

Automation of Solid Edge Using External Clients Written in C++

A.C. Putman^{1*}, K. Willmert¹

¹Mechanical & Aeronautical Engineering Department, Clarkson University,
8 Clarkson Ave, Potsdam, NY, 13699

* Corresponding Author, e-mail: putmanac@clarkson.edu

Abstract

Presented in this paper are methods to write C++ programs to automate certain tasks in Solid Edge. Application programming interfaces (APIs) exist that allow users to write programs to automate and customize Solid Edge. In this paper, three sample tasks were chosen to be automated: creating an L block, a bolt, and a three bar slider assembly. Using Microsoft Visual C++ Express 2010, a program was written in C++ to accomplish each of these tasks. This paper shows that it is possible to write a relatively simple program in C++ to quickly automate a number of tasks in Solid Edge. Several resources are given to aid in the creation of different or more complex automations.

1. Introduction

Various CAD programs offer support for automation via third party scripts or programs. With support for third party automation, users can customize and extend the capabilities of their software packages in a great number of ways. Some examples include: adding support for importing or exporting additional file types, creating spreadsheets with data extracted from drawings, or creating parts or assemblies based on dimensions input by the user.

*AMO - Advanced Modeling and Optimization. ISSN: 1841-4311

Solid Edge provides a set of component object model (COM) based application programming interfaces (APIs) that can be used by any COM enabled programming language [Newell, 2009]. The languages commonly used to automate Solid Edge are Visual Basic.NET, C#, and Visual C++. A Solid Edge automation program can be created as an executable file (EXE) or a dynamically linked library (DLL). The Solid Edge Version 15 Programmer's Guide specifically defines the term "add-in" as the DLL type.

"Specifically, Solid Edge defines an add-in as a dynamically linked library (DLL) containing a COM-based object that implements the ISolidEdgeAddIn interface. More generally, an add-in is a COM object that is used to provide commands or other values to Solid Edge," [Electronic Data Services, 2004].

Programs compiled as DLLs can also be called "internal clients" whereas EXEs are referred to as "external clients."

The focus of this paper is an external client written in C++ to automate Solid Edge. The program, SE_Project, was written to perform several different tasks in Solid Edge. The main tasks are as follows:

- Automate the creation of a part with a simple extrusion feature.
- Automate the creation of a more complex part with additional features.
- Automate the creation of an assembly and its associated parts.
- Automate the physical properties analyses of a part.

Automation of Solid Edge Using External Clients Written in C++

The objects chosen for these tasks are, respectively, an L-shaped block, a bolt, and a three bar slider assembly as shown in Figure 1. An external client was chosen over an add-in due to the relative simplicity of the source code, the user interface, and the installation of the program. A command line external client is also well suited to perform the type of tasks listed above.

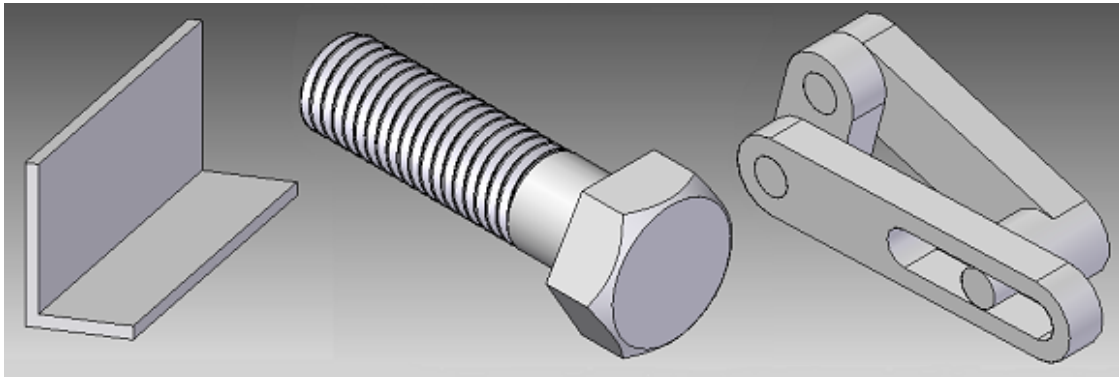


Figure 1: L-shaped block, bolt, and three bar slider rendered in Solid Edge

Examples of Solid Edge automation programs, and a general explanation of how they can be created and how they work will be provided in the following section. It will be shown, generally, how the program performs the previously listed individual tasks in Section 3.

Conclusions on the external client program, SE_Project, and recommendations for the creation of Solid Edge automation programs will be discussed in the final section. The source code for the SE_Project program discussed in this paper is contained in [Putman, 2011]. Three tutorials on writing Solid Edge external clients in C++ for Solid Edge are also in [Putman, 2011].

2. Background

There are several available add-ins and external clients for Solid Edge. Many are commercial products. Among them are: InspectionXpert by Extensible CAD Technologies, Border Control by IngeneaSoft, File Control by IngeneaSoft, Solid Edge Spy by Jason Newell, and several file importing and exporting programs by SYCODE [Cope, 2010], [IngeneaSoft Border Control, 2010], [IngeneaSoft File Control, 2010], [Newell, n.d.], and [SYCODE, n.d.]. All of these examples are closed source, meaning the source code is not publicly available. InspectionXpert and the SYCODE programs are add-ins. The rest connect to Solid Edge as external clients. Only Border Control and Solid Edge Spy are available free of charge.

Also available are resources for creating Solid Edge add-ins, external clients, and for Solid Edge programming in general. The Solid Edge ST Addins - Part I article and Solid Edge AddIn Wizard by Jason Newell are good resources for learning how to create add-ins written in C++ [Newell, 2009] and [Newell, 2006]. Both the article and the wizard are based on the Active Template Library (ATL), which causes them to require the standard version or greater of Microsoft Visual Studio to compile. Once a C++ add-in is created it needs to be registered using regsvr32.exe for it to be recognized by Solid Edge. This process adds entries to the Windows registry. There are 32bit and 64bit versions of Solid Edge. On a 64bit Windows operating system, depending on whether Solid Edge is 32bit or 64bit, the Solid Edge installation process will create registry keys in different locations. This difference between 32bit and 64bit registry keys also affects where the entries need to be made for Solid Edge to recognize an add-in. As a result, specific 32bit and 64bit versions of the add-in may need to be compiled. After the add-in

Automation of Solid Edge Using External Clients Written in C++

is registered, a running instance of Solid Edge can connect to the add-in either at startup, by user request, or by an external program. Users typically interact with a Solid Edge add-in via command buttons, which can be added, along with Solid Edge toolbars, by the add-in.

The Solid Edge Version 15 Programmer's Guide contains several code examples for external clients [Electronic Data Services, 2004]. Most of the examples are written in Visual Basic, but one example, in chapter 15, is a C++ command line program. The programs written for this project and the tutorials (see [Putman, 2011]) are all based on that C++ example program. These programs are not based on ATL, and can be compiled using Microsoft Visual C++ Express which is available from Microsoft free of charge. External clients do not need to be registered like add-ins, and can be started without a running instance of Solid Edge. They can start and connect to a new instance of Solid Edge or connect to one that already exists. Users typically interact with an external client by simply running the program.

Other online resources for Solid Edge programming include the www.jasonnewell.net forums and the Solid Edge newsgroups (available at <http://bbsnotes.ugs.com/vbulletin>). The programming forums and newsgroups cover topics relating to VB.NET, C#, and C++, although most topics are focused on VB.NET. Another resource, the Solid Edge ST Programmer's Guide included with Solid Edge ST, ST2, and ST3, may not be terribly helpful for writing Solid Edge automation programs in C++ as it focuses entirely on VB.NET and C# [Siemens Product Lifecycle Management Software Inc., 2008]. Solid Edge Spy is a great tool for Solid Edge developers, as it allows the user to view the objects, processes, and events of an active instance of Solid Edge. The object browser in Microsoft Visual Studio and the Solid Edge SDK provide

important information and definitions for all of the various Solid Edge APIs [Siemens Product Lifecycle Management Software Inc., 2009]. The Solid Edge SDK also contains code examples. However, all of the examples are written in VB.NET. The SDK can be accessed by going to the Solid Edge help index and clicking on "Programming With Solid Edge." Lastly, Siemens Global Technical Access Center (GTAC) can be contacted directly by phone (for details see <http://support.ugs.com/gtac.shtml>).

Writing the program as an external client is an attractive option for the tasks presented in this paper and was selected for a number of reasons:

- The program does not need to be registered on every computer to use it.
- The program can be run without having to open a document in Solid Edge.
- The user interface can be handled entirely by a command line window rather than Solid Edge toolbars, command buttons, and windows.
- The program can be compiled with the free Microsoft Visual C++ Express.
- The program is compatible with both 32bit and 64bit versions of Solid Edge and Windows.

To create a C++ external client to automate Solid Edge, Solid Edge needs to be installed and the directories "\Solid Edge ST2\SDK\include" and "\Solid Edge ST2\Program" must be added as include directories in the compiler's project properties. The necessary Solid Edge type libraries must also be imported (e.g. `#import "framework.tlb"`). Also, `objbase.h` and `comdef.h` header files need to be included. The basic steps the program must perform are as follows:

Automation of Solid Edge Using External Clients Written in C++

- Initialize the COM object.
 - `CoInitialize(NULL);`
- Declare a smart pointer for the Solid Edge application object.
 - `SolidEdgeFramework::ApplicationPtr pSEApp;`
- Point the application smart pointer to the application object of a running or new instance of Solid Edge.
 - `pSEApp.GetActiveObject("SolidEdge.Application");` or
 - `pSEApp.CreateInstance("SolidEdge.Application");`
- Perform the desired Solid Edge automations.
- Uninitialize the COM object.
 - `CoUninitialize();`

The type libraries each contain information about Solid Edge objects, their methods, and their properties. Solid Edge automation is performed by creating and manipulating objects. For these objects to be accessible, their respective type libraries must be imported. The one type library that must be imported in order to automate Solid Edge is the framework type library, `framwrk.tlb`. This type library contains the Application object, which is the gateway to all of the other Solid Edge objects. A list of all of the Solid Edge type libraries and their respective namespaces can be found in Table 1. With the release of ST3, the PartSync and AssemblySync type libraries were removed and their methods and properties merged with the Part and assembly type libraries.

Type Library File Name	Type Library Namespace
constant.tlb	SolidEdgeConstants
framewrk.tlb	SolidEdgeFramework
fwksupp.tlb	SolidEdgeFrameworkSupport
geometry.tlb	SolidEdgeGeometry
Part.tlb	SolidEdgePart
assembly.tlb	SolidEdgeAssembly
draft.tlb	SolidEdgeDraft
installdata.tlb	SEInstallDataLib
revmgr.tlb	RevisionManager
PartSync.tlb	SolidEdgePartSync
assemblysync.tlb	SolidEdgeAssemblySync

Table 1: Solid Edge ST2 Type Libraries

Once an object is made accessible by importing its type library, it can eventually be manipulated by using smart pointers. The first smart pointer any Solid Edge automation program must use is the application smart pointer, ApplicationPtr, from the SolidEdgeFramework namespace. In the example above, the application smart pointer was named pSEApp. To point pSEApp to the Solid Edge application object, the GetActiveObject or CreateInstance methods from the COM API can be used. Once pSEApp points to the Solid Edge application object, the rest of the objects in the Solid Edge API can be accessed. In order to access these objects, the object hierarchy of the Solid Edge API must be followed. From [Putman, 2011]:

"The Solid Edge object hierarchy can be depicted as a tree diagram, always starting with the Application object. This tree then branches out into the other objects such as the Documents collection object, which leads to a Document object, which leads to the ProfileSets collection object, and so on. There are often plural and singular forms of objects in this hierarchy, such as Documents and Document. The plural form is always the parent of the singular form, child

Automation of Solid Edge Using External Clients Written in C++

object. The Documents collection object is an indexed collection of Document objects, just as the ProfileSets collection object is an indexed collection of ProfileSet objects."

A child object can be accessed or created by using the properties or methods of its parent object. The properties and methods of an object are accessed using the "-">" operator. For example, the document object can be created and a smart pointer pointing to it can be returned by using the GetDocuments and Add methods, although those methods are part of the COM API. The GetDocuments method returns a documents object smart pointer. The Add method creates and returns a document object smart pointer, to which the smart pointer pPartDocument is then assigned.

```
pPartDocument = pSEApp->GetDocuments()->Add("SolidEdge.PartDocument");
```

The ProfileSets smart pointer, pProfileSets, can then be assigned to the ProfileSets object smart pointer returned by using the ProfileSets property of the PartDocument object smart pointer, pPartDocument.

```
pProfileSets = pPartDocument->ProfileSets;
```

A ProfileSet object can then be created and pointed to by the smart pointer pProfileSet using the Add method of the ProfileSets object, pProfileSets.

```
pProfileSet = pProfileSets->Add();
```

Alternatively, the above two steps could be performed in one line similar to how the PartDocument object was created.

```
pProfileSet = pPartDocument->ProfileSets->Add();
```

In this way, much of what can be accomplished by hand in Solid Edge can be automated.

Detailed information about all of the Solid Edge objects, their associated properties and methods, and the arguments of those methods can be found in the Solid Edge SDK.

3. SE_Project Program Overview

As previously stated, the SE_Project external client connects to a running instance of Solid Edge, or creates and connects to a new instance. The program then automates Solid Edge to produce an L-shaped block, a bolt, or a three bar slider assembly. It also gives the option to run a physical properties analysis on the L-shaped block or bolt parts. This is accomplished by running SE_Project.exe and providing the input requested in the command line window. The SE_Project external client is based on the example C++ program in chapter 15 of the Solid Edge Version 15 Programmer's Guide [Electronic Data Services, 2004]. This example program is made up of an error-handling macro, HandleError, and four functions: main, RunSEAutomation, CreateAssemblyUsingPartFile, and CreateDrawingUsingAssemblyFile.

The main function initializes the COM object, calls the RunSEAutomation function, and then uninitializes the COM object. RunSEAutomation first tries to connect to an existing instance of Solid Edge. Failing that, it starts a new instance of Solid Edge. In either case, an application smart pointer to Solid Edge is created. RunSEAutomation then sets the Solid Edge application to

Automation of Solid Edge Using External Clients Written in C++

be visible using the Visible property of the Application object, pSEApp. It then calls the CreateAssemblyUsingPartFile and CreateDrawingUsingAssemblyFile functions. Those functions in turn automate Solid Edge. The HandleError macro is used in main and RunSEAutomation to display error messages and to ensure that any necessary cleanup is performed before the program exits after an error.

The RunSEAutomation function exists to separate all of the smart pointer declarations from the main function. Runtime crashes can occur if any COM objects or interfaces are used or destructed after the COM object is uninitialized. Smart pointers are created on the stack, so if a smart pointer is declared and a COM object is uninitialized in the same scope the smart pointer is destructed after CoUninitialize is called.

The programs in the tutorials given in [Putman, 2011] use a slightly updated version of the example program with new automation functions replacing the CreateAssemblyUsingPartFile and CreateDrawingUsingAssemblyFile functions. Additional code is also written to provide the interface for a user to input dimensions and to check the input data for errors. The SE_Project program expands on this by adding several functions to automate Solid Edge and allowing the user to select a desired unit of length for the input dimensions. The code for SE_Project is also separated into various source and header files, whereas the source code for the example and tutorial programs are each contained in a single source file. The organization of the source code for SE_Project is shown in Table 2. The complete, commented source code for SE_Project can be found in [Putman, 2011]. The rest of this section is intended to be read alongside of the source code.

File Name	Description
stdafx.h	Contains the include(s) and imports the type libraries necessary for the other files.
SE_Project.cpp	Contains HandleError macro and the base functions of the program: main and RunSEAutomation.
SE_Common.h	The SE_Common class contains the functions used by multiple other classes and RunSEAutomation: InputInt, InputDouble,
SE_Common.cpp	SelectUnit, and WritePhysicalProperties.
LBlock.h	The LBlock class contains the functions that are specific to the creation of an L-shaped block part file: CreateLBlock,
LBlock.cpp	LBlockDlg, and MakeLBlock.
Bolt.h	The Bolt class contains the functions that are specific to the creation of a bolt part file: CreateBolt, BoltDlg, and MakeBolt.
Bolt.cpp	
ThreeBarSlider.h	The ThreeBarSlider class contains the functions that are specific to the creation of a three bar slider assembly file and its associated part files: CreateThreeBarSlider, ThreeBarSliderDlg, MakeLink,
ThreeBarSlider.cpp	MakePin, and MakeAssy.

Table 2: SE_Project Source Code Files

3.1 *stdafx.h*

The *stdafx.h* header file contains the includes and imports the type libraries that are needed for most of the source files. The *partsync.tlb* and *assemblysync.tlb* type libraries from Solid Edge ST2 are not imported, as they do not exist in ST3 and are not used in this program. This allows *SE_Project* to work on both the ST2 and ST3 versions. The *draft.tlb*, *installdata.tlb*, and *revmgr.tlb* type libraries exist in both ST2 and ST3 versions, but are commented out because they are not currently used in this program.

3.2 *SE_Project.cpp*

The *SE_Project.cpp* source file contains an adapted form of the C++ program example from the Solid Edge Version 15 Programmer's Guide. The main function initializes the COM object, calls *RunSEAutomation*, and uninitialized the COM object. The *RunSEAutomation* function looks for an instance of Solid Edge to connect to or, failing that, creates a new one. The *SelectUnit*

Automation of Solid Edge Using External Clients Written in C++

function from the SE_Common class is then called to allow the user to select the desired unit of length for future data input. Next, the user is asked to select an automation routine to run. The user input is returned as an integer by the InputInt function from the SE_Common class. The appropriate function is called based on the user input or, if the user input was erroneous, an error message is displayed and the user is asked again to select a routine.

3.3 The LBlock

The LBlock.h header file and LBlock.cpp source file make up the LBlock class. This class contains the CreateLBlock, LBlockDlg, and MakeLBlock functions. The LBlockDlg function retrieves the dimensions for the L-shaped block from the user, while MakeLBlock uses the input dimensions to create a part document and draw the L-shaped block object. LBlockDlg and MakeLBlock are both private functions that cannot be called outside of the LBlock class. CreateLBlock is the public function that can be called outside of the LBlock class and in turn calls the LBlockDlg and MakeLBlock functions to run the L-shaped block automation routine, a result of which is shown in Figure 2.

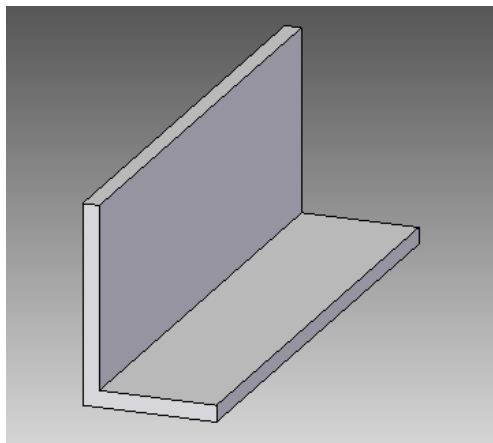


Figure 2: L-shaped block rendered in Solid Edge

3.3.1 LBlockDlg

The LBlockDlg function handles the interaction between the user and the L-shaped block routine. It begins by displaying a description of what the routine will do. The InputDouble function from the SE_Common class is then used to return the user input as doubles. These values are converted to meters using a conversion factor that was previously set by the SelectUnit function from the SE_Common class when it was called from RunSEAutomation. The results are assigned to the variables that define the dimensions of the L-shaped block, illustrated in Figure 3. The dimensions are then checked for any errors. If any errors are detected, an error message will be displayed and the user will be asked again to input the dimensions of the L-shaped block. If no errors are detected the function will return. This process prevents errors in Solid Edge when constructing the objects as well as warning the user up front about any data entry errors.

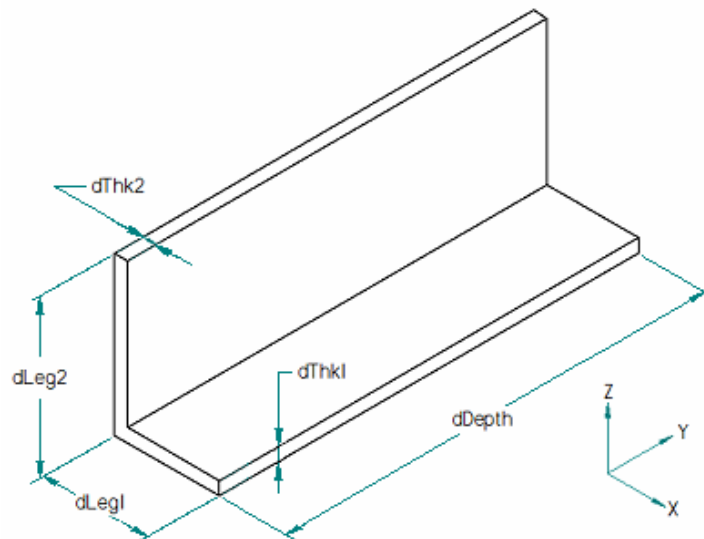


Figure 3: Dimensioned drawing of the L-shaped block

3.3.2 MakeLBlock

The MakeLBlock function takes the dimensions assigned by the LBlockDlg function and makes the L-shaped block in Solid Edge. The L-shaped block, as shown in Figure 2, is simply an extruded protrusion of an L-shaped profile consisting of six lines as shown in Figure 4.

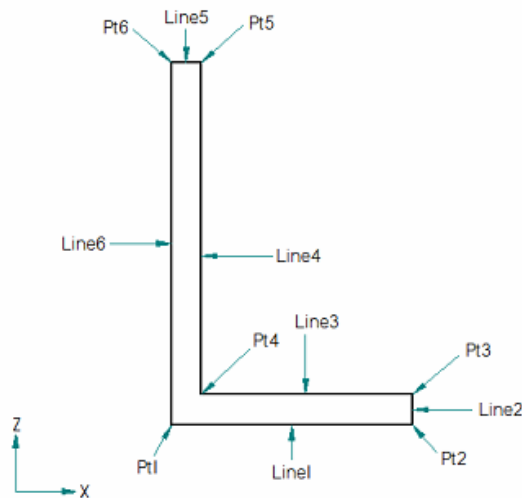
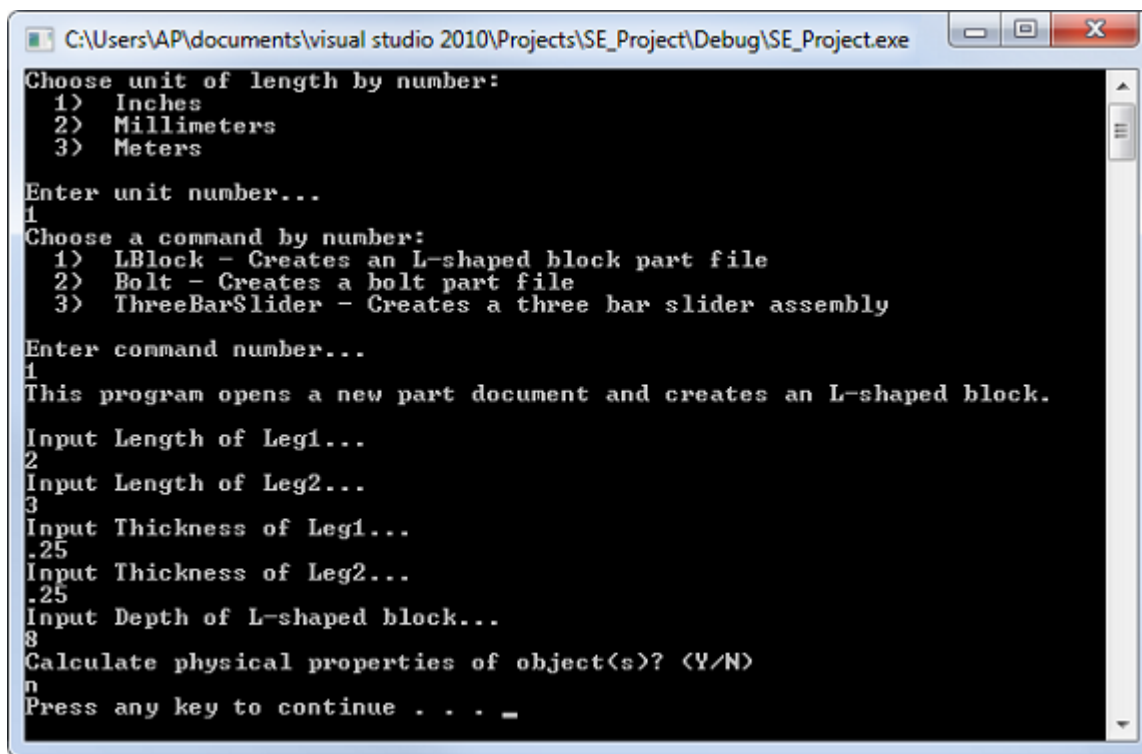


Figure 4: Labeled profile of the L-shaped block

First a new part document is created using the GetDocuments and Add COM functions. A Profile object, pProfile, is then created. The profile is added to the third reference plane also known as the "Front" or x-y plane. The dimensions of the part are used to assign values to variables representing the x and z coordinates of the six points that define the profile. Six lines are then created using these values and the AddBy2Points method of the Lines2d object, pLines2d. After the lines for the profile are drawn, keypoints need to be defined at the intersections between each of the lines. Once all six keypoints are defined the profile can be closed using the end method of the Profile object pProfile. Finally, the profile is extruded using the AddFiniteExtrudedProtrusion method of the Models object pModels. The safearray used for

the creation of the extruded protrusion is then destroyed. Safearrays, unlike smart pointers, are not automatically cleaned up and must be manually destroyed before leaving scope. The WritePhysicalProperties function from the SE_Common class is then called. Finally the function returns, leaving the part document for the user to save or discard. Figure 5 shows a command line window for a completed L-shaped block routine.



```
C:\Users\AP\documents\visual studio 2010\Projects\SE_Project\Debug\SE_Project.exe
Choose unit of length by number:
 1> Inches
 2> Millimeters
 3> Meters
Enter unit number...
1
Choose a command by number:
 1> LBlock - Creates an L-shaped block part file
 2> Bolt - Creates a bolt part file
 3> ThreeBarSlider - Creates a three bar slider assembly
Enter command number...
1
This program opens a new part document and creates an L-shaped block.
Input Length of Leg1...
2
Input Length of Leg2...
3
Input Thickness of Leg1...
.25
Input Thickness of Leg2...
.25
Input Depth of L-shaped block...
8
Calculate physical properties of object(s)? (Y/N)
n
Press any key to continue . . . _
```

Figure 5: A command line window for a completed L-shaped block

After starting SE_Project.exe, the command line window appears and the program attempts to connect to Solid Edge. If that fails it creates an instance of Solid Edge and connects to it. The program then prompts the user to select a unit of length by entering an integer. The options are: 1 for inches, 2 for millimeters, and 3 for meters. In Figure 5 "1" was entered for inches. Next, the user is prompted to choose an object to create by entering an integer. In this case the LBlock

Automation of Solid Edge Using External Clients Written in C++

object was selected by entering "1." A description of what the LBlock routine accomplishes is displayed and the user is asked to input the dimensions for the object. The dimensions of the L-shaped block created in Figure 5 are as follows: Leg1 is 2 inches, Leg2 is 3 inches, Thickness of Leg1 = 0.25 inches, Thickness of Leg2 = 0.25 inches, and Depth is 8 inches. A new part file is then created and the L-shaped block is drawn in Solid Edge. The user is then asked whether or not to calculate the physical properties of the L-shaped block. In this case "n" is entered and the physical properties analysis is not performed. The program then disconnects from Solid Edge and pauses, displaying the "press any key to continue" message. After pressing a key the program exits and the command line window closes.

3.4 The Bolt

The Bolt.h header file and Bolt.cpp source file make up the Bolt class. This class contains the CreateBolt, BoltDlg, and MakeBolt functions. The BoltDlg function retrieves the dimensions for the bolt from the user, while MakeBolt uses the input dimensions to create a part document and draw the bolt. BoltDlg and MakeBolt are both private functions that cannot be called outside of the Bolt class. CreateBolt is the public function that can be called outside of the Bolt class and in turn calls the BoltDlg and MakeBolt functions to run the bolt automation routine. An example of a bolt created with this routine is shown in Figure 6.

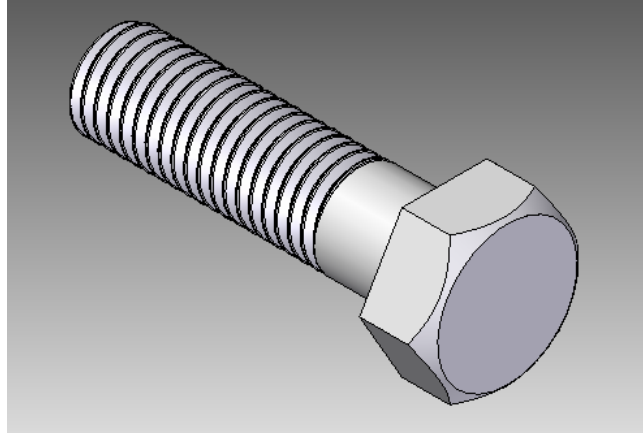


Figure 6: Bolt rendered in Solid Edge

3.4.1 BoltDlg

The BoltDlg function handles the interaction between the user and the bolt routine. It begins by displaying a description of what the routine will do. The InputDouble function from the SE_Common class is then used to return the user input as doubles. These values are converted to meters using a conversion factor that was previously set by the SelectUnit function from the SE_Common class when it was called from RunSEAutomation. The minimum allowable threads per unit length is calculated and displayed during that process to help prevent the user from entering an erroneous value. The results are assigned to the variables that define the dimensions of the bolt, some of which are shown in Figure 7. The dimensions are then checked for any errors. If any errors are detected, an error message will be displayed and the user will be asked again to input the dimensions of the bolt. If no errors are detected the function will return.

Automation of Solid Edge Using External Clients Written in C++

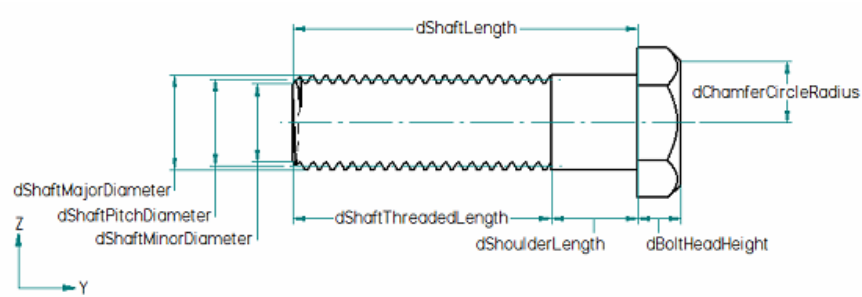


Figure 7: Dimensioned drawing of the bolt

3.4.2 MakeBolt

The MakeBolt function takes the dimensions assigned by the BoltDlg function and makes the bolt in Solid Edge. The bolt is made up of multiple extruded protrusions, two revolved cutouts, and one helical cutout.

First a new part document is created using the GetDocuments and Add COM functions. Several additional parameters are defined for the bolt that are either predetermined or calculated using the values assigned by BoltDlg.

The threaded shaft of the bolt is drawn first. The profile, pProfile, is created on the third reference plane. A Circle2d object is then created on that profile using the AddByCenterRadius method of the Circles2d object, pCircles2d. Because it is a circle, the profile does not require any keypoints and does not need to be manually closed. The circle is extruded using the AddFiniteExtrudedProtrusion method of the Models object, pModels. The safearray aProfiles is not destroyed, but assigned a NULL value. This is done because the aProfiles safearray will be assigned new values and used later in the function.

Now that the base protrusion for the threaded shaft is complete, the helical cutout for the threads will be created. First, dimensions and coordinates for the helical cutout's profile are defined. Next, a new profile, pHelicalCutoutProfile, is created, this time on the second reference plane. Note that pHelicalCutoutProfile is a child of a new Profiles object that was also created, but not assigned to a smart pointer. Each feature should have a new Profiles object as well as a new Profile object to avoid errors. A reference axis, pRefAxis, is then created as an argument for the helical cutout. The lines and arcs that make up the profile of the helical cutout are created and the profile is then closed. Two planes are created using the AddParallelByDistance method of the RefPlanes object, pRefPlanes. These planes mark the beginning and end of the helical cutout. The helical cutout can then be created using the AddFromTo method of the HelixCutouts object. Dimensioned and labeled drawings of the profile for the helical cutout are shown in Figure 8 and Figure 9 respectively.

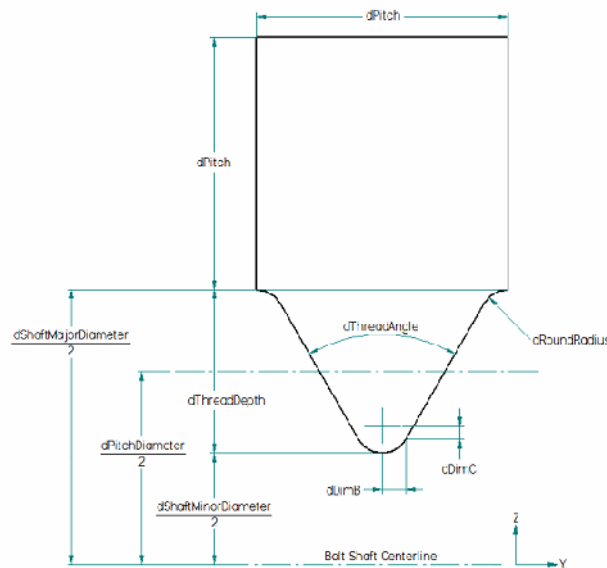


Figure 8: Dimensioned drawing of the helical cutout profile

Automation of Solid Edge Using External Clients Written in C++

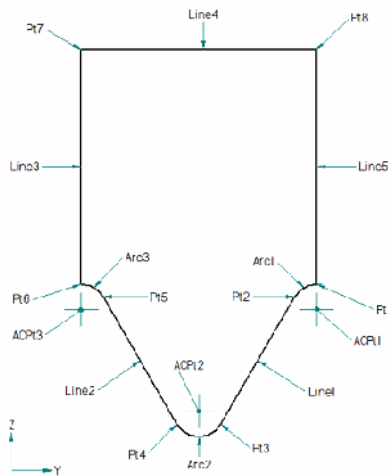


Figure 9: Labeled drawing of the helical cutout profile

After the helical cutout is complete the safearray `aProfiles` is destroyed, as no additional safearrays will be needed for the duration of the function. If the user inputs a shoulder length greater than zero, the function will then create that shoulder by extruding another circle. A new profile, `pShoulderProfile`, is created on the third reference plane for this extrusion. The `AddFiniteExtrudedProtrusion` method of the `Models` object was used to create the extrusion for the threaded portion of the bolt shaft as a base feature. For additional extrusions, the `AddFinite` method of the `ExtrudedProtrusions` object can be used, such as this case with the shoulder of the bolt.

The final extrusion, the hex head of the bolt, is created on the new profile, `pHeadProfile`, located again on the third reference. The calculated dimensions and coordinates are used to create the hexagonal profile as shown in Figure 10. Again, the `AddFinite` method is used to create the extrusion for the bolt head.

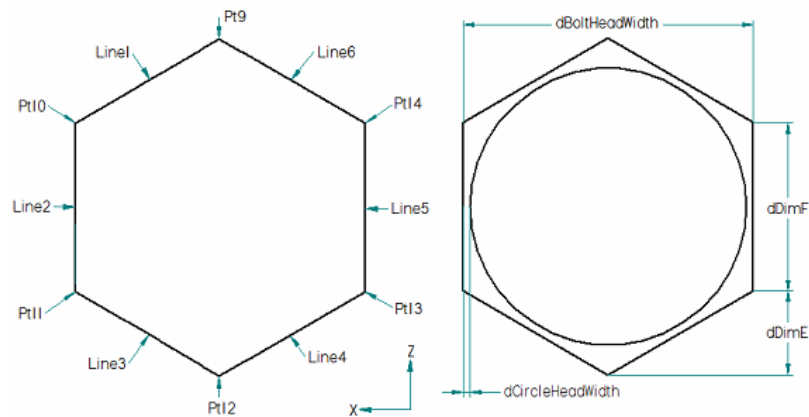


Figure 10: Labeled and dimensioned drawings of the bolt head

After the hexagonal extrusion is created, a chamfer, shown on the right in Figure 10, is needed to finish the bolt head. This is created using a revolved cutout. A new profile, pHeadChamferProfile, is created on the second reference plane for this feature. The dimensions and coordinates for the triangular profile, shown in Figure 11, are calculated and the profile is drawn and closed. The revolved cutout is created using the AddFinite method of the RevolvedCutouts object.

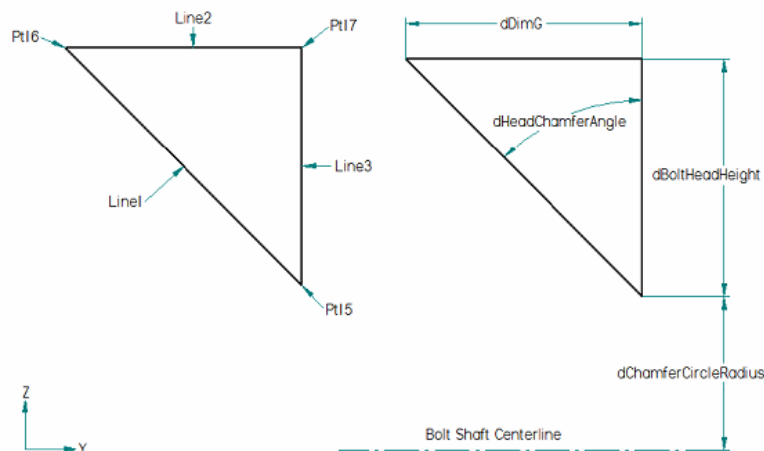


Figure 11: Labeled and dimensioned drawings of the bolt head chamfer profile

Automation of Solid Edge Using External Clients Written in C++

The final feature is the chamfer on the threaded end of the bolt. Another triangular profile is drawn on the new profile pEndChamferProfile as shown in Figure 12. The AddFinite method is once again used to create a revolved cutout feature.

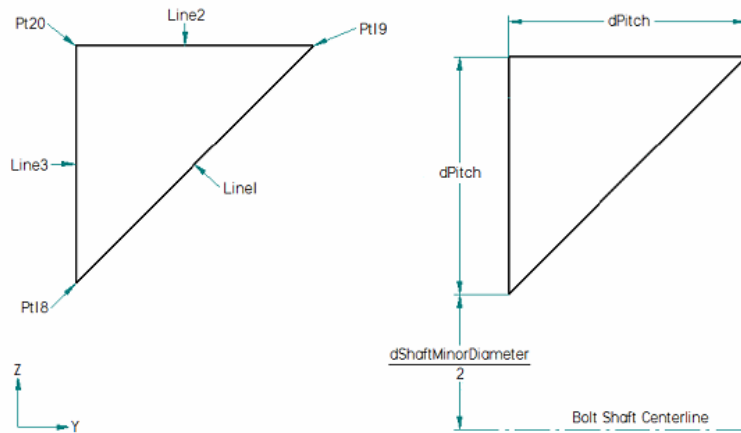
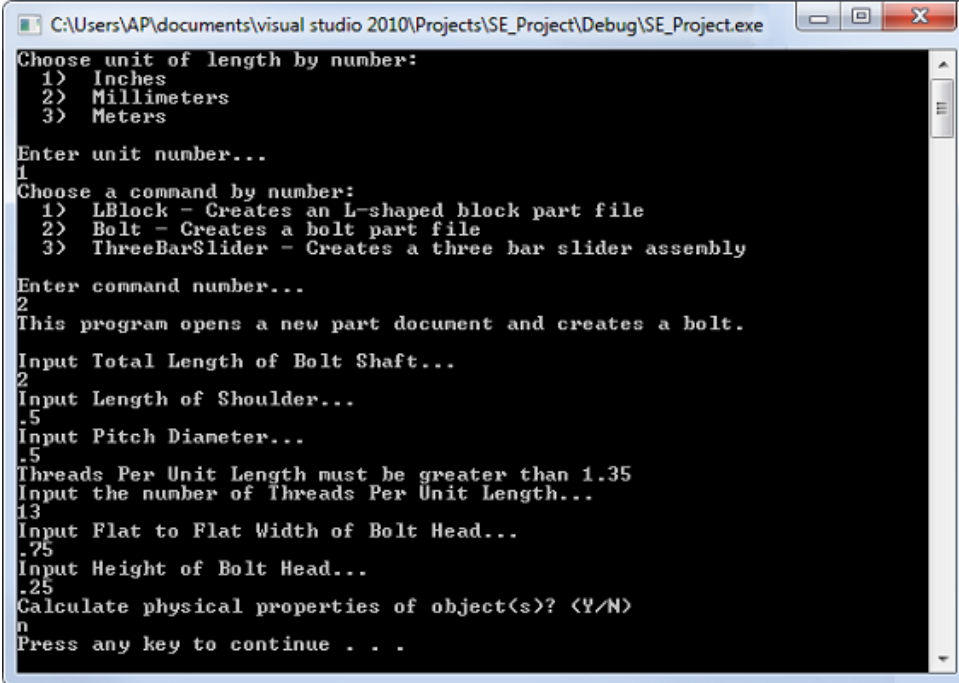


Figure 12: Labeled and dimensioned drawings of the bolt end chamfer profile

The WritePhysicalProperties function from the SE_Common class is then called. Finally the function returns, leaving the part document for the user to save or discard. Figure 13 shows a command line window for a completed bolt routine.



```
C:\Users\AP\documents\visual studio 2010\Projects\SE_Project\Debug\SE_Project.exe
Choose unit of length by number:
1) Inches
2) Millimeters
3) Meters
Enter unit number...
1
Choose a command by number:
1) LBlock - Creates an L-shaped block part file
2) Bolt - Creates a bolt part file
3) ThreeBarSlider - Creates a three bar slider assembly
Enter command number...
2
This program opens a new part document and creates a bolt.
Input Total Length of Bolt Shaft...
2
Input Length of Shoulder...
.5
Input Pitch Diameter...
.5
Threads Per Unit Length must be greater than 1.35
Input the number of Threads Per Unit Length...
13
Input Flat to Flat Width of Bolt Head...
.75
Input Height of Bolt Head...
.25
Calculate physical properties of object(s)? (Y/N)
n
Press any key to continue . . .
```

Figure 13: A command line window for a completed bolt

3.5 The Three Bar Slider Assembly

The ThreeBarSlider.h header file and ThreeBarSlider.cpp source file make up the ThreeBarSlider class. This class contains the CreateThreeBarSlider, ThreeBarSliderDlg, MakeLink, MakePin, and MakeAssy functions. The ThreeBarSliderDlg function retrieves the dimensions for the three bar slider from the user, while MakeLink and MakePin use the input dimensions to create, draw, and save the links and pins in the assembly. MakeAssy then creates, assembles, and saves the assembly for the three bar slider. ThreeBarSliderDlg, MakeLink, MakePin, and MakeAssy are private functions that cannot be called outside of the ThreeBarSlider class. CreateThreeBarSlider is the public function that can be called outside of the ThreeBarSlider class and in turn calls the other functions to run the three bar slider automation routine. Figure 14 shows a three bar slider created using this routine.

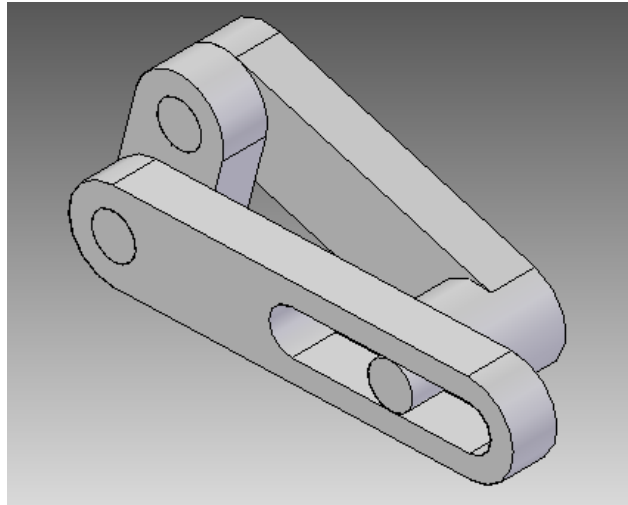


Figure 14: Three bar slider rendered in Solid Edge

3.5.1 ThreeBarSliderDlg

The ThreeBarSliderDlg function handles the interaction between the user and the three bar slider routine. It begins by displaying a description of what the routine will do. The InputDouble function from the SE_Common class is then used to return the user input as doubles. These values are converted to meters using a conversion factor that was previously set by the SelectUnit function from the SE_Common class when it was called from RunSEAutomation. The results are assigned to the variables that define the dimensions of the parts that make up the three bar slider assembly. The dimensions are then checked for any errors. If any errors are detected, an error message will be displayed and the user will be asked again to input the dimensions of the three bar slider. If no errors are detected the lengths of the pins will be calculated, the directory for the parts to be saved in will be created, the file paths for each part and assembly will be assigned, and the function will return. The file and path name section of the function is set up so that, if desired, the file names and directory path could be input by the

user with a few changes to the code. The default names for the six parts of the assembly are illustrated in Figure 15.

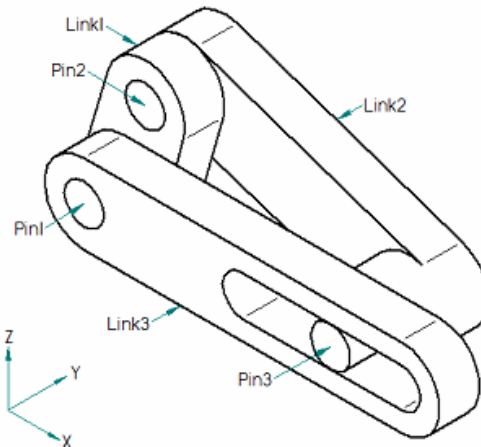


Figure 15: Labeled drawing of the three bar slider assembly

3.5.2 MakePin

The MakePin function takes some of the dimensions and a file path character array assigned by the ThreeBarSliderDlg function, makes a pin in Solid Edge, and saves the part file. The pin is simply an extruded circle. The MakePin function is called three times, once for each of the three pins.

First a new part document is created. A profile, pProfile, is created on the third reference plane, a circle is drawn and extruded. The dimensions for the pins, shown in Figure 16, that were calculated by the ThreeBarSliderDlg function are data members of the ThreeBarSlider class and are used as arguments for this function. Next the WritePhysicalProperties function is called. The part document is then saved using the SaveAs method of the PartDocument object, pPartDocument. Finally, the document is closed using the Close method of the PartDocument object.

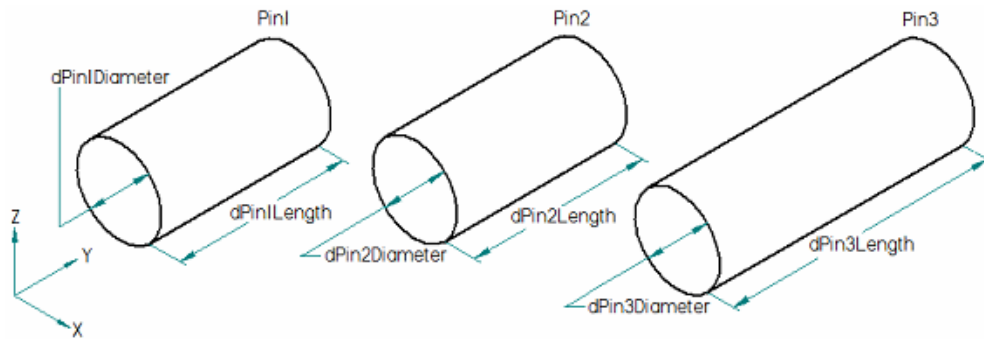


Figure16: Dimensioned drawings of the three pins

3.5.3 MakeLink

The MakeLink function also takes some of the dimensions and a file path character array assigned by the ThreeBarSliderDlg function, makes a link in Solid Edge, and saves the part file. Each of the links is slightly different in construction. Link1 has a profile consisting of two lines and two arcs that is extruded with two hole features added. Link2 is similar to Link1 with a cylindrical extrusion added onto one end prior to the holes. Link3 is similar to Link1 with a slot shaped cutout replacing one of the holes.

First a new part document is created. Figure 17 shows the dimensions for the links that were collected by the ThreeBarSliderDlg function. These dimensions are data members of the ThreeBarSlider class, and are used as arguments for this function. A profile, pProfile, is created on the third reference plane. The coordinates for the base feature, shown in Figure 18, are calculated, and the profile of lines and arcs is drawn, closed, and extruded.

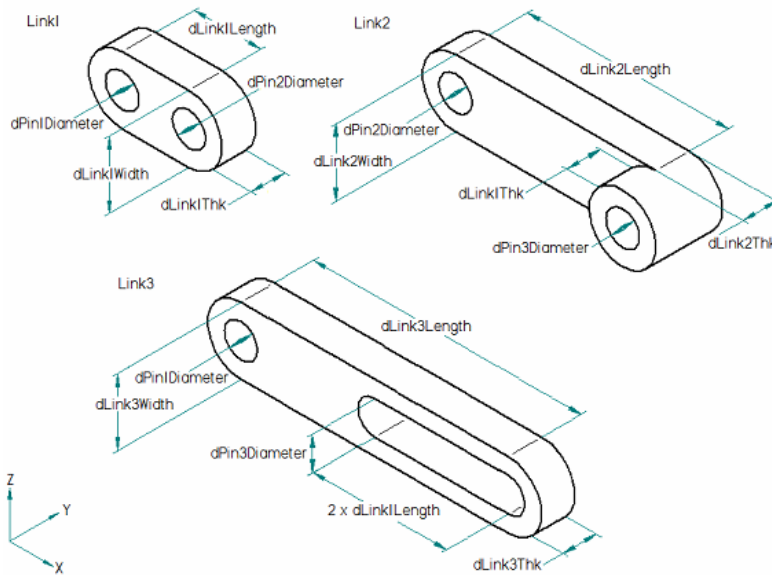


Figure 17: Dimensioned drawings of the three links

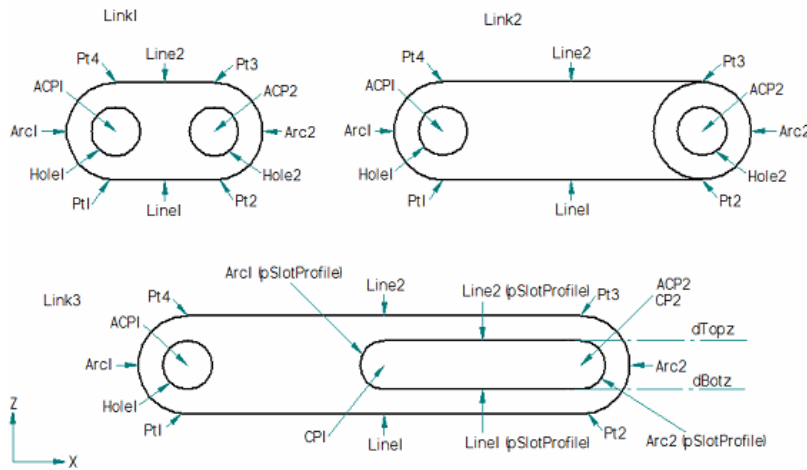


Figure 18: Labeled drawings of the three link profiles

Dimensions for the two holes are assigned. The dimension for Hole1 is used for Link1 and Link3. The dimension for Hole2 is used for Link1 but not for Link3. The function must then determine whether or not the link being drawn is Link2. If Link2 is being drawn, the cylindrical extrusion must be added prior to the holes. The extrusion is created by making a new profile,

Automation of Solid Edge Using External Clients Written in C++

pProfile2, drawing a circle, and using the AddFinite method of the ExtrudedProtrusions object. The dimensions for the holes are then reassigned to be used for Link2.

Next, the first hole, Hole1, needs to be created regardless of which link is being drawn. A new profile, pHoleProfile, is created as well as Holes2d, Hole2d, HoleDataCollection, and HoleData objects. The actual hole feature is created using the AddThroughAll method of the Holes object, pHoles.

Now that Hole1 has been made, the function determines whether or not the link being drawn is Link3. If Link3 is being drawn, a slot in the link will be made using an extruded cutout. A new profile, pSlotProfile, is created on the third reference plane. Additional coordinates, shown in Figure 18, are calculated and the profile of the slot is drawn and closed. The extruded cutout is created by using the AddThroughAll method of the ExtrudedCutouts object, which is exposed by using the ExtrudedCutouts property of the Model object, pModel. At this point Link3 is complete.

If the link being drawn is not Link3, the second hole, Hole2, needs to be created. This hole is created just like Hole1. Finally, the part document is saved, closed, and the function returns.

3.5.4 MakeAssy

The MakeAssy function is called after all the pin and link documents are created. First a new assembly document is created. Several safearrays are then created and assigned a NULL value. This is necessary because Solid Edge API methods will later be used to put elements into the safearrays. The smart pointer pOccurrences is then defined using the Occurrences property of the

AssemblyDocument object. An occurrence object for Link3, pLink3Occurrence, is created using the AddByFilename method of pOccurrences. An occurrence in an assembly file is a part or a subassembly. To create an assembly, an occurrence for each part or subassembly must be created.

Now that the occurrence for Link3 has been created, specific information about this part must be obtained to create the relationships between it and the parts that it connects with. The information collected will be smart pointers to faces, and safearrays containing coordinates that exist on some of those faces. The faces, shown in Figure 19, will be used in the process to create all of the part relations. The safearrays won't be needed for the axial relations, but they will be used to make the planar and tangential relations.

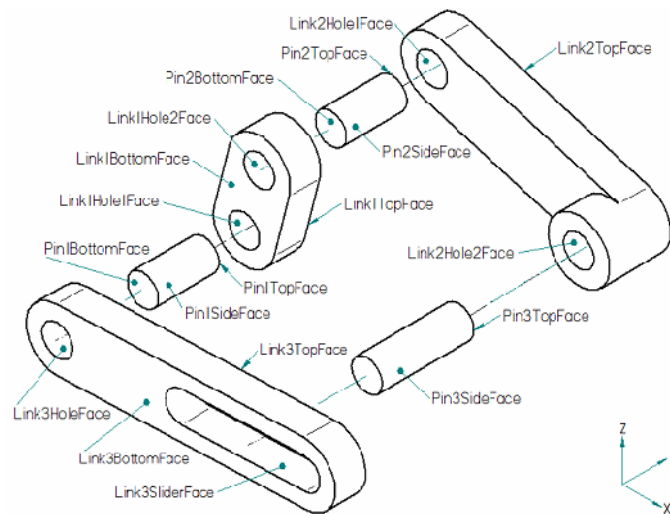


Figure 19: Labeled exploded drawing of the three bar slider assembly

The smart pointers pModel and pExtrudedProtrusion are assigned to the Model and ExtrudedProtrusion objects belonging to Link3. These generically named smart pointers will be reused for the other five parts. The smart pointer pLink3BottomFace is defined as the bottom

Automation of Solid Edge Using External Clients Written in C++

cap of Link3's extruded protrusion. In this case, the top faces of all of the extrusions in these parts are the faces that have the greatest y value in the x-z plane. The bottom faces are on the opposite side. The GetParamRange method of the Face object is used to get a parameter range for Link3's bottom face. The parameter range is written to the aMinParam and aMaxParam safearrays. The aMinParam safearray is then used for the GetPointAtParam method, which gets a point at that parameter and then writes the coordinates of that point to the aLink3BottomFacePoint safearray. Link3's top face is then defined using the TopCap method and a point on that face is assigned to the aLink3TopFacePoint safearray. The next face needed is one of the planar faces of the extruded cutout. The Item method of the ExtrudedCutouts object (which is exposed by the ExtrudedCutouts property of the Model object, pModel) is used to return a pointer to the extruded cutout feature. The SideFaces property of the ExtrudedCutout object is used to return a pointer to all four of the side faces of the extruded cutout. The Item method of the Faces object is then used to return one of the two planar side faces. The index for this planar side face, in this case 2, was determined using trial and error. The final face required from Link3 is a side face of its hole feature. A smart pointer to the hole object is obtained using the Item method of the Holes object returned by the Holes property of the Model object. Next, the Faces pointer pFaces is returned using the SideFaces property of the hole object, pHole. Finally, the face pointer pLink3HoleFace is defined by using the Item method of the Faces object pFaces.

The next parts to be imported are, in order: Pin1, Link1, Pin2, Link2, and Pin3. Smart pointers are defined for the rest of the faces labeled in Figure. Coordinates are placed in additional

safearrays for the top faces of all the remaining parts, the bottom faces of Pin1, Pin2, and Link1, and the side face of Pin3.

After the faces and coordinates are defined, the references can be created. Each reference is made using the CreateReference method of the AssemblyDocument object. The second argument for the CreateReference method is "const _variant_t &Entity." The entity in this case is a smart pointer to a face. To use the CreateReference method, each face must first be placed inside a _variant_t object. The following line constructs a _variant_t object named tempFace1 and attaches to it the smart pointer pLink3BottomFace:

```
_variant_t tempFace1(pLink3BottomFace, true);
```

The face smart pointer is then detached from the _variant_t object when creating the reference:

```
pLink3BottomFaceReference = pAssemblyDocument->CreateReference(  
    Link3Occurrence, &tempFace1.Detach());
```

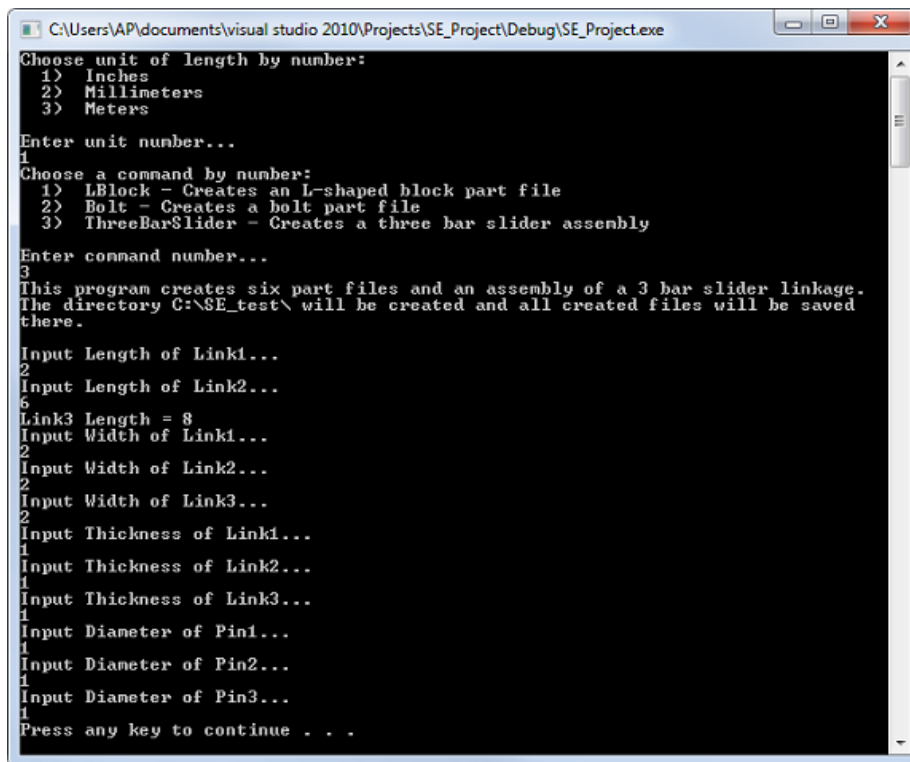
This is repeated for all of the remaining surfaces in Figure 19.

Before the relationships between the parts can be created, it is important to note that every part that was imported programmatically to the assembly was assigned a ground relation. For this assembly it is only desirable for Link3 to be grounded. The other five ground relations must be deleted. A smart pointer to the Relations3d object, pRelations3d, is defined. The second Relation3d object in the document is then defined as the GroundRelation3d smart pointer, pGroundRelation3d. This relation is then deleted using the Delete method of the GroundRelation3d object. The last two steps are placed inside a while loop and executed five times. As the Relation3d objects are indexed as they are imported, the ground relations for Pin1, Link1, Pin2, Link2, and Pin3 are deleted in that order.

Automation of Solid Edge Using External Clients Written in C++

Now the parts can be assembled by creating Relation3d objects. Planar relations are created by using the AddPlanar method of the Relations3d object, axial relations by the AddAxial method, and the tangent relation by the AddTangent method. The Pin parts each have their axial relations locked for one of the links they are connected to. One extra relation (the planar relation between Link3 and Link1) is created for rotational motion to be applied to it in the motion analysis environment.

After all of the relations are created, every safearray is destroyed and the assembly document is saved using the SaveAs member of the AssemblyDocument object. Figure 20 shows a command line window for a completed three bar linkage routine.



```
C:\Users\VAP\documents\visual studio 2010\Projects\SE_Project\Debug\SE_Project.exe
Choose unit of length by number:
1> Inches
2> Millimeters
3> Meters
Enter unit number...
1
Choose a command by number:
1> LBlock - Creates an L-shaped block part file
2> Bolt - Creates a bolt part file
3> ThreeBarSlider - Creates a three bar slider assembly
Enter command number...
3
This program creates six part files and an assembly of a 3 bar slider linkage.
The directory C:\SE_test\ will be created and all created files will be saved
there.
Input Length of Link1...
2
Input Length of Link2...
6
Link3 Length = 8
Input Width of Link1...
2
Input Width of Link2...
2
Input Width of Link3...
2
Input Thickness of Link1...
1
Input Thickness of Link2...
1
Input Thickness of Link3...
1
Input Diameter of Pin1...
1
Input Diameter of Pin2...
1
Input Diameter of Pin3...
1
Press any key to continue . . .
```

Figure 20: A command line window for a completed three bar linkage

3.6 The SE_Common class

The SE_Common.h header file and SE_Common.cpp source file make up the SE_Common class. This class contains a number of functions that are used by the other classes and the function RunSEAutomation: InputInt, InputDouble, SelectUnit, and WritePhysicalProperties. The InputInt and InputDouble functions display the text in the cOutputString argument and return the user input as an integer or a double respectively. SelectUnit asks the user to select from inches, millimeters, or meters as the unit of length for the dimensions that will be input later in the program. As the default unit of length in Solid Edge is the meter, the user input will need to be converted to meters before any part feature are created. An appropriate conversion factor is selected based on the user input and used when assigning values to any dimensions of length. Lastly, WritePhysicalProperties collects the properties of a part file based on a user input density and writes that data to a user named text file.

3.6.1 Physical Properties

The WritePhysicalProperties function first asks the user whether or not they want the physical properties of the part to be calculated. If this is not desired the function returns; otherwise the user is prompted to enter a file name for the output. This file name is then combined with a predetermined directory path and file extension to form a character array for the complete file path.

A character array of document type constant names indexed by their value is created. This array is used to display document type names for an error message. The variables for all of the physical properties are then declared, the safearrays being set to NULL. Density and accuracy

Automation of Solid Edge Using External Clients Written in C++

are inputs to the method that calculates the physical properties. The value for density is input by the user, while the value for accuracy is set to 0.0001.

The function then determines the document type. If the active document is a PartDocument, the physical properties of the part are calculated using the ComputePhysicalProperties method of the Model object. If the active document is not a part file, an error message is displayed and the function returns. The ComputePhysicalProperties method assigns the following values to the previously declared variables: volume, area, mass, center of gravity, center of volume, global moments of inertia, principal moments of inertia, principal axes, radii of gyration, relative accuracy achieved, and status. Status is "an integer value that indicates the status of the physical properties of the model" [Siemens Product Lifecycle Management Software Inc., 2009]. It is not included in the output file, but its value must still be written to a variable for the program to compile and function properly.

After the physical properties are assigned to their corresponding variables, the text file is created and the data is written to it, as shown in Figure 21. Three character arrays and several while loops are used to simplify the process of formatting of the data. After all of the data is written, the text file is closed, all of the safearrays are destroyed, and the function returns. The command line window used to create Figure 21 is shown in Figure 22.

```
LBlock Properties.txt - Notepad
File Edit Format View Help
Solid Edge ST2 Physical Properties Analysis
Requested Accuracy = 0.0001
Relative Accuracy Achieved = 0.0001
Density = 1 kg/m^3
Volume = 9.5 m^3
Area = 82.375 m^2
Mass = 9.5 kg

Center of Mass coordinates (m):
x = 0.493421
y = -4
z = 0.993421

Center of volume coordinates (m):
x = 0.493421
y = -4
z = 0.993421

Mass Moments of Inertia: (kg-m^2)
Ixx = 220.74
Iyy = 23.5208
Izz = 208.115
Ixy = -18.75
Ixz = 1.61719
Iyz = -37.75

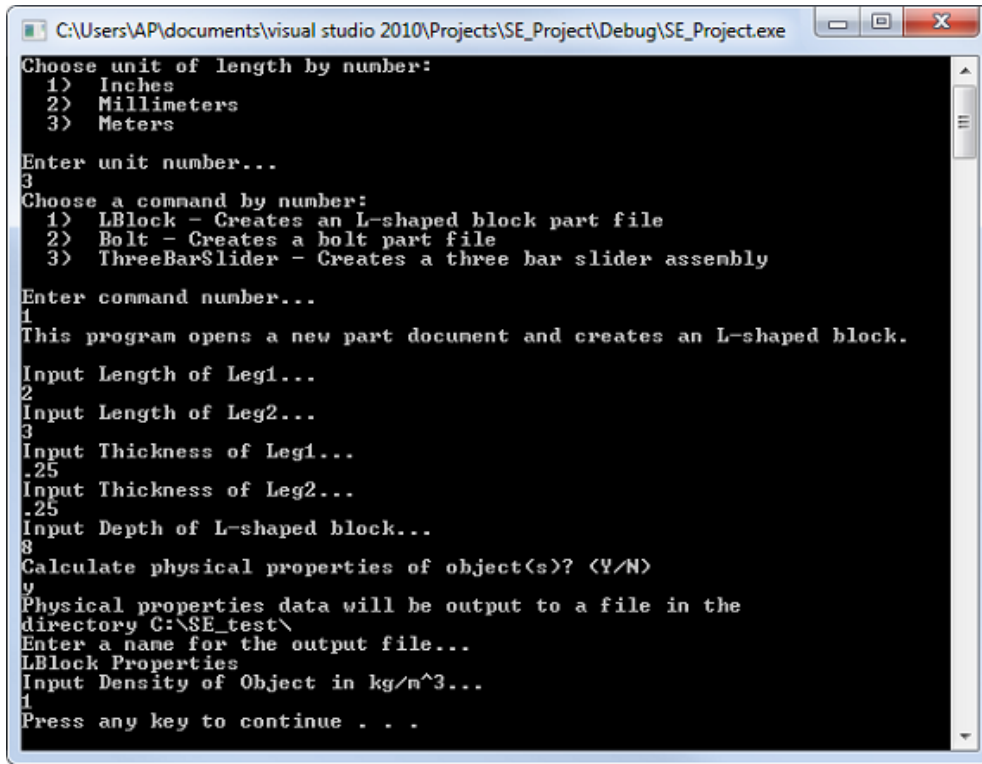
Principal Moments of Inertia: (kg-m^2)
I1 = 60.7028
I2 = 52.463
I3 = 11.8325

Orientation of Principal Axes:
1x = 0.91517
1y = 0
1z = 0.403067
2x = -0.403067
2y = 0
2z = 0.91517
3x = -0
3y = -1
3z = -0

Radii of Gyration (m):
K1 = 2.5278
K2 = 2.34998
K3 = 1.11603
```

Figure 21: Physical properties of an L-shaped block output to a text file

Automation of Solid Edge Using External Clients Written in C++



```
C:\Users\AP\documents\visual studio 2010\Projects\SE_Project\Debug\SE_Project.exe
Choose unit of length by number:
1) Inches
2) Millimeters
3) Meters
Enter unit number...
3
Choose a command by number:
1) LBlock - Creates an L-shaped block part file
2) Bolt - Creates a bolt part file
3) ThreeBarSlider - Creates a three bar slider assembly
Enter command number...
1
This program opens a new part document and creates an L-shaped block.
Input Length of Leg1...
2
Input Length of Leg2...
3
Input Thickness of Leg1...
.25
Input Thickness of Leg2...
.25
Input Depth of L-shaped block...
8
Calculate physical properties of object(s)? <Y/N>
y
Physical properties data will be output to a file in the
directory C:\SE_test\
Enter a name for the output file...
LBlock Properties
Input Density of Object in kg/m^3...
1
Press any key to continue . . .
```

Figure 22: The command line window used to create Figure

This concludes the overview of the source files for the SE_Project program. With these files, anyone who has an installation of Solid Edge ST2 or ST3 can create a project in Microsoft Visual C++ Express and compile the source code to produce this program.

4. Conclusions

The SE_Project program is able to perform all of the desired tasks as well as allowing the user to select from a list of unit lengths. The program is small, and only requires its exe and an installation of Solid Edge to function. The speeds of the automation routines are only limited by how fast Solid Edge can execute the commands. The source code is relatively simple and can be easily modified or expanded upon. Alternatively, instead of having one program with the option

of running three different automation routines, SE_Project could easily be split into three different programs.

In addition to physical properties analysis, the automation of both kinematic analysis and finite element analysis (FEA) were investigated for possible inclusion to the SE_Project. Animation of an assembly in Solid Edge can be accomplished by using motors or the Intellimotion builder. There is currently no API for either of these methods, so automating a kinematic analysis was not possible. However, the constraints in the three bar slider case discussed previously were set up so that a manual kinematic analysis can easily be done on the assembly.

To perform a finite element analysis, users can use the Simulation Express command in the Tools toolbar, or the commands in the Simulation toolbar. There is no API for Simulation Express, but there is an API for the commands in the Simulation toolbar. The first step is to create an FEA study. This is done by using the AddStudy method of the StudyOwner object, which is returned by the StudyOwner property of the PartDocument or AssemblyDocument objects. However, there is no StudyOwner object in a freshly created part or assembly and there are no methods in the API to create a StudyOwner object. The only way to create a StudyOwner object is to manually create a new study in Solid Edge. As a result, the creation or modification of FEA studies can only be automated for parts that already contain a FEA study.

There are several options for creating Solid Edge automation programs. First, programs can be written as external clients or add-ins. Furthermore, the programs can be written in any COM enabled language; mainly VB.NET, C#, and C++. Generally, someone new to Solid Edge

Automation of Solid Edge Using External Clients Written in C++

programming would find it easiest to learn by using whatever methods they are most familiar with. Regardless of the chosen method of Solid Edge automation, the Solid Edge SDK and the object browser in Microsoft Visual Studio are essential for writing Solid Edge programs. The Solid Edge Spy application can also be very useful for observing the inner workings of Solid Edge.

The base program code for the external clients presented in this paper and [Putman, 2011] is objectively less complex than the code presented in Jason Newell's article, Solid Edge ST Addins – Part I, and his Solid Edge AddIn Wizard [Newell, 2009] and [Newell, 2006]. The code for the wizard itself is larger than the code for the entire SE_Project program. Because of this added complexity, it is easier for someone without much programming experience to write an external client rather than an add-in for Solid Edge. However, some applications of Solid Edge automation are much better suited to add-ins rather than external clients, such as the file importing and exporting performed by the SYCODE add-ins [SYCODE, n.d.].

For someone with no experience in C++ COM programming, the creation of a Solid Edge add-in or external client can be a daunting task. Documentation, examples, and community support are much more limited for C++ than for C# and especially for VB.NET. In addition, C++ is notorious for being more difficult and time-consuming to learn and write than VB.NET and C#. However, C++ is also known for being a more flexible and powerful language. For these reasons, a beginner may consider starting off with VB.NET or C# to learn Solid Edge programming before progressing to C++. Even so, this paper and the tutorials in [Putman, 2011] should give a beginner a good head start in C++ Solid Edge programming.

References

Cope, J. N. (2010), InspectionXpert for Solid Edge [Software], Available from <http://www.inspectionxpert.com/Products/ForSolidEdge/tabid/79/language/en-US/Default.aspx>

Electronic Data Services (2004), Solid Edge Version 15 Programmer's Guide. Available from <http://support.ugs.com/docs/se/v15/ProgGuide.pdf>

IngeneaSoft (2010), Border Control [Software], Available from <http://www.ingeneasoft.com/products/solid-edge-control-utilities/border-control/>

IngeneaSoft (2010), File Control [Software], Available from <http://www.ingeneasoft.com/products/solid-edge-control-utilities/file-control/>

Newell, J. (2009, February 22), Solid Edge ST Addins – Part I, Retrieved from <http://www.codeproject.com/KB/COM/sestaddin1.aspx>

Newell, J. (n.d.), Solid Edge Spy v1.0 [Software], Retrieved from <http://www.jasonnewell.net/Products/SolidEdgeSpy.aspx>

Newell, J. (2006, August 21), Solid Edge AddIn Wizard for Visual Studio .NET 2005. Retrieved from <http://www.jasonnewell.net/Products/SolidEdgeAddinWizard.aspx>

Putman, Aaron Charles (2011), Automation of Solid Edge ST2 Using External Clients Written in C++, Master of Engineering Report, Mechanical and Aeronautical Engineering Department, Clarkson University, Potsdam, NY 13699.

Siemens Product Lifecycle Management Software Inc. (2008), Solid Edge ST Programmer's Guide. Available from <http://support.ugs.com/docs/se/v100/mu28000.pdf>

Siemens Product Lifecycle Management Software Inc. (2009), Solid Edge ST2 SDK.

SYCODE. (n.d.), Solid Edge Add-ins [Software], Available from http://www.sycode.com/products/solid_edge/index.htm