Parasolid V13.0

Getting Started With Parasolid

June 2001

Important Note

Parker's House
46 Regent Street
Cambridge CB2 1DP
UK
Tel: +44 (0)1223 371555
Fax: +44 (0)1223 316931
email: ps-support@ugs.com
Web: www.parasolid.com

Trademarks

Parasolid is a trademark of Unigraphics Solutions Inc.

HP and HP-UX are registered trademarks of Hewlett-Packard Co.

SPARCstation and Solaris are trademarks of Sun Microsystems, Inc.

Alpha AXP and VMS are trademarks of Digital Equipment Corp.

IBM, RISC System/6000 and AIX are trademarks of International Business Machines Corp.

OSF is a registered trademark of Open Software Foundation, Inc.

UNIX is a registered trademark of UNIX Systems Laboratories, Inc.

Microsoft Visual Studio, Visual C/C++ and Window NT are either registered trademarks or trademarks of Microsoft Corp.

Intel is a registered trademark of Intel Corp.

Silicon Graphics is a registered trademark, and IRIX a trademark, of Silicon Graphics, Inc.

Table of Contents

Introduction 1

Welcome to the Parasolid Getting Started Guide. This quick-start guide will get you working with Parasolid as simply and efficiently as possible. If you are a newcomer to Parasolid, it explains the steps you need to complete in order to implement Parasolid in your application. It also touches on other design considerations you need to think about, though you should refer to the full Parasolid documentation set for more details on these.

When you begin working with a system as complex as Parasolid, it can be difficult finding out what your first steps should be. This is equally true if you are developing a Parasolid-powered application from scratch, or if you are integrating Parasolid into an existing application. This guide shows you just what those first steps should be, and points you to code examples supplied on the CD so that you can get started as quickly as possible.

The contents of the rest of the guide are as follows:

- Chapter 2, “Application Design” describes the design of a typical Parasolid-enabled application, and introduces you to the various components in a typical system.
- Chapter 3, “Supplying A Frustrum” describes how to supply a frustrum. The frustrum is the part of your application that interfaces with Parasolid in order to provide file, memory, and graphics handling capabilities.
- Chapter 4, “The Example Application” describes a real frustrum implementation that is available with the Parasolid release.
- Appendix A, “Further Implementation Decisions” gives an overview of non-essential functionality that you might want to include in your frustrum.
- Appendix B, “Using the PK Interface” gives you an introduction to calling Parasolid functions for modeling part data.

Please remember that this guide is intended as an introduction to Parasolid. It tells you the minimum you *need* to know in order to get up and running. It does not mention everything you might *like* to know about, and it may not cover all the issues that are relevant to your particular application. You should refer to the full Parasolid documentation set if you need more information.

1.1 Some assumptions

The frustrum code described in this guide addresses issues that are most relevant if you are developing applications for use on Windows NT or Windows 2000. If your application runs on a UNIX platform, you can find source code for a

platform-independent frustrum on the Parasolid Release CD. See the Parasolid *Release Notes* for more details.

Application Design 2

When integrating Parasolid into your product, some of the design decisions you need to make are constrained by the requirements of Parasolid itself. Your ultimate aim is to make calls to Parasolid routines from within your own application code in order to perform solid modeling operations. In order to do this, you need to supply code that ensures that your application and Parasolid can interact correctly. This chapter gives you an overview of the code you need to supply before you can start making calls to Parasolid routines from your own code.

Figure 2–1 shows how Parasolid interacts with a typical application:

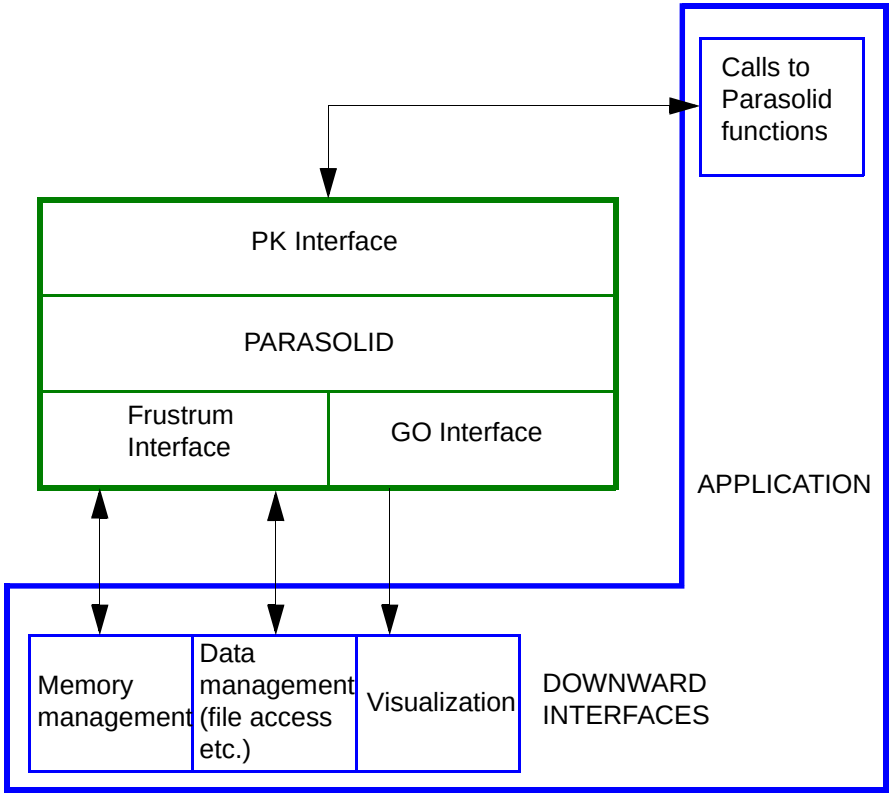

Figure 2–1 How Parasolid interacts with a typical application

The code that you need to supply for integration with Parasolid falls into two categories:

- Downward interface code.
- Calls to Parasolid functions.

2.1 Downward interfaces

Your application controls all interaction between Parasolid and the operating system. The parts of your application that deal with this are referred to generically as *downward interfaces*. You usually need to have two parts to your downward interface:

- The frustrum
- Graphical Output (GO) – required for displaying models

Because Parasolid itself makes calls to routines in these downward interfaces, they need to be registered with Parasolid before your application can call any Parasolid functions.

Note: In practice, GO functions are implemented as part of the frustrum. You can think of them as a separate downward interface to the extent that: (a) they are not needed if you do not need to deal with model display; and (b) they require use of a separate graphics library.

2.1.1 The frustrum

The *frustrum* is a suite of functions that deals with

- **File handling:** Saving and retrieving Parasolid part files and other data.
- **Memory management:** Allocating memory for internal calculations and data structure storage.

2.1.2 Graphical output

If your application needs to display models – whether by rendering them on screen, or printing them to a plotter or laser printer – you must provide GO functions for Parasolid to use.

Parasolid calls the GO functions whenever your application calls a Parasolid rendering function in order to draw one or more parts. The GO functions encapsulate the graphical information output by Parasolid, and pass it to a graphics library (that you also provide) in order to render the image.

You can write the graphics library yourself, or you can use a third party library such as OpenGL or DirectX.

2.2 Calls to Parasolid functions

Parasolid is supplied as a library of functions and is designed as a toolkit of component software that you can embed into your product. The Parasolid API is called the PK. The PK contains functions for:

- Building, modifying, and combining parts
- Finding out information about the properties of parts (such as mass, geometric information, or rendering information)
- Functions for saving parts, and loading parts back into Parasolid

See Appendix B, “Using the PK Interface”, for an introduction to using functions in the PK.

2.2.1 Managing Parasolid errors

Your application must be able to handle errors returned by Parasolid. Parasolid provides two different strategies for handling errors, so that you can choose the strategy that best matches the way that your application handles other errors. See Section 3.7, “Error handling”, for more details.

Supplying A Frustrum 3

3.1 Introduction

As described in Chapter 2, “Application Design”, you need to supply frustrum code to Parasolid in order to perform the following tasks:

- Frustrum control
- File (part data) handling
- Memory management
- Graphical output

This section gives you an overview of what you need to supply in order to accomplish these tasks.

3.2 What code do you need to supply?

This section outlines the frustrum functions that you need to supply in order to build a working frustrum. Some functions are optional; what you need to supply depends on the requred functionality of your application.

The use of the frustrum functions is explained in more detail in Section 3.4, “File handling”, Section 3.5, “Memory management”, and Section 3.6, “Graphical output” later on in this chapter.

Note: Frustrum functions are referred to throughout this guide using the field names below. You can decide on the actual function names yourself, however, since the function names you choose are mapped onto the field names when the frustrum is registered.

3.2.1 Required functionality

You must provide the following frustrum functions:

Frustrum control	
FSTART	start up the frustrum
FSTOP	shut down the frustrum

File (part data) handling (see Section 3.4, “File handling”)	
FFOPRD	open all guises of file (except rollback) for reading
FFOPWR	open all guises of file (except rollback) for writing
FFREAD	read from file
FFWRIT	write to file
FFCLOS	close file

Memory handling (see Section 3.5, “Memory management”)	
FMALLO	allocate virtual memory
FMFREE	free virtual memory

3.2.2 Optional functionality

You can choose whether to implement the following frustrum functions. If you do not, they should be left as NULL. When you initialize the frustrum, all available functions are set to NULL, so you can effectively ignore any functions listed here that you don’t require.

Frustrum control	
FABORT	called at the end of an aborted kernel operation
FTMKEY	returns sample name keys (used for testing FFOPRD and FFOPWR)

Graphical output (see Section 3.6, “Graphical output”)	
GOOPSG	open hierarchical segment
GOSGMT	output non hierarchical segment
GOCLSG	close hierarchical segment

3.3 Registering the frustrum

Before you can start the modeler and make PK calls, you need to register the frustrum functions with Parasolid. To do this, use the PK function PK_SESSION_register_frustrum, which takes a list of pointers to the frustrum functions you have supplied.

You declare and initialize the frustrum in the same way that you set up any option structure in the PK. The following code extract shows how this might be done. For more details, refer to Appendix B, “Using the PK Interface”.

```
PK_SESSION_frustrum_t fru; /* Declare fru as the frustrum */
PK_SESSION_frustrum_o_m( fru );
                    /* Initialize the frustrum functions */

/* Point to each of the frustrum functions you are supplying */

fru.fstart = MyAppStartFrustrum;
fru.fabort = MyAppAbortFrustrum;
fru.goopsg = MyAppOpenSeg;

/* Note: Many functions missing from the list above */

/* Register the frustrum with Parasolid */
PK_SESSION_register_frustrum( &fru );
```

In the code above, note that MyAppStartFrustrum represents a function that you have actually supplied a definition for, and fstart represents a field required by PK_SESSION_register_frustrum.

A specific example showing how the frustrum is registered in the Parasolid Example Application is described in Section 4.3, “The frustrum”.

3.4 File handling

The files that Parasolid uses for storing part data are referred to as ‘transmit’ or ‘XT’ files. In order to transfer data through the frustrum, you need to write to and read from these files. Parasolid can also use several other file types, as described in Section 3.4.2, “File types and file extensions”. You need to decide the format and location of these files in order to write your frustrum functions.

3.4.1 Structuring the file system for your application

You can configure Parasolid to handle many different kinds of archiving systems, such as:

- **A controlled directory structure**, in which part files are saved as individual files on the host computer, or
- **A database** that integrates Parasolid part files with application-specific information.

Your application should provide the following functionality for managing the files generated and required by Parasolid.

- Deleting parts from an archive.
- Keeping a record of, and listing, the keys used to archive parts, in the current session and in previous sessions.

The frustrum is passed a *key* (a text string) that identifies the part. This key can be used as a filename or as an index into the database, depending on the archiving system you have chosen.

3.4.2 File types and file extensions

Parasolid uses several file types, each of which can be written in one of the following formats:

- Neutral binary (recommended)
- Machine-dependent binary
- Text

At a minimum, your application must support Transmit files. Depending on its functionality, it may also need to deal with some of the other file types described here.

Each file type has a recommended file extension that is also dependent on the file system you are writing to or reading from, as shown in the table below. Note that file extensions ending in `_t` or `_txt` denote text-based formats, and extensions ending in `_b` or `_bin` denote binary-based formats.

File type	UNIX/NTFS extensions	FAT extensions	Comment
Transmit (Part)	`.xmt_txt` `.xmt_bin`	`.x_t` `.x_b`	Transmit (or XT) files are used to store part or assembly data and are often used for transferring data between Parasolid-powered systems.
Schema	`.sch_txt` `.sch_bin`	`.s_t` `.s_b`	Parasolid uses schema files to read and write data from and to previous versions of Parasolid.
Journal	`.jnl_txt` `.jnl_bin`	`.j_t` `.j_b`	Journal files are used to keep a record of all of the commands issued to Parasolid within a session. Journal files can be used for debugging.
Snapshot	`.snp_txt` `.snp_bin`	`.n_t` `.n_b`	A snapshot file is a memory dump of a Parasolid session. These files are very rarely used, but are sometimes useful for reproducing faults.

File type	UNIX/NTFS extensions	FAT extensions	Comment
Partition	`.xmp_txt` `.xmp_bin`	`.p_t` `.p_b`	Partitions enable related parts to be archived as a single item. Rollback information allows your application to return a Parasolid session to an earlier state. If your application uses either of these mechanisms, the relevant data is written to a partition file. See Chapter A, “Further Implementation Decisions” for more information.
Delta transmit	`.xmd_txt` `.xmd_bin`	`.d_t` `.d_b`	Delta files are used within the rollback mechanism and are controlled by a separate suite of functions called the *delta frustrum*. Delta transmit files are an aggregate of all existing delta files, and their creation is controlled by the frustrum. See Chapter A, “Further Implementation Decisions” for more information.

The file handling functions in your frustrum must handle all the file types that you decide to support, adding the appropriate extensions. These functions may also test whether a file resides on a DOS style FAT device or a long filename NTFS type device before adding the extension, unless you decide to support only the FAT extensions, regardless of the filesystem. If you support both FAT and NTFS extensions, files can simply be renamed when transferring between the different systems. An example that shows how extensions can be added to filenames is described in Section 4.4.2, “File extensions” in Chapter 4, “The Example Application”.

3.4.3 Example file handling code

Section 4.4, “File handling” gives details about how to implement file handling in the frustrum.

3.5 Memory management

You need to supply two memory management functions in the frustrum that allow Parasolid to allocate and free memory for its use. These functions control the way that memory is allocated to and freed for use in:

- Internal calculations
- Data structure storage

The memory management functions are FMALLO (allocate virtual memory) and FMFREE (free virtual memory).These functions map closely to the C functions malloc and free, and any function definitions you supply should be type-compatible with malloc and free.

You could consider implementing some buffering in order to improve performance compared to the standard functions. For example, with suitable definitions for FMALLO and FMFREE:

- Small amounts of memory (around 0.1 MBytes) could be served by a pool of memory set up by FSTART and managed by your application.
- Large amounts of memory (e.g. 1MB or more) could be requested from the operating system.

The memory management functions are registered when registering the frustrum.

A specific example showing how memory management functions are used in the Parasolid Example Application is described in Section 4.5, “Memory management”.

3.6 Graphical output

If your application needs to display parts – whether by rendering them on screen, or printing them to a plotter or laser printer – you need to do three things:

- Call PK rendering functions to obtain the necessary information from Parasolid.
- Supply a set of functions known as the GO (Graphical Output) interface to interpret that information.
- Supply a graphics library that is used to render the parts on the appropriate device.

When a call is made to one of the PK rendering functions, the graphical data output by Parasolid is passed to the GO functions, which catch and interpret the data, filter it, and use the functionality of the graphics library to render the parts on the appropriate device.

3.6.1 Calling PK rendering functions

In order to render part data, your application needs to call a PK rendering function. There are three such functions available:

Function	Description
PK_GEOM_render_line	Renders a geometric entity as a wire frame drawing that is independent of the view required.
PK_TOPOL_render_line	Renders a list of topological entities as either: ■ A view independent wire frame drawing ■ A view dependent wire frame drawing ■ A hidden line drawing
PK_TOPOL_render_facet	Generates a faceted representation of topological entities.

Each of these functions uses the supplied GO functions to pass graphical data to the appropriate graphics library.

The PK outputs graphical information in the form of *segments*. These correspond to identifiable portions of the model being rendered, and can be curves or facets (depending on the PK function used). There are two types of segments:

- Single-level segments, which contain data describing a curve or facet to be drawn.
- Hierarchical segments, which usually contain other hierarchical or single-level segments.

Note: It is possible to generate faceted information without using the GO, by using the function PK_TOPOL_facet. This approach may be useful, for instance, if your application needs to perform many calculations on a faceted representation of the model.

3.6.2 Supplying GO functions

Parasolid passes segment data to the GO functions. These functions use the supplied graphics libraries and may generate, for example, screen pictures, plot files and laser print files.

There are three GO functions in the frustrum. They all take the same arguments, but interpret them in different ways.

Function	Description
GOOPSG	Open a hierarchical segment.
GOSGMT	Output a single-level segment.
GOCLSG	Close a hierarchical segment.

GO functions are registered together with the other frustrum functions. See the Section 3.3, “Registering the frustrum” for details.

A specific example showing how GO functions are defined in the Parasolid Example Application is described in Section 4.6, “Graphics”.

3.6.3 Choosing which graphics libraries to use

Parasolid’s graphical data output can be used in conjunction with many different graphical libraries. You must supply a graphics library for every device you want to use for rendering output. If your application renders to several different devices (for instance, it might render to the screen, and print paper copy), then you may need to supply several different libraries. You can write your own graphics libraries if you wish, or you can use third party libraries. The GOSGMT function calls the appropriate libraries in order to perform the necessary rendering.

3.7 Error handling

Parasolid can raise errors in a number of circumstances. For example, if your application passes incorrect data to a PK function, or a PK function fails in some way, then Parasolid raises an error. Your application needs to be able to handle these sorts of errors. This section describes how you go about doing this.

3.7.1 Choosing an error handling strategy

There are two approaches to handling errors from Parasolid.

- Your application can supply and register an error handler with Parasolid. This error handler is invoked when a PK function is about to return a failure. The error handler then takes the appropriate action.
- Your application can allow the PK function to complete, examine the returned value, and take appropriate action, usually by means of a wrapper function.

The strategy you choose probably depends on how errors are handled by the rest of your application. You are strongly recommended not to mix the two strategies, if this is possible.

Whichever strategy you choose, the same action needs to be taken. This action depends on the severity of the error, as supplied in the error call. See Section 3.7.3, “What to do when an error occurs” for more details.

Registering an error handler

You use PK_ERROR_register_callbacks to register an error handler with Parasolid. When a PK function is about to return an error, the error handler is passed a standard structure (PK_ERROR_sf_t) containing:

- The function returning the error
- The error severity
- The specific error code
- Other information to assist with isolating the problem

This data is always stored, and can be retrieved again if necessary using PK_ERROR_ask_last. The data for the last error can be cleared using PK_ERROR_clear_last.

You can see how an error handler is registered in the Parasolid Example Application in Section 4.7, “Error handling” in Chapter 4, “The Example Application”.

3.7.2 Handling non-zero error codes

Every PK function returns an *error code* that indicates the result of the operation. If this value is zero – PK_ERROR_no_errors – the operation has completed successfully and your application can proceed.

However, if this value is non-zero, then the operation has failed to complete. Each numerical value indicates a predictable error, and your application must handle it appropriately:

- If your application uses a registered error handler, then the error handler must examine each return value and take suitable action.
- If your application does not use a registered error handler, then you must ensure the application itself examines each return value and takes appropriate action.

Note: Do not assume that a a zero error code means that the operation has completed exactly as you intended. A zero error code simply means that the function completed. It is still possible that the operation did not complete in the way you expected, and, in particular, many PK functions return error tokens that indicate specific problems in the values passed to the function or the results produced.

3.7.3 What to do when an error occurs

Whether you register an error handler or decide to handle each error explicitly in your application code, the action that needs to be taken when an error occurs is much the same.

The three error severity levels that your application has to manage are:

Severity	Current State	Required Action
Fatal	Modeler memory has been corrupted; rolling back, if implemented, will not be effective. You may not be able to restart Parasolid	Your application should shut down Parasolid
Serious	The parts loaded in Parasolid may be corrupt	If rollback is implemented, your application should roll back to a valid state
Mild	The operation failed; the parts involved were not corrupted	The application can continue with any Parasolid operation

3.8 Starting and stopping a Parasolid session

You use the function PK_SESSION_start to start the Parasolid modeler. This takes an options structure that you can use to specify session specific options, as described in Section A.4, “Session parameters”.

You use the function PK_SESSION_stop to stop the Parasolid modeler.

Warning: You **must** register the frustrum before attempting to start a Parasolid session.

The Example Application 4

To complement the overview of frustrum implementation in Chapter 3, “Supplying A Frustrum”, this chapter describes a simple Parasolid-powered application whose source code you are free to examine. The Parasolid Example Application described here is provided on the Parasolid CD for PC platforms.

The Example Application is a Windows NT application that demonstrates how to combine the various components necessary into a working application. As well as providing a simple frustrum, it shows you how to make calls to PK functions, and can be used in a limited way to prototype PK function calls. Because it is designed specifically to demonstrate all the components of a Parasolid-powered system, it may not necessarily reflect the optimal design for your application.

Note: PS/Workshop, supplied on the Parasolid CD for PC platforms, is a dedicated prototyping application that you can use to test calls to PK code under Windows NT and Windows 2000. The Example Application is provided for demonstration purposes only and should not be used for extensive code prototyping.

The application is written in Microsoft Visual C++ and uses Microsoft Foundation Classes. It was compiled and built using Microsoft Visual C++ Version 6. It uses the OpenGL graphics library for displaying part models.

The compiled application can be run on Windows NT4, Windows 2000, and 95/98.

Full documentation for the Example Application is provided in the manual *Using the Example Application*.

4.1 Building and running the Example Application

The Parasolid CD contains the complete set of files necessary to examine, build, and run the Example Application using Microsoft Visual C++ running under Windows NT 4 or 2000. To build and run the Example Application:

- Place the Parasolid CD into the CD-ROM drive on your computer.
- Use an archiving application such as WinZip to extract the file *X*`:\Example_Application\ExampleApp.zip` to a folder on your local hard disk (where *X* represents the name of your CD-ROM drive).
- To load the Example Application into Visual C++, double-click the file `Source\Example App.dsw` in the folder that you extracted application to.
- Press Ctrl+F5 to build the Example Application and run it. Follow any prompts from Visual C++ to build the necessary files.

The Example Application has a single window that contains a menu bar, toolbar, and a viewing area that is initially empty.

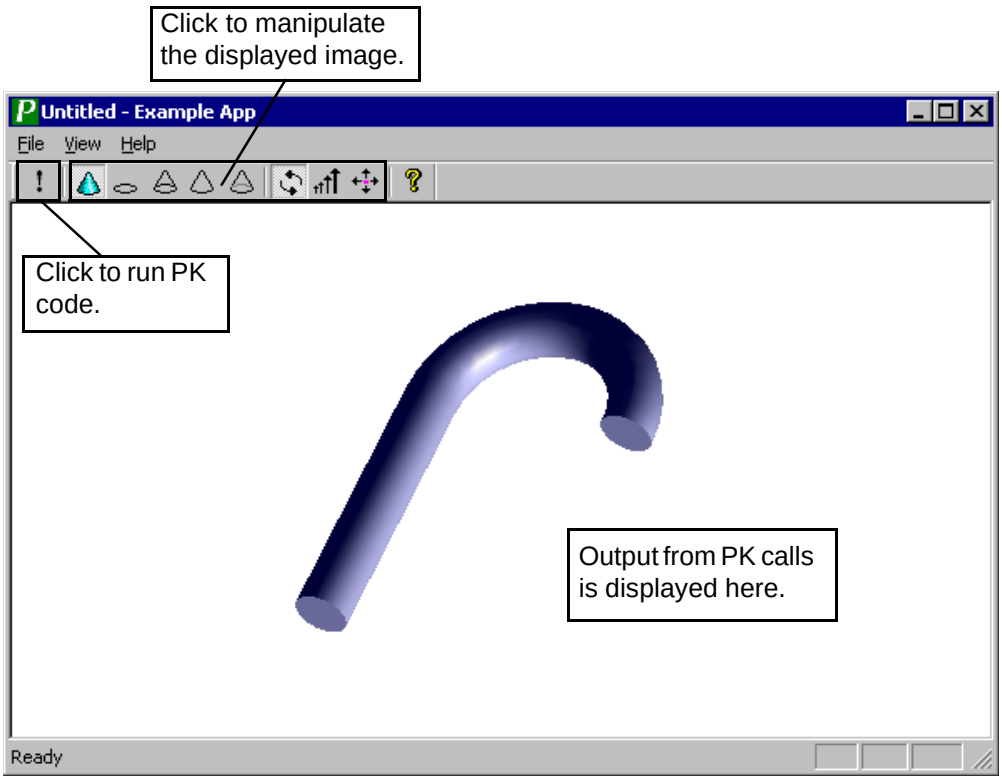

Figure 4–1 The Example Application

If the PK calls in the code you are testing are written as a series of steps, then click ! to execute each step in turn, examining the output along the way. See Section 4.2, “Calling PK functions”, for details about structuring code this way.

4.2 Calling PK functions

To make use of Parasolid functionality in the Example Application, you add PK function calls to the CMyCode class; specifically, to the function CMyCode::RunMyCode. This function is reserved for PK function calls so that you can evaluate Parasolid functionality easily. You can find the definition for this function in the file MyCode.cpp.

You can add code that is structured as a single step, or as multiple steps using a case statement. When running code that contains multiple steps, click the ! button in the Example Application to execute the next step. The following example shows a case statement that consists of two steps: a block is drawn in the first step, and a cylinder is drawn in the second step.

```
int CMyCode::RunMyCode(int step)
{

// The code below is sample code. On exiting the function
// the bodies that exist are drawn

    static PK_BODY_t block;
    static PK_BODY_t cylinder;
    CString text;
    BOOL finished = FALSE;

    switch( step )
    {
    case 1:

/*   Create a block
     Size: 10 x 10 x 10.
     Location & orientation: default (pass NULL
     as pointer to the basis_set argument).
*/

        PK_BODY_create_solid_block
                ( 10.0, 10.0, 10.0, NULL, &block );
        break;

    case 2:

    /* Create a cylinder
    Radius: 2.5
    Height: 20.0
    Location & orientation: default.  */

        PK_BODY_create_solid_cyl
                ( 2.5, 20.0, NULL, &cylinder );
        break;

    default:
        finished = 1;
    }
    return finished;
}
```

The model display is updated in the viewing area of the Example Application whenever:

- CMyCode::RunMyCode completes an additional step (if PK code uses a case statement).
- CMyCode::RunMyCode finishes executing (if PK code doesn't use a case statement).
- You click one of the buttons in the Example Application's toolbar.

When the display is updated, the Example Application checks for new parts in the session and re-facets or re-renders according to the selected display type.

Note: The ability to add PK function calls to the Example Application code is provided for demonstration purposes only. You should use PS/Workshop if you want to prototype Parasolid functionality.

4.3 The frustrum

The Example Application uses a very simple frustrum that is set up and initialized by functions in the CSession class (see the file session.cpp).

The CSession::Start function sets up and registers the frustrum, initializing further functions that are needed for opening, closing, reading and writing files.

The sequence of events involved in setting up and registering the frustrum is as follows:

- A frustrum is declared, and then initialized using a macro.
- All the required frustrum functions are declared.
- The frustrum is registered using PK_SESSION_register_frustrum.

4.4 File handling

File handling in the Example Application is very straightforward. The application does not use its own database; it only uses Parasolid data files when reading and storing part information.

The following frustrum functions are used for file handling:

- OpenReadFrustrumFile (FFOPRD)
- OpenWriteFrustrumFile (FFOPWR)
- CloseFrustrumFile (FFCLOS)
- ReadFromFrustrumFile (FFREAD)
- WriteToFrustrumFile (FFWRIT)

You can find the definitions for all these functions in the file frustrum.cpp, or listed under the Globals definitions in the ClassView tab of the Workspace window in Visual C++.

4.4.1 File types

The Example Application supports a subset of the file types described in the section Section 3.4.2, “File types and file extensions”. In particular, delta files are not supported, because no rollback mechanism has been implemented.

Journal files are supported, but only when journaling is switched on when setting up and starting the modeler, as described in Section A.4, “Session parameters”.

4.4.2 File extensions

The Example Application uses three character extensions for all file types. Attaching the correct file extension for a given type of file is a two-stage process:

- Determine the file type to use, and attach the first part of the file extension.
- Determine the file format to use (text or binary), and attach the last part of the file extension.

These stages are performed by the `filetype_guise_string` and `filetype_format_string` functions, respectively. Both these functions return pointers to the relevant parts of the extension. You can find both of these functions in frustrum.cpp.

Both of these functions are called by the FFOPRD and FFOPRW frustrum functions. Other frustrum functions used for file handling access the pointer values returned by these functions directly.

4.5 Memory management

The memory handling functions supplied to Parasolid are defined in CSession::Start and registered along with other frustrum functions (see the file session.cpp). These functions, GetMemory (FMALLO) and ReturnMemory (FMFREE), enable Parasolid to request and relinquish memory. They make use of the “new” and “delete” operators in C++.

The Example Application manages all the memory associated with entities in the session, freeing memory when it becomes available. However, it does not free any output arrays that have been allocated in CRunMyCode, so you should take care to free memory that is no longer needed when adding functions to CRunMyCode. For example, if your code calls

```
PK_BODY_ask_faces(body, nfaces, faces)
```

then you need to free the returned `faces` array yourself.

See Appendix B, “Using the PK Interface” for information on how to do this.

4.6 Graphics

The Example Application uses the OpenGL graphics library, and makes calls to it when starting up the application and when drawing parts. The display is set up by CExampleAppView::InitOpenGL (in the file OpenGl.cpp), which initializes colors, pixel format, projection matrices, model matrices, and lights.

The relevant graphics functions are registered with Parasolid in the function CExampleAppDoc::OnNewDocument (in the file ‘Example AppDoc.cpp’). This

registers the following functions to frustrum functions that handle graphical output:

- CopenSegment (GOOPSG)
- CcloseSegment (GOCLSG)
- CoutputSegment (GOSGMT)

The definitions of each of these functions can be found in GO.cpp.

As well as opening, closing, and outputting graphical segments supplied by Parasolid, they also set appropriate colors and material properties using various calls to Parasolid and OpenGL. The data is filtered to deal with displaying lines, circles, ellipses and facets separately, and then the appropriate OpenGL function called.

4.7 Error handling

The Example Application takes a simple approach to handling Parasolid errors; a function called CSession::PKerrorHandler is registered with Parasolid as an error handler, and this is called whenever an error is returned.

The error handler takes the character string corresponding to the error and displays this on the screen in a standard Windows message box. No corrective action is performed.

The error handler is defined and registered in the CSession::Start function (session.cpp).

4.8 Starting the Parasolid session

The Parasolid modeler is started towards the end of the call to CSession::Start (session.cpp). This is done using the PK_SESSION_start function, which uses an options structure in the normal way.

If required, specific session options can be set between the macro call and the function call. See Section A.4, “Session parameters” for more details.

Further Implementation Decisions

A

This appendix discusses other functionality that you may want to consider using when integrating your application with Parasolid. None of the issues discussed here are essential in order to get a working frustrum. For more information on any of the functionality discussed here, refer to the full Parasolid documentation.

A.1 RTE and interrupt handling

In addition to an error handler, you can register a signal handler with the operating system to recover from run-time errors and/or user interrupts.

If you don't use a signal handler, then your application will crash when a run-time error occurs.

A.2 Rollback

Parasolid has two types of rollback, both of which are available through the same mechanism.

- Session level rollback allows your application to roll back the contents of the entire session.
- Partition level rollback allows your application to rollback an individual partition independently of other partitions.

If implemented, you can use rollback in your application in several ways. For example:

- To revert to a known state after a failed operation (i.e. as an undo facility).
- To mark the major milestones in the design history of a part within a feature modeling environment.

You could implement partition level rollback so that each partition has only one part, users could make changes to individual parts and roll them back without affecting other parts in the session.

Rollback is enabled by registering a delta frustrum before the session is started. The delta frustrum performs a similar function to the standard frustrum, and specifies how rollback (delta) information is written and read.

A.3 Tracking entities

Your application may need to track various entities within any Parasolid session. For example, tracking is required in order to:

- Update graphical display information
- Track entities in a feature-based system

There are several ways of tracking entities in Parasolid, and you need to decide which approach you want to take. The most straightforward ways of tracking entities are:

- Using tracking information returned in PK function calls
- Using attribute information attached to entities. See Chapter 46, “Attributes”, in the *Functional Description* for details.
- Setting up a bulletin board to record certain events within a Parasolid session. See Chapter 56, “Bulletin Board” in the *Functional Description* for details.

A.4 Session parameters

There are a number of session level parameters that you can set up to alter the behavior for the whole session. Example of session-wide parameters include:

Parameter	Description
Continuity and self intersection checking	These settings affect the level of checking performed by various checking functions, and also control whether G1-discontinuous or self-intersecting geometry can be attached to topology.
General topology	This setting allows general bodies – such as non-manifold or disconnected bodies – to be created from Boolean operations.
Roll forward	This option defines whether a session can be rolled forward at any stage. The setting of this option has an affect on the rollback management strategy your application needs to use.
SMP (Symmetric Multi Processing)	This option enables Parasolid to make best use of machines that have more than one processor.
Journaling	A journal file is a record of all the Parasolid commands that are called within a session. It is mainly useful for debugging purposes.

Session level parameters are set just before the Parasolid session is started. For example, journaling can be switched on for the current session as follows:

```
PK_SESSION_start_o_t options;
PK_SESSION_start_o_m( options );

options.journal_file = "c:\\temp\\test"

PK_SESSION_start( &options );
```

Warning: Journaling is a useful debugging tool, but you should not turn it on by default in your application, as this can adversely affect performance. Journaling is typically used as an internal debugging tool, although if necessary you can include a switch in your application code that lets the user turn journaling on when required.

Using the PK Interface B

B.1 Introduction

The PK interface is a collection of C declarations for tokens, structures and functions. The complete set of definitions are listed in the file ‘parasolid_kernel.h’ on the Parasolid Release CD. This appendix explains some of the conventions used for naming and calling functions in the PK.

B.2 PK interface functions

All PK functions have names of the form PK_<*CLASS*>_<*operation*>, where

- <*CLASS*> denotes the class of entity on which the function can be called, and
- <*operation*> is a verb/noun combination that describes the operation to be performed.

Parasolid entities are organized in an object-oriented class hierarchy such that a function that is called on <*CLASS*> can also be called on all the subclasses of <*CLASS*>. For example, PK_CURVE_ask_fin can be called on any subclass of CURVE, such as CIRCLE, ELLIPSE, or LINE.

Each function has a fixed set of arguments: some are used to supply data, and others are used to return information. Argument types are simple values, arrays, and structures.

You must set each argument explicitly when calling PK functions, rather than omitting them completely. This is made easier by the use of option structures and initialization macros.

B.2.1 Types associated with PK classes

When a PK class has an object belonging to it (which it nearly always has) the name of the C type for that object is of the form PK_<*CLASS*>_t.

A PK class may have other types associated with it – structures or token enumerations – and these have names of the form PK_<*CLASS*>_<*text*>_t. An example of this is PK_SESSION_frustrum_t, which is the type you use when declaring a frustrum. When these types are more directly associated with a particular function their names reflect this, as there are naming conventions for options structures, type classifications, etc.

B.2.2 Using function arguments correctly

A PK function argument is used either to receive information from the application, or to return information to your application, but never both.

For example, given a PK function with the following declaration:

```
PK_ERROR_code_t  PK_THING_do_something
(
/* received arguments */
const PK_THING_t  *in_thing
/* returned arguments */
PK_THING_t *const  out_thing
)
```

You should never call this function as follows, since the results are undefined:

```
PK_THING_do_something(&my_thing, &my_thing);
```

Warning: The order in which entities are returned from a given PK function, and the underlying geometric representations of faces and edges, are **not** guaranteed to be consistent between different versions of Parasolid. Consequently, your application should not depend on either of these things.

B.3 Types of structures

Much of the information passed in PK function code is collected together in related groups, or structures. There are three basic structures that you need to be aware of:

- Options structures, used to pass parameter values in function calls.
- Standard forms, used as templates for classes of information.
- Return structures, used to return values from a function call.

B.3.1 Passing arguments in options structures

Optional arguments and option switches passed to functions are generally collected together in a single structure and passed as one argument, known as an options structure. Option structures are named by adding the string "_o_t" to the function name.

Warning: The first field of any options structure is the version number (`o_t_version`). Your application must never set or alter this value.

Using macros to initialize options structures

Before you can use an options structure, every field in the structure must be given a value. To make this easier, a macro is available for each option structure that sets every field in the structure to a default value.

After calling the macro, you just need to set any fields that you want to use a different value for. Functions themselves do not have a default action; they always reference the given option structure.

To get the macro name for a given option structure replace the "_o_t" string at the end of the structure name with "_o_m".

An example call to a PK function that uses an options structure is shown below:

```
/* Declare entities */
PK_FACE_t    face;
PK_TOPOL_t   topol;
PK_VECTOR_t  point;
point.coord[0] = 5.0;
point.coord[1] = 5.0;
point.coord[2] = 0.0;

/* Declare options structure */
PK_FACE_contains_vectors_o_t  option;

/* Initialize options structure fields */
PK_FACE_contains_vectors_o_m(option);

option.vectors = &point;    /* Override default values for two
fields*/
option.n_vectors = 1;

/* Make the function call */
pk_ifail = PK_FACE_contains_vectors (face, &option, &topol);
/* Test for pk_ifail */
```

B.3.2 Standard form structures

Many classes have a special structure known as a "standard form", that needs to be used when creating instances of that class. You can think of a standard form as a template for given class instance. Standard forms always have names of the form PK_<*CLASS*>_sf_t.

Most entities have associated standard forms that contain the definitions of curves, surfaces, and so on required for those entities (though some, such as bodies and vertices, do not). Some other classes also have an associated standard form (such as PK_AXIS1_sf_t), even though there is no equivalent entity.

The same standard form for a class is used whether the calling function is an input function or an output function. For example, PK_CYL_sf_t is used by both PK_CYL_create and PK_CYL_ask.

The following example shows how a cylinder is created using the standard form for a cylinder:

```
PK_CYL_t my_cylinder;
PK_CYL_sf_t my_cyl_sf;

my_cyl_sf.basis_set.location.coord[0] = 1.0;
my_cyl_sf.basis_set.location.coord[1] = 2.0;
my_cyl_sf.basis_set.location.coord[2] = 3.0;
my_cyl_sf.basis_set.axis.coord[0] = 1.0;
my_cyl_sf.basis_set.axis.coord[1] = 0.0;
my_cyl_sf.basis_set.axis.coord[2] = 0.0;
my_cyl_sf.basis_set.ref_direction.coord[0] = 0.0;
my_cyl_sf.basis_set.ref_direction.coord[1] = 1.0;
my_cyl_sf.basis_set.ref_direction.coord[2] = 0.0;
my_cyl_sf.radius = 5;

PK_CYL_create(&my_cyl_sf, &my_cylinder);
```

B.3.3 Return structures

Some PK functions pass information back to your application using a return structure. Return structures are used when a lot of related information needs to be handed back to your application, such as tracking information describing the changes made to a part as the result of a function call. The names of return structures end in “r_t” – for example, the return structure that returns information about which topology was split, deleted, or created, is called PK_TOPOL_track_r_t.

B.4 Memory management for returned arguments

Returned arguments to PK functions consume variable amounts of data that are returned from the PK as C arrays. The extent to which you need to control memory allocation for these arguments depends on how easy it is to determine how much memory is required for them.

Requirement	Procedure
If the size of the argument is known at compile time.	**Your application declares the space at compile time and passes a pointer to it to Parasolid.** Arguments like this are shown in the PK function headers as, for example, `type name[3]`. You do not need to explicitly free space declared in this way.
If the size of the argument can be determined by your application at run time before making the function call.	**Your application allocates space at run-time and passes a pointer to it to Parasolid.** Arguments like this are shown in the PK function headers as, for example, `type name[]` or `type *const name`. Your application needs to free the space allocated when it is no longer required.
In all other cases Parasolid dynamically allocates space to return the array, using PK_MEMORY_alloc.	**Your application declares a pointer to the returned type, and passes a pointer to this pointer to Parasolid.** Parasolid sets this pointer to point to the returned information. Arguments like this are shown in the PK function headers as, for example, `type **const name`. Your application needs to free the space allocated when it is no longer required. This is described in Section B.4.1, “Memory management functions in the PK”.

B.4.1 Memory management functions in the PK

Your application allocates and frees memory for variable length return arguments using PK_MEMORY functions. These functions ensure that your application can free memory when it has been allocated by Parasolid, and that space is freed consistently regardless of where it was originally allocated. If the functions are not registered, then the default memory allocation and freeing functions for the operating system are used.

- PK_MEMORY_alloc allocates space for a returned argument before calling the PK function. This function is called automatically by Parasolid when allocating space dynamically for memory. Your application could use it for other types of memory.
- Use PK_MEMORY_free to free space used by a returned argument after the PK function has succeeded.

If a PK function fails, and has already allocated space for variable length returns, then Parasolid frees the space by calling the FMFREE function directly.

The example below demonstrates how to use PK_MEMORY_free to free space used by a return argument once it is no longer needed.

```
PK_BODY_t my_body;
int n_faces;
PK_FACE_t *my_faces;

PK_BODY_ask_faces(my_body, &n_faces, &my_faces);
...
...
...
PK_MEMORY_free(my_faces);
```

B.4.2 Return structures containing pointers

In general, whenever the returned argument is a pointer to a pointer, Parasolid allocates space for whatever is pointed to at the end of the line, and your application frees that space when it is finished with, using PK_MEMORY_free. You can spot two levels of indirection like this easily by looking for a declaration of the form:

```
type **const name
```

Sometimes, a PK function returns a pointer to a structure that itself contains a pointer. In this case, it can be harder to spot the second level of indirection. In particular, standard forms (structures that represent the data encapsulated by an object of a particular class) have a fixed size, but they may point to variable length arrays.

For example, PK_BCURVE_ask returns a standard form PK_BCURVE_sf_t which, because it is a fixed size, is declared as:

```
PK_BCURVE_sf_t *const bcurve_sf
```

The structure PK_BCURVE_sf_t contains the field:

```
double *knot
```

and space for the knot vector is allocated by Parasolid.

Most PK functions that return information using return structures have equivalent memory destructing functions to free up all the memory allocated to the structure in one call. The names of these functions end in "_r_f" – for example, the freeing function for the return structure PK_blend_rib_r_t is PK_blend_rib_r_f.

B.4.3 Optional return arguments

You can set some return arguments declared in the form

```
type **const name
```

to NULL in the function call to indicate that this information is not to be returned and no space is to be allocated for it. Such arguments are indicated by the word 'optional' in the function header.

An example of an optional return argument is shown below:

```
PK_ERROR_code_t PK_BODY_ask_faces
(
--- received arguments ---
PK_BODY_t body, --- a body
--- returned arguments ---
int        *const n_faces, --- number of faces (>= 0)
PK_FACE_t **const faces --- faces (optional)
)
```

In order to just return the number of faces, rather than the faces themselves, set the faces argument to NULL in the function call as follows:

```
PK_ERROR_code_t status;
PK_BODY_t body = ...
int  n_faces;
status = PK_BODY_ask_faces (body, &n_faces, NULL);
```

This use of NULL is only allowed where it is explicitly documented. Functions which have option structures never have optional return arguments.

B.5 Example code

The following simple example demonstrates how the PK interface is used. This example sets an attribute on a face, but ignores the possibility of errors.

```
/* Assume a face in a valid model has been selected */
PK_FACE_t    face = ... ;

/* Local variables for attribute code */
PK_ERROR_code_t status;
PK_ATTDEF_t     colour_defn;
PK_ATTRIB_t     colour_attrib;
double          rgb[3];

/* Locate the definition of the   */
/* system defined color attribute */
status = PK_ATTDEF_find("SDL/TYSA_COLOUR", &colour_defn);

/* Create a new attribute of type color */
/* attached to the given face           */
status = PK_ATTRIB_create_empty
               ( face, colour_defn, &colour_attribute );

/* Fill in the fields of the attribute */
/* with the desired values             */
rgb[0] = 0.25; rgb[1] = 0.25; rgb[2] = 0.5;
status = PK_ATTRIB_set_doubles(colour_attrib, 0, 3, rgb);
```

B.6 Integrating with MS Visual Studio

If you are using Microsoft Visual Studio to develop your Parasolid-powered application, you need to integrate Parasolid's DLL, LIB, and header files correctly into your project environment. This is done as follows:

- Include the file `pskernel.dll` in either the same folder as your application executable, or somewhere on your PATH.
- Add `#include <parasolid_kernel.h>` to any source code files that call PK functionality.
- Add the `pskernel.lib` object library to the MS Visual C++ environment, as described in Section B.6.1.
- Add Parasolid header files to the MS Visual C++ environment as an additional include directory, as described in Section B.6.2.

B.6.1 Adding pskernel.lib to MS Visual C++

To add `pskernel.lib` into the MS Visual C++ environment, choose **Project > Settings** to display the Project Settings dialog, and follow the instructions in Figure B–1.

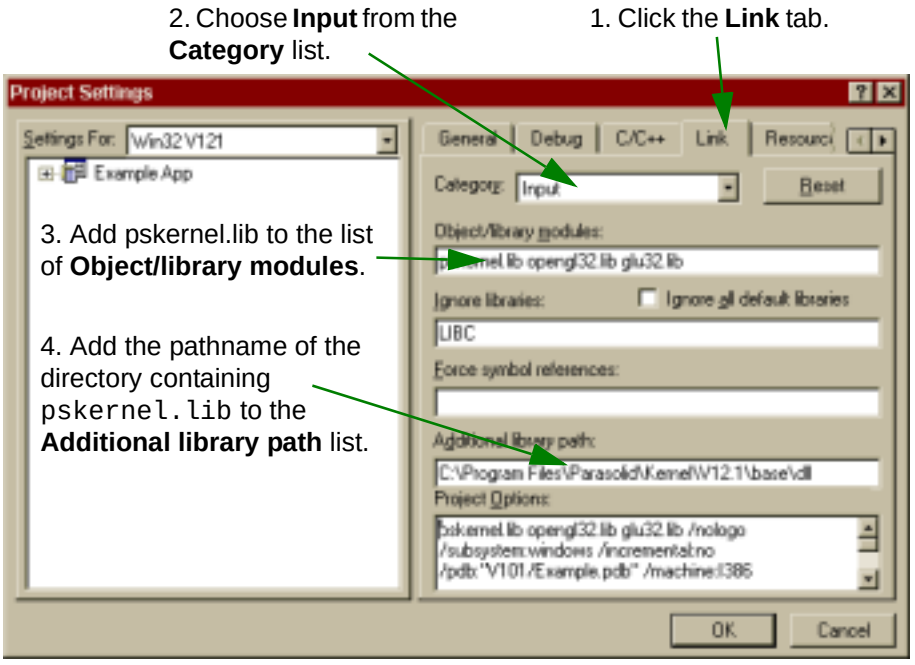

Figure B–1 Adding pskernel.lib to the MS Visual Studio environment

B.6.2 Specifying an additional include directory

You can add one or more directories to the list of directories that are searched for include files. Choose **Project > Settings** to display the Project Settings dialog, and follow the instructions in Figure B–2.

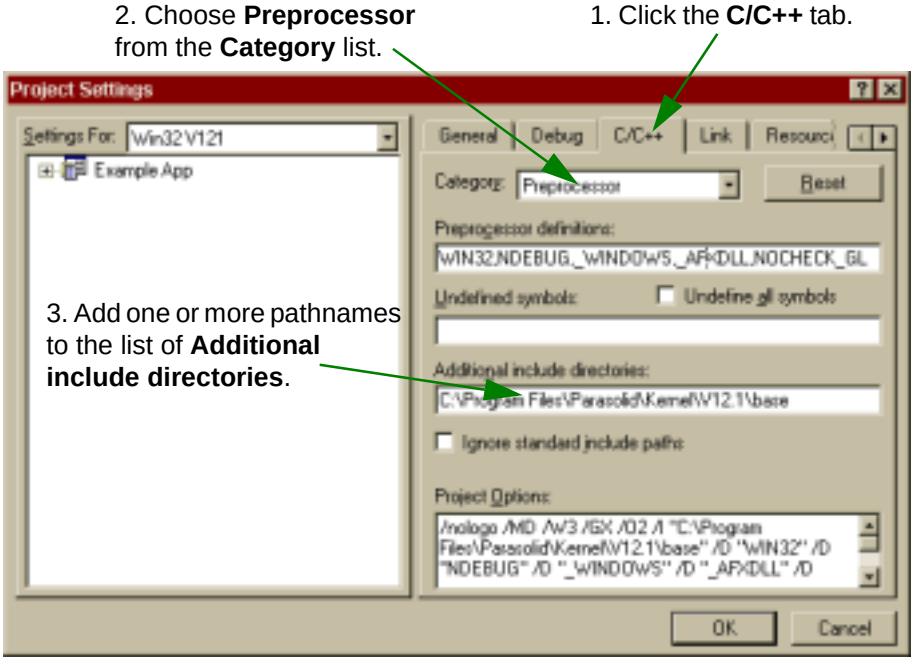

Figure B–2 Adding an additional include directory to MS Visual Studio