doc__)

    # SCI commands return a namedtuple if multiple values are returned.
    # ReportKernelVersion returns a version number and a description.
    # You can access the return values by name or by indexing the tuple.

    v = velox.ReportKernelVersion()
    print('The kernel version is', v.Version, 'and', v.Description)
    print('The kernel version array is', v[0], 'and', v[1])

    # a more convenient example is to unpack the tuple into variables
    # as part of the function call.
    # here we get variables named ver and desc and use them directly
    ver, desc = velox.ReportKernelVersion()
    print('Got ', ver, 'and', desc)

    # The Echo Data command just returns its own first input.
    # Echo data only returns the first string in the parameter list.
    # To get a string with spaces to display, we need to add double
quotes.
    # Try it both ways to see the difference. The first one below does
    # what we expect. The second only prints the first word.
    # The third assigns a return value to a variable

    print(velox.EchoData('Hi World'))
    print(velox.EchoData(Hi World))
    evalue = velox.EchoData(Echo_This)

    # ReadChuckPosition returns 3 values. Here we print the entire tuple
    # to show the tuple name and the names of the 3 return values
    print(velox.ReadChuckPosition(Inches))

    # it is possible to send raw command strings with the
    # sendSciCommand function.
    # it isn't recommended, but in some cases, it might be necessary.
    # it is much easier and 'safer' to use the defined SCI functions.
    # note that this is a function of MessageServerInterface,
    # not a velox module function

    print(msgServer.sendSciCommand(echodata, rparams='RawHello Velox'))
```

Example 2—A minimal example – vpisampleminimal.py

Communication with Velox is very simple using the VPI. As simple as the following example.

```
import velox

with velox.MessageServerInterface() as msgServer

    ver, desc = velox.ReportKernelVersion()
    print('Got ', ver, 'and', desc)
    x,y,z = velox.ReadChuckPosition(Inches)
    print(x,y,z)
```

SCI Command Exception Handling

Example—Exception Handling – vpiexceptionsample.py

This code will raise an exception if Spectrum is not running. Velox command errors raise a velox.SciException.

```
import velox
with velox.MessageServerInterface() as msgServer

    # FindFocus will throw an exception because we aren't properly set
up yet.
    # This is to demonstrate exception handling.
    try
        print(velox.FindFocus())
    except velox.SciException as e
        print('We caught an exception as expected')
        print(e)
```

Hints and Tips

- Remember to prefix SCI functions with velox, as in velox.ReportKernelVersion()
- Python functions are often named using lower case, but the velox library uses mixed case names to match the SCI command documentation. Use mixed case when calling SCI commands.
- Use print(velox.commandname.__doc__) to get information about what a command does and what inputs and outputs to expect.
- SCI commands return Python namedtuples for commands that return multiple values. SCI commands that return a single value return a simple variable type.
- Use multiple return values for convenience. See the example using ReportKernelVesion.
- Use exception handling and catch the velox.SciException for SCI command errors.
- EchoData returns only the first word of a string passed to it unless it is contained in double quotes surrounded by single quotes. Example `EchoData('"Hello World"')`.

Chapter

8 Migrating from Legacy Tools

ProberBench Programmer Tools

The Velox Integration Toolkit is simpler and easier to use than the ProberBench Programmer Tools. Applications which use ProberBench Programmer Tools can be smoothly migrated to the Velox Integration Toolkit. Most command-handling functions directly map to functions in the Velox Integration Toolkit.

Simple applications relying only on DoProberCommand are easy to port to Velox Integration Toolkit. Beside the registration, all calls to DoProberCommand need to be replaced with vxit_do_prober_command.

Note

The Proberbench Programmer Tools interface is still supported. Applications which use Proberbench Programmer Tools do not need be migrated to Velox Integration Toolkit to be compatible with Velox.

Command-Dispatch Functions

Legacy function	Replaced by	Changes
RegisterProberApp	vxit_register_application	• The flag-parameter of RegisterProberApp is split into two boolean parameters • The return code is zero on success and negative on error • before reopening again, vxit_close_application must be called.
CloseProberApp	vxit_close_application	• Returns 0 on success, a negative value on error
DoProberCommand	vxit_do_prober_command	• Can return a negative value on usage error
SendCommand	vxit_send_command	• Can return a negative value on usage error • no command number parameter • now takes the whole command string
GetResponse	vxit_get_response	• The return value is the message ID or 0 if no responses are pending • Can return negative values on usage errors
GetCommand	vxit_get_command	• The return value is the command ID or 0 if no commands are pending • Can return negative values on usage errors
PostResponse	vxit_send_response	• Returns 0 on success

Additional Functions

Legacy name	Replaced by	Function
IsAppRegistered	(same)	Use the remote command "IsAppRegistered" (e.g., vxit_do_prober_command("IsAppRegistered ApplicationName", buffer))
GetCmdNumber	vxit_get_command_number	Returns a negative value on error or unknown command
GetErrorDescription	vxit_get_error_description	Returns 0 on success
SetBlockMode	vxit_set_block_mode	Returns 0 on success

Other commands aren't supported.

Programmer Tools LabVIEW Migration

The Velox Integration Toolkit supports only a subset of all available probe station commands. All older Virtual Instrument (VI) project filess must be replaced by the new Velox Integration Toolkit VIs.

- Init.vi provides the functionality to connect to the Message Server.
- Close.vi closes the connection.

Older command VIs that are not available in the Velox Integration Toolkit must be reimplemented by using the DoProberCommand.vi. For some commands, the parameter list is different. For details, see the Velox Remote Interface Help.

Cascade LabVIEW Integration Toolkit

The LabView Registration and Deregistration must be adapted to use the new Velox Integration Toolkit Virtual Instrument project files.

- All commando-VIs must be replaced with commando-VIs of the Velox Integration Toolkit.
- The SCPI command set is not supported by Velox Integration Toolkit. All commands must be SCI commands.

You can establish connection by network, either to a locally running Message Server, or remotely using sockets.

Index

Corporate Headquarters
7005 Southfront Road
Livermore, CA 94551
Phone: 925-290-4000
www.formfactor.com

FormFactor™