Parasolid V36.0

Installation Notes

July 2023

Important Note

Tel: +44 (0)1223 371555
email: parasolid.support.plm@siemens.com
Web: www.parasolid.com

SIEMENS

Trademarks

Table of Contents

A

Introduction 1

1.1 Introduction

Parasolid is a copyright product of Siemens Industry Software Inc. This document covers details of the release of Parasolid and supporting utilities, and retrieval of the product onto the following platforms:

- Intel and AMD-based PCs running Windows or Linux
- ARM-based devices running Windows
- ARM-based systems running Linux
- Intel-based Apple Macintosh systems running macOS, or the iOS Simulator
- ARM-based Apple Macintosh systems running macOS
- ARM-based Apple devices running iOS or iPadOS
- ARM-based devices running Android

It assumes a basic knowledge of the operating system for the relevant platform (creating and moving directories, deleting and listing files for example).

1.2 Contents of this manual

The chapters of this document describe:

Chapter	Description
Chapter 2, “The Product”	What this release comprises
Chapter 3, “Installing Parasolid”	How to install the product from DVD
Chapter 4, “Parasolid Release Area”	What files are included in the release
Chapter 5, “Using Parasolid”	How to run Parasolid
Chapter 6, “Environment”	The operating system environment where the product was created
Appendix A, “Floating-Point Traps”	Information about floating-point traps

1.3 Platform-specific information

Throughout this manual, text and instructions that are specific to a particular platform or group of platforms are referred to as follows:

Label	Platform
X64 WIN	AMD-based or Intel-based PCs running 64-bit Microsoft Windows.
ARM WIN	ARM-based devices running 64-bit Microsoft Windows 10.
Intel NT	Intel-based or AMD-based PCs running 32-bit Microsoft Windows.

Label	Platform
Intel Linux	Intel-based or AMD-based PCs running Linux.
ARM Linux	ARM-based systems running Linux.
ARM MacOS	ARM-based Apple Macintosh systems running macOS.
Intel MacOS	Intel-based Apple Macintosh systems running macOS.
ARM iOS	ARM-based Apple devices running iOS or iPadOS.
Intel iOS	The iOS Simulator, running on Apple Macintosh systems.
ARM Android	ARM-based devices running Android.
UNIX	Includes the platforms referred to by the following labels: ■ Intel Linux ■ ARM Linux ■ ARM MacOS ■ Intel MacOS
Windows	Includes the platforms referred to by the following labels: ■ X64 WIN ■ ARM WIN ■ Intel NT
Linux	Includes the platforms referred to by the following labels: ■ Intel Linux ■ ARM Linux
MacOS	Includes the platforms referred to by the following labels: ■ ARM MacOS ■ Intel MacOS
iOS	Includes the platforms referred to by the following labels: ■ ARM iOS ■ Intel iOS
Android	Includes the platforms referred to by the following labels: ■ ARM Android

The Product 2

2.1 Release contents

This release of Parasolid is distributed on a single DVD. It contains:

- The Parasolid library, driver program, acceptance tests, and other files necessary for the programs to run, for all platforms.
- The Parasolid Jumpstart Kit (which includes documentation, extensive code examples, Parasolid Workshop, and example applications), together with a standalone version of the Parasolid Documentation Suite in HTML and PDF formats

2.2 Compatibility

Parts created in earlier versions of Parasolid are upward compatible with the current release of Parasolid.

2.2.1 Filename extensions on Windows

KID and the dummy Frustrum test whether a file resides on a DOS style FAT device or a long name NTFS type device before opening the file, and act accordingly:

- Files on NTFS devices use the same 7 character extensions (“.xmt_txt”, “.sch_txt” etc.) as on the other platforms.
- Files on DOS style FAT devices use the same 3 character extension as at V6.1. They are shown in the following table in upper case for clarity, though the case is ignored.

This table shows the relationships between the FAT and NTFS names (files can simply be renamed when transferring between the different systems):

	FAT	**NTFS**
part	.X_T	.xmt_txt
schema	.S_T	.sch_txt
journal	.J_T	.jnl_txt
snapshot	.N_T	.snp_txt
partition	.P_T	.xmp_txt
delta	.D_T	.xmd_txt
mesh	.M_T	.xmm_txt
binary	.*_B	.***_bin

Installing Parasolid 3

3.1 Disk space requirements

3.1.1 Software

At this release of Parasolid, the library and supporting files require the following disk space:

X64 WIN	317 MB
ARM WIN	308 MB
Intel NT	256 MB
Intel Linux	236 MB
Intel Linux, GCC7-compatible build	236 MB
ARM Linux	208 MB
ARM MacOS	153 MB
Intel MacOS	179 MB
ARM iOS	93 MB
Intel iOS	104 MB
ARM Android	184 MB

3.1.2 Parasolid Jumpstart Kit

The on-screen installation process for the various components of the Jumpstart Kit tells you how much disk space is required. See Section 3.3 for more information on the Jumpstart Kit.

3.1.3 Parasolid documentation suite

Full installation, with search and PDF files	<<< 246 MB
Search files	<<< 141 MB
PDF files	<<< 36 MB

If you are installing the documentation to a local hard disk, you can save disk space by omitting the search files or PDF files. See Section 3.3, “Parasolid Jumpstart Kit”.

3.2 Copying files to a local disk

Follow these instructions to copy the Parasolid distribution onto a local hard disk.

3.2.1 Using the installer

If you are running on a Windows platform, you can install Parasolid as follows:

- Insert the DVD in the drive.
- Navigate to the DVD in Windows Explorer, click **Setup.exe** and follow the instructions on screen to install Parasolid onto the file system.
- Alternatively, open a Command Prompt window, navigate to the DVD, and run this command to install Parasolid, using the installer's default options:
  `setup.exe /S`

Note: You can install Parasolid on a Windows platform without using the installer if you wish: just copy the files to your file system manually, as described in Section 3.2.2.

3.2.2 Copying files manually

For platforms without installers, or if you do not want to use the supplied installer, copy the files to a file system manually. On Windows platforms, use Windows Explorer to copy the files to a folder of your choice. On MacOS, use the Finder. For any other platforms, follow the instructions below:

- Insert the DVD in the drive. If the DVD auto-mounts, then proceed to the next step. Otherwise, please consult your system administrator.
- Copy the files from the appropriate directory, for example:

```
cp -r /dvdrom/parasolid/platform/base .
```

- For iOS development, you will need to build your app using an Apple Macintosh computer. Your Mac will make use of the files we supply, in building your app, which can then be run on iOS devices or the Simulator.
- For Android development, you will need to build your app using a Windows, Linux or Macintosh computer and an appropriate Android Software Development Kit. This development machine will make use of the files we supply, in building your app, which can then be run on Android devices or the Android Emulator.

3.3 Parasolid Jumpstart Kit

The Parasolid Jumpstart Kit contains all the documentation available for Parasolid, including introductory material, a wide range of code examples within sample applications, and an application which allows you to view and analyse Parasolid models.

Most of the documentation is supplied in both HTML and PDF formats. Some introductory material is in HTML only.

3.3.1 Components of the Parasolid Jumpstart Kit

The Parasolid Jumpstart Kit contains the components described in this section. All of the components described can be installed using the installer on the DVD.

3.3.1.1 Jumpstart Kit Website

This is a website, installed locally, that contains a wealth of material to help you get started using Parasolid quickly. It includes an overview of Parasolid functionality, an introduction to the Parasolid API, a glossary of terminology, and a complete list of functions in the API, as well as access to the other components that can be installed from the DVD.

3.3.1.2 Software

A number of software applications are included on the DVD:

- **Parasolid Workshop.NET** is an extensible application for viewing and analysing XT data.

  As well as the application, Parasolid ships with the complete source code for Parasolid Workshop.Net. Written in C#, and developed using Microsoft Visual Studio,Workshop.Net makes use of the .NET Binding for Parasolid, and demonstrates best practice guidelines for developing an application based on Parasolid.

  Parasolid Workshop.Net is built using the Parasolid Framework, a library of re-usable and extensible C# classes, whose plugin mechanism allows you to add your own functionality to Workshop.Net or easily build your own viewing application.

  The **Parasolid Example Applications** provide a framework within which you can prototype and demonstrate any aspect of Parasolid functionality quickly and simply. The following example applications are available for Intel NT and X64 WIN:

  - C++ Example Application for Windows
  - C# Example Application for Windows

  Within the framework of the applications, you can add and execute calls to Parasolid functionality, and display the results on screen. The code you add can either be executed all at once or in stages. At the end of each stage the contents of the Parasolid session are displayed on the screen.

Note: Example applications for other platforms may be available on request.

Note: Parasolid Workshop.NET is not for redistribution.

3.3.1.3 Parasolid code examples

Also included on the DVD is an extensive suite of example Parasolid application code that makes it easy to learn by example. Each example application contains a folder containing many code examples that demonstrate different functional areas of Parasolid. The code examples are organized by functional area and many contain additional notes and images to add extra useful information. Also included are instructions on how you can try out each example yourself using the relevant example application.

Note: All Parasolid code examples provided in this collateral set are classed as sample code and are intended for educational purposes only.

3.3.1.4 Parasolid Documentation Suite

The Parasolid Documentation suite consists of the following Parasolid manuals, provided in both HTML and PDF format:

- What's New in Parasolid
- Overview of Parasolid
- Installation Notes (this document)
- Functional Description
- PK Reference
- Downward Interfaces
- Kernel Interface Driver (KID)
- Reporting Faults to Parasolid
- XT (Transmit File) Format manual
- Foreign Geometry User Guide
- KI Programming Reference

See Section 3.3.2 for instructions on how to view the Documentation Suite without having to install other components from the Parasolid DVD.

3.3.2 Viewing the documentation suite

If you do not wish to install it to your hard disk as part of the Jumpstart kit, you can access the Parasolid Documentation Suite directly from the DVD as follows:

Windows
- Place the Parasolid DVD in a suitable drive.
- Use Windows Explorer to open the file `X:\online_docs\index3.html`, where *X* is the drive letter for your DVD drive.

UNIX
- Mount the DVD on the system, as described in Section 3.2.2, “Copying files manually”.
- Use your web browser to open the file `online_docs/index3.html`.

iOS
- Use your development Mac to view the documentation, as described in Section 3.2.2, “Copying files manually”.
- Use your web browser to open the file `online_docs/index3.html`.

MacOS
- Use your development Mac to view the documentation, as described in Section 3.2.2, “Copying files manually”.

Android
- Use your development machine to view the documentation, as described above under Windows or UNIX.

Note: The Parasolid documentation uses JavaScript by default. If you view it using a browser without JavaScript, you should be redirected to a page with a link to a version of the documentation without JavaScript. If you are not automatically redirected, please open the `index4.html` file in the `online_docs` directory.

3.3.3 Using the PDF documentation

You need a program that reads PDF files, such as the free Adobe Reader from Adobe Systems Inc., to view the PDF documentation. You can download versions of Adobe Reader for many operating systems from `https://get.adobe.com/reader`

3.3.4 Copying the documentation to a local hard disk or intranet

If you wish, you can copy the entire documentation set onto a local hard disk, or a disk on your local intranet. To do this, either install the Parasolid Jumpstart Kit using the Parasolid installer, or copy the entire `online_docs` folder to a location of your choice. You can run the Parasolid installer by running the Setup.exe directly from the DVD. If you are using UNIX, you may need to mount the DVD drive first, as described in Section 3.2.2, “Copying files manually” above.

Once you have copied the documentation, do one of the following:

- If you are viewing the documentation via an intranet site, create a bookmark or link to the `index.html` file in the `online_docs` directory.
- If you are viewing the documentation via a local or mounted disc, create a bookmark or link to the `index3.html` file in the `online_docs` directory.
- If you are viewing the documentation from a mobile device or Javascript is disabled on your chosen web browser, create a bookmark or link to the `index4.html` file in the `online_docs` directory. This file contains a version of the documentation that does not use Javascript.

Note: On Windows-based machines, you must view this file using either a mounted drive name, or via a web server. Specifically, you cannot access the documentation using a UNC path name (one that begins with `\\`).

The HTML documentation uses Cascading Style Sheets to control the appearance of the documentation: if your browser does not support CSS, or you do not wish to use it, then you can rename or delete the file `online_docs\ps_doc.css`.

Note: Customers must not make Parasolid documentation accessible outside the organization that has licensed Parasolid.

3.3.5 Minimizing disk requirements for documentation

If you do not want to use the search functionality provided in the full documentation set, you can reduce the disk space used by the documentation, by removing the `searchfiles` sub-directory from your installation

This will save you approximately 100MB of disk space, but you will not be able to use the Parasolid documentation’s built-in full-text search.

You can save more disk space by removing the PDF documentation.

- Remove the `pdf` sub-directory from your installation
- If you wish, remove the `pdf_index.html` page from your installation

Warning: Removing any part of the Parasolid documentation suite will inevitably break some hyperlinks in the suite, or render parts of it unusable. Do this at your own risk.

Parasolid Release Area 4

4.1 Contents of the release area

The top level of the Parasolid DVD contains a `parasolid` folder. Inside this folder, there is a sub-folder for each platform available on the DVD. For example, the Parasolid folder contains sub-folders such as `intel_nt`, `intel_linux`, `x64_win`, and `intel_macos`.

Inside each directory is a `base` directory that contains all the files necessary to install and use Parasolid for that platform.

This chapter describes the contents of these directories in some more detail. Apart from some platform-dependent filenames (noted below), the contents are essentially the same for each platform.

Note: This chapter describes the contents as they are structured on the DVD. Your own installation of Parasolid will not necessarily reflect this structure.

Note: An alternative directory name is used for some builds of Parasolid, as shown in the table below. These directories hold files for variant builds of a platform, with the same names as files for the default build.

Platform and build	Directory name
Intel Linux, GCC7-compatible build	`base gcc7compat`

4.1.1 Privileges

You must make sure that all Parasolid users are able to read these files once they have been copied from the DVD.

4.1.2 The ‘base’ directory

The `base` directory contains all the files needed to install and use the full Parasolid kernel.

Sample code for foreign geometry, frustrum and testing:

File	Contents
`fg.c`	Foreign Geometry example source code
`frustrum.c`	Dummy frustrum source code
`frustrum_delta.c`	Example Partitioned Rollback Frustrum code

File	Contents
`frustrum_test.c`	Source of Frustrum library acceptance test
`parasolid_test.c`	Source of Parasolid library acceptance test

Build scripts, pre-built libraries and LISP test (not supplied for iOS), all of which are sample code:

File	Contents
`fg.lib`	Foreign Geometry example library.
`fg_library.bat`, `fg_library.com`	Commands to compile `fg.c` and create a Foreign Geometry library. The `.com` extension is used on UNIX.
`frustrum.lib`	Pre-built dummy frustrum library.
`frustrum_library.bat`, `frustrum_library.com`	Commands to compile `frustrum.c` and create the Frustrum library. The `.com` extension is used on UNIX.
`frustrum_link.bat`, `frustrum_link.com`	Commands to compile and link `frustrum_test.c`. The `.com` extension is used on UNIX.
`kid_test.lsp`	KID acceptance test.
`parasolid_link.bat`, `parasolid_link.com`	Commands to compile and link `parasolid_test.c`. The `.com` extension is used on UNIX.

4.1.2.1 The ‘lispdata’ sub-directory

This directory contains just one file, not supplied for iOS. This is for use with KID (see below) and is not for redistribution.

`bbcini.lsp`	KID LISP initialization start-up file.

4.1.2.2 The ‘schema’ sub-directory

This directory contains a number of schema files whose names have the form `sch_`*XXX*`.sch_txt`.

4.1.2.3 The ‘dll’ sub-directory (Windows)

This directory contains a Parasolid DLL, its corresponding import libraries, and a KID executable linked against the Parasolid DLL. The following DLLs are available:

- `pskernel.dll` – DLL for Parasolid kernel
- `pskernel_net.dll` – .NET binding DLL for Parasolid (relevant Windows platforms only). See Chapter 12, “Calling Parasolid From .NET Code” in the Parasolid *Functional Description* manual for more details.

4.1.2.4 The ‘shared_object’ sub-directory (UNIX & Android)

This contains the Parasolid shared library, a KID linked against it, and some supporting files:

- `libpskernel.so` – Parasolid shared library.
- `libkid_support.so` – shared library of support functions for KID.

On MacOS platforms, the filename extension for shared libraries is `.dylib` rather than `.so`

4.1.2.5 iOS

No form of shared or dynamic library, nor a KID executable, are supplied for iOS.

4.2 Explanation of files

4.2.1 Files in ‘base’

4.2.1.1 Parasolid archive library

For most purposes, it will be more convenient to use the Parasolid “shared library”, found in the ‘dll’ subdirectory on Windows platforms, or the ‘shared_object’ subdirectory on UNIX and Android. The archive library is supplied for use on iOS platforms, and for customers who need to link their own shared library,

4.2.1.2 PK and KI interface header files

Interface functions and type declarations for C/C++ applications.

4.2.1.3 Ifails and tokens header files

These are necessary for an application program to avoid using numeric values for PK arguments.

4.2.1.4 Parasolid test code

This is included so that you can make a simple test that the library is set up correctly before integrating it with the application program. Parasolid requires a Frustrum - thus a dummy frustrum library is also included. This should only be used for experimentation and learning, any customer Frustrum should be written to suit the application. Parasolid can also use Foreign Geometry functions, and example functions are included, although you should contact Parasolid Support before using them.

4.2.1.5 Frustrum test code

This tests the dummy frustrum library in isolation from the Parasolid library. It uses the Frustrum tester function (TESTFR), see the Parasolid Downward Interface manual for a full explanation of this. This program can also be used to test your own Frustrum.

4.2.1.6 Frustrum source code and library

Supplied for building and running the frustrum test code.

4.2.1.7 Frustrum library command file

Supplied to build a Frustrum library from the source code.

4.2.1.8 Frustrum link and Parasolid link command files

These are supplied to link the test programs with the libraries.

4.2.2 The Kernel Interface Driver (KID)

The KID program provides a LISP front-end to the Parasolid library, which can be used to learn the Kernel, and prototype applications. For a full description see the Parasolid Kernel Interface Driver (KID) Manual. KID loads the LISP initialization file at start-up. The KID test file is supplied as an acceptance test for the KID program.

Note: The KID program is not for redistribution.

4.2.3 Schema files

The content and format of Parasolid part files (either transmit or partition files) is referred to generically as the *schema*. The schema needs to be present so that a Parasolid-powered application can

- read part files created using other versions of Parasolid
- write part files that can be read in other versions of Parasolid

New schemas are released from time to time with new releases of Parasolid. You can find out what the current schema version is using PK_SESSION_ask_schema_version.

By default, since Parasolid V14.0, the relevant schema is embedded inside every Parasolid part file. This ensures that Parasolid parts can be received not only by later versions of Parasolid (e.g. you can read a V14.0 part file into Parasolid V15.0), but by earlier versions as well (e.g. you can read a V15.0 part file into Parasolid V14.0).

However, part files saved by versions of Parasolid before V14.0, and part files saved in any version of Parasolid using an explicit transmit version, do not have the relevant schema embedded. In these cases, an external *schema file* must be present in order to receive and transmit Parasolid data.

Schema files are platform independent, text based files with a name of the form `sch_XXX.sch_txt` (or `sch_XXX.s_t` for DOS FAT file systems).

The appropriate schema file must be present when receiving a part file created in an older version of Parasolid, or when transmitting a part file using an explicit transmit version. For instance if you are trying to read a V11.0 or V11.1 part file into Parasolid V15.0, the relevant schema file (`sch_11004`, in this case) must be present.

- Parasolid guarantees upward compatibility of part files between major versions of Parasolid. To support this, schema files corresponding to all previous versions of Parasolid are provided with each new release.
- In Parasolid V14.0 and later, Parasolid guarantees downward compatibility of part files with an embedded schema to at least the previous major version. Thus, a V15.0 part file can be received by Parasolid V14.0.
- Parasolid also guarantees two-way compatibility within a major version of Parasolid (e.g. a V14.0 file can be read into V14.1 and vice-versa).
- Caveat: parts and partitions containing facet geometry can be received into Parasolid V28.0, but the facet geometry will not be present. They work normally in Parasolid V28.1 onwards.

Warning: You should not attempt to “force” downward compatibility for part files that do not contain an embedded schema by copying the schema file for a later release into an earlier release of Parasolid. The behavior under these circumstances is not guaranteed or supported by Parasolid. Furthermore, the behavior of downstream operations on such parts is also not guaranteed.

4.2.4 Finding shared libraries

You can link your application(s) against the Parasolid shared library, and then upgrade them with a newer version of Parasolid by simply providing the new shared library file. You must make sure that the new library is a Parasolid version that is compatible with the old one.

If an application needs to check the Parasolid version at run-time, it can use run-time library functions to open the shared library (e.g. dlopen() on some UNIX machines, or LoadLibrary() and GetProcAddress() on Windows platforms) and call PK_SESSION_ask_kernel_version to extract the version information.

The KID linked against the Parasolid shared library contains embedded information about where to look for the Parasolid library. This information may need to be overridden, so that the run-time loader will search the directory where the customer has put the libraries:

4.2.4.1 UNIX

The KID executable in `base/shared_object/kid.exe` is built consistently on all UNIX platforms and expects to use a PATH-like environment variable to locate the Parasolid shared library. You need to ensure that the directory containing the Parasolid library is included in the value of this environment variable. The name of the environment variable used varies depending on the platform, as shown below:

Platform	Variable name
Linux	`LD_LIBRARY_PATH`
MacOS	`DYLD_FALLBACK_LIBRARY_PATH`

Note: For MacOS platforms, see Section 6.6.5, for notes on using `install_name_tool` to avoid the need to use `DYLD_FALLBACK_LIBRARY_PATH`. This is recommended, because the DYLD family of variables are limited by the System Integrity Protection security system.

Further details should be available in the linker and loader documentation for your platform. The `ld` man page is usually a good starting point.

4.2.4.2 Windows

The PATH environment variable is also used as the DLL search path. The search path for a DLL is:

- The directory where the executable module for the current process is located.
- The current directory.
- The Windows system directory.
- The Windows directory.
- The directories listed in the PATH environment variable.

See Section 6.6.1, “Windows run-time libraries”.

See also Section 6.9, “Linking Windows run-time libraries”

4.2.4.3 iOS

No form of dynamic or shared library is supplied for these platforms, nor is a KID executable. This section is not applicable to these platforms.

4.2.4.4 Android

The KID executable in `base/shared_object/kid.exe` is built to run in the ADB shell environment, not on a standalone device. It expects to use the `LD_LIBRARY_PATH` variable to locate the Parasolid shared library, very much like Linux. You need to ensure that this environment variable is set up, and the directory containing the Parasolid library is included in its value. Parasolid also requires the C++ run-time library from the Android NDK, `libc++shared.so`, which you must copy to the device, and place in a directory that is included in `LD_LIBRARY_PATH`. See Section 6.3.8, “ARM Android” for more details.

Using Parasolid 5

5.1 Files supplied

UNIX, Android Parasolid is supplied as

- **A shared object library.** An executable program can be made by linking the object files of a main program, your frustrum, and the Parasolid shared object library in the 'shared_object' sub-directory.
- **An archive library.** An executable program can be made by linking the object files of a main program, your frustrum, and the Parasolid library.

Windows Parasolid is supplied as:

- **A DLL** (`pskernel.dll`) created from the object file library. An executable program can be made by linking the object files of a main program, your frustrum, the Parasolid 'interface library', `pskernel.lib` in the `dll` directory.
- **An object file library** (`pskernel_archive.lib`). An executable program can be made by linking the object files of a main program, your frustrum, and the Parasolid library.

iOS Parasolid is supplied as:

- **An archive library** (`libpskernel_archive.a`). This can be linked into an Xcode project that builds an iOS app, which must supply a frustrum.

All platforms Function headers for the functions available in the full Parasolid kernel can be found in the file `parasolid_kernel.h`.

In order to use the supplied DLL, your application must register a frustrum – see Chapter 5, "Registering the Frustrum" in the *Downward Interfaces* manual for further details. See Chapter 6, "Environment", in this manual, for more details on building and linking Parasolid.

UNIX, Windows, Android The file `parasolid_link.com`, or `parasolid_link.bat` on Windows platforms, shows examples of linking or binding Parasolid.

Warning: If you link Parasolid using the archive library, it is important that you call only the functions in the supplied headers and do not attempt to call any Parasolid internal functions. The effects of doing so are undefined and unpredictable, and will not be consistent from version to version. Crashes and infinite loops are likely results.

5.2 Schema Files

When transmitting part files using an explicit (non-default) transmit version, the schema file is written to the schema directory of your Parasolid installation if it is not already present.

When receiving part files:

- If the part was originally transmitted using Parasolid V14.0 or later, using the default transmit version, then no schema file is required to read the part into a Parasolid session.
- If the part was originally transmitted using a version of Parasolid earlier than V14.0, or if an explicit (non-default) transmit version was used, then the schema file for the relevant version must be present in the schema directory of your Parasolid installation.

In a Windows application, your frustrum could typically look for schema files in one of two ways:

- Use a registry setting
- Use an environment variable

The dummy frustrum uses an environment variable called P_SCHEMA. Your finished application should not use the dummy frustrum.

Note: If you use the dummy frustrum on Windows, the %P_SCHEMA% environment variable is assumed to point to a directory on an NTFS file system. If %P_SCHEMA% points to a DOS FAT file system instead, you must copy and rename the schema files so that they have the extension `.s_t`.

Take care to preserve schema files when updating Parasolid versions.

Note: On iOS, you will have to include the schemas in your application bundle. If you use the dummy frustrum, you will need to call the function `implement_ios_paths()` to provide this frustrum with the path to the schemas.

5.3 Frustrum

When writing simple Parasolid programs, you may wish to initially use the dummy frustrum provided. While this frustrum works, it is not complete, and is only provided as an outline to help you develop your own frustrum. The notes in the source file `frustrum.c` provide more guidance.

Warning: The dummy frustrum is not suitable for use in a robust application. More error checking is required for robust code, but is omitted for the sake of clarity. Do not attempt to use the dummy frustrum as a “black box.” It is necessary to understand the purpose of the frustrum, and write one suitable for your application.

The command files `frustrum_library.com` on UNIX platforms. or `frustrum_library.bat` on Windows platforms, will create a dummy frustrum library from the ‘C’ file, although this library is provided with the release.

Note: On Windows platforms, we recommend that your frustrum creates text files as stream, i.e. LF terminated, rather than using the DOS default (CR and LF terminated). Implementation details can be found in the dummy frustrum. UNIX, Android and iOS platforms naturally create files that are LF terminated.

UNIX, Windows, Android If you wish to use the dummy frustrum, then you must either set the environment variable P_SCHEMA, to the full pathname of the schema directory or you must modify the code in the dummy frustrum so that the schema directory is specified explicitly, and P_SCHEMA is not used. If you use P_SCHEMA, you must make sure it is set every time you log in.

iOS As noted above, the location of schemas is an app design decision on iOS.

5.4 Kernel Interface Driver (KID)

Applies to: UNIX, Windows, Android

Note: Running KID on Android requires some knowledge of the ADB tool, and the interactive ADB shell environment, Running KID is not necessary for app development using an IDE.

5.4.1 Specifying environment variables

In order to run KID, you need to setup some environment variables:

- PARASOLID refers to the ‘base’ directory; it provides a useful shortcut to specifying the full path name to this directory at the command line.
- P_LISP points to a directory that contains information needed by KID.
- P_SCHEMA points to the directory that contains the schema files.

If your application uses the dummy frustrum, then you may have already specified P_SCHEMA.

5.4.1.1 UNIX

The environment variables can be set up as follows:

Bourne shell

```
$ PARASOLID="base directory pathname"
$ P_LISP="full LISP directory pathname"
$ P_SCHEMA="full SCHEMA directory pathname"
$ export PARASOLID P_LISP P_SCHEMA
```

‘C’ shell

```
$ setenv PARASOLID "base directory pathname"
$ setenv P_LISP "full LISP directory pathname"
$ setenv P_SCHEMA "full schema directory pathname"
```

You need to make sure these commands are executed before you run KID: you could add the statements shown above to a script, perhaps run from your login script. You should specify full path names so that KID can be run from any directory location.

5.4.1.2 Windows

The environment variables can be set up in a Command Prompt as follows:

```
set PARASOLID=base directory pathname
set P_LISP=full LISPDATA directory pathname
set P_SCHEMA=full schema directory pathname
```

You should set up these environment variables as part of your Windows login account, and you should specify full path names so that Parasolid can be run from any folder location.

5.4.1.3 Android

The environment variables can be set up in the ADB shell environment in the same way as for the Bourne shell on UNIX.

5.4.2 Running KID

KID is supplied as an executable file and can be invoked as described below

UNIX Type the following at a UNIX prompt:

```
$ $PARASOLID/shared_object/kid.exe
```

You should ensure you have set the library search environment variable appropriately. See Section 4.2.4, “Finding shared libraries”. When executed, KID will attempt to load the file `$P_LISP/bbcini.lsp`.

Windows If you installed Parasolid using the installer provided (see Section 3.2.1), and you have set up the environment variables in your account, you can run KID by double-clicking `kid.exe` inside either the `intel_nt\base\dll` folder or the `x64_win\base\dll` folder (as appropriate) in your installation directory.

Android You will need to use the ADB tool to transfer files to your device. This must be to a directory where you have write and execute permissions, and you will need to run KID from a directory where you have write permission, since it creates journal files by default. The directory `/data/local/tmp` is suitable for both of these purposes, and you can create sub-directories of it if you wish.

Use `adb push` to transfer the contents of the shared_object directory, the `lispdata` directory and the `schema` directory to your device. Start the shell environment with `adb shell`, use `cd` to move to a directory where you have write permission, set the environment variables, and run `kid.exe`.

5.4.3 KID graphics

This section provides a quick guide to initializing KID’s graphics facilities. For more information, see the *KID Manual*.

5.4.3.1 Opening a graphics window

Intel Linux, ARM Linux, Intel MacOS KID graphics uses the X Windows system. Make sure that X is running before starting KID, then before using any other graphics commands, type one of the following at the LISP prompt:

```
> (graphics open_device 'x)
> (graphics open_device 'xcolour)
```

To use KID graphics on Intel MacOS, you need to install and start the X11 server. This can be found under **Applications > Utilities > X11** on any Intel MacOS system.

Windows platforms KID graphics uses Microsoft Windows GDI functions. Monochrome and color windows are available as alternatives, by typing either of the following at the LISP prompt:

```
> (graphics open_device 'nt)
> (graphics open_device 'ntcolour)
```

ARM MacOS, Android KID graphics are not available on these platforms.

5.4.3.2 Other graphics commands

If no graphics are required, use:

```
> (graphics open_device ’null)
```

This will allow you to call Parasolid rendering functions without getting graphics output (for example, for testing purposes

You can ensure KID graphics are set up correctly using

```
> (graphics enquire)
```

5.5 Stack sizes

Executing Parasolid code requires enough stack space: at least 1MB for 32-bit platforms, and 2MB for 64-bit platforms. There are currently two platforms where the default stack size for the program's first thread is too small, and several platforms where the default stack size for additional threads (which can be different) is too small. The latter applies both to application threads that call Parasolid, *and* to threads that Parasolid creates internally for its SMP functionality.

Note: A thread's stack size cannot be changed once the thread exists: it is fixed when the thread is created.

X64 WIN, ARM WIN For these platforms, use the following option to set the minimum default stack size: `/stack:2097152,131072`. A larger stack size (the first value) will be required if your application uses significant amounts of stack space itself.

- The KID test harness includes a LISP interpreter, which can require significant amounts of stack space. KID is given 4 MB of stack.
- The .NET binding does not impose significant stack space overheads.

You can do this with either `link.exe`, when linking your executable file(s), or `editbin.exe`, to modify them after linking. Note that this is required for *all* 64-bit Windows programs that use Parasolid, even if they do not use multiple application threads or Parasolid SMP threads.

5.5.1 Platform default thread stack sizes

The default thread stack sizes for each platform are shown in the table below. Sizes marked with * are too small by default, so any Parasolid-based application must change them.

Platform	**Required**	**First thread default**	**Additional threads default**
X64 WIN	2MB	1MB*	1MB*
ARM WIN	2MB	1MB*	1MB*
Intel NT	1MB	1MB	1MB
Intel Linux	2MB	8MB	2MB
ARM Linux	2MB	8MB	2MB
ARM MacOS	2MB	8MB	512KB*
Intel MacOS	2MB	8MB	512KB*
ARM iOS	3MB	See below	512KB*
Intel iOS	3MB	See below	512KB*
ARM Android	2MB	See below	1MB*

5.5.2 Setting stack sizes for application threads

For X64 WIN and ARM WIN, we recommend that you use the `/stack` option documented above to change the default stack size for all threads, including application threads.

For UNIX, Android and iOS platforms where the stack size is too small, use `pthread_attr_setstacksize` to set a stack size attribute in the `pthread_attr_t` structure that you pass to `pthread_create` when creating your application threads.

For iOS, we recommend that you should not call Parasolid from the first thread of your app. Some Parasolid calls can take enough time to cause iOS to decide that the app has become unresponsive, and kill it. Create a thread with an appropriate size stack and use that to call Parasolid.

For Android, you should likewise not call Parasolid from the UI thread of your app. As for iOS, Parasolid calls can take enough time to cause Android to decide that the app has become unresponsive, and display an “Application Not Responding” message. Create a thread with an appropriate size stack and use that to call Parasolid.

Calling PK_SESSION_set_smp_stacksize has no effect on application threads: that PK function only affects the stack sizes of threads created by Parasolid.

5.5.3 Setting stack sizes for Parasolid internal SMP threads

Call PK_SESSION_set_smp_stacksize before turning on Parasolid SMP. For example:

```
size_t new_stk = 256 * 1024 * sizeof(char*); /* 384, not 256, on iOS */
PK_SESSION_smp_o_t opt;
PK_SESSION_smp_o_m( opt ); /* Set SMP parameters in opt */
PK_SESSION_set_smp_stacksize( new_stk );
PK_SESSION_set_smp( &opt );
```

5.6 Symmetric Multi-Processing (SMP)

Parasolid has been adapted for Symmetric Multi-Processing (or parallel processing) via the multi-threading environments supported under Windows and the Posix threads model on UNIX. The following platforms support SMP at this release:

- X64 WIN
- Intel NT
- Intel Linux
- ARM Linux
- ARM MacOS
- Intel MacOS

5.6.1 Guidelines for setting the number of threads

There are several issues that need to be considered when deciding how many threads you should use, some of which are platform-dependent.

5.6.1.1 Parasolid’s threads

Parasolid is thread-safe and can be called from multiple threads in the same application. The Parasolid PK and KI APIs manage multiple application threads to ensure safety. See Section 114.2, “How to set up and use application threads”, in the Parasolid *Functional Description* for details. Parasolid can also create and use “worker” threads internally, to improve performance. This is described in Chapter 115, “Symmetric Multi-Processing In Parasolid”, in the *Functional Description*, with more details below.

For many Parasolid functions, the thread that enters Parasolid from the calling application does all the work. For functions that use SMP, the caller’s thread does preparatory work, and then sets Parasolid SMP “worker” threads to work. Those threads are created by Parasolid, and perform most of the work of functions that use SMP. The caller’s thread waits for the worker threads to finish, collects their results, and returns to the application. The number of these worker threads is set by PK_SESSION_set_smp. That is the maximum number of such threads that Parasolid will ever be using at one time. Application threads are not included in this number.

The Parasolid functions that use SMP are functions that perform jobs that can be decomposed into many smaller and independent tasks. For example, mass properties such as surface area are calculated by integrating over all the faces of a body. The faces are independent, so separate threads can handle them easily. A side effect of this design is that the worker threads all do similar kinds of calculations. Their performance is limited by the speed of the computer’s processors and their memory subsystems; the speed of other parts of the computer has little effect.

If Parasolid SMP cannot create internal worker threads, it carries out the work they would have performed in the thread used to call Parasolid. This means that Parasolid continues to operate correctly (though possibly less quickly) if the operating system is unable to create worker threads when Parasolid needs them. Parasolid reports this situation through the Parasolid report mechanism. If worker threads subsequently can be created again, they will be used. This is also reported via the Parasolid report mechanism.

5.6.1.2 Limited number of threads

There is a limit on the number of threads that Parasolid can use, noted in the documentation of PK_SESSION_smp_o_t.

5.6.1.3 Multi-core processors

Symmetrical multi-core processors, with two or more complete and identical physical processors in one package, work very well with Parasolid SMP.

Asymmetrical multi-core processors, such as the Apple M1, used in the first generation of ARM MacOS systems, have a mixture of types of processors. The usual configuration is a mix of high-performance and power-efficient cores, to enable performance to be traded off against battery life. You may need to limit the number of Parasolid SMP worker threads to the number of high-performance cores, but experimentation and benchmarking with your application’s usage of SMP is advisable.

5.6.1.4 Symmetric Multi-Threading (SMT) and HyperThreading

SMT describes a particular style of processor design. This is useful for some kinds of applications, but not – so far – for Parasolid. The best-known SMT system is Intel’s HyperThreading. This shares the execution units of a single processor between two “logical processors”. Both of these are visible to the operating system, and can be given work to do, but the threads they run only get a share of the processor’s power, rather than the full use of a processor. This makes SMT most useful for applications that need a large number of threads but where the threads don’t have to do a lot of calculations. Web serving, file/print serving and similar tasks often benefit from SMT.

HyperThreading isn’t very beneficial to Parasolid, although they work together correctly. Some Parasolid operations can run faster by using the extra threads that HyperThreading provides, but others run slower, and the overall effect is usually a small loss, on “NetBurst” architecture processors, or no benefit with the improved HyperThreading of the Core i-series processors. Parasolid worker threads are limited by processor power, and HyperThreading doesn’t provide more of that; it only provides a different way of accessing the existing power. Parasolid SMP threads also tend to be doing similar work, and thus competing for the same execution units.

For best Parasolid performance on old NetBurst processors, turn off SMT. For Core i-series and later processors, there is no need to do this.

5.6.1.5 Processor affinity

We do not recommend that Parasolid-based applications should try to programmatically associate themselves with particular processors in an SMP system. If two or more programs try to do such things simultaneously, the usual result is worse performance.

ARM WIN is a special case. The usual processors for this platform are asymmetric. This works well in interactive usage, but causes problems in Parasolid's automated testing. For reasons that are not yet clear to us, if an ARM WIN device has been running test code with no user interface for several days, it tends to run everything on its slow cores, dramatically slowing down testing. We suggest that any automated testing systems you create for applications that run on ARM WIN should use processor affinity to ensure that they run on the fast cores. This solved our problems with Parasolid testing. An example program, `probe_affinity.c`, is supplied with the Parasolid Jumpstart Kit. This examines a Windows computer to discover if any of the cores are significantly faster, and provides examples of setting affinity. We suggest that you experiment with processor affinity in your application(s); we do not yet know enough to recommend that you should, or should not, use it in interactive applications.

5.6.1.6 Type of application

For applications that are the main work of the computer running them, such as interactive CAD on a desktop machine, use as many threads as there are available physical processors, since response time to the user is of primary importance.

For applications such as translators, which may form a small part of the workload of a server, consider the issue more carefully. Parasolid doesn't run twice as fast with two threads, so the overall throughput of a heavily loaded server may decrease if you use Parasolid SMP. However, if a server with many processors isn't heavily loaded, using Parasolid SMP can be beneficial. For server applications, it is wise to make the use of SMP something that can be configured, rather then being decided automatically by the program.

5.6.1.7 Intel NT and X64 WIN platforms

There is a wide variety of SMP and SMT hardware that can be used for these platforms. Microsoft Windows provides an API, GetLogicalProcessorInformation, which can be used to identify some of the processor facilities of a computer running Windows, and is recommended for use with Parasolid.

Use this API to find out the number of processor *cores* available. If this is greater than 1, use that number of threads, to a maximum of the Parasolid limit, with the PK_thread_absolute_c thread format.

This method avoids using a thread per logical processor on machines with HyperThreading active.

5.6.1.8 Intel Linux platform

This platform can use most of the types of hardware that are available for Microsoft Windows platforms.

To discover the processor configuration, first use `sysconf(_SC_NPROCESSORS_ONLN)` to get the number of processors. If this is greater than 1, read and parse the text file

`/proc/cpuinfo`. If the processor entries in that file contain “core id” values, then count the number of different “core id” values (they are not always consecutive numbers), to find out how many processor cores exist. With no “core id” values, you have a machine with single-core processors, so count the number of “physical id” values to get the number of processor cores. If there is more than 1 processor core, use that number of threads, to a maximum of the Parasolid limit, with the PK_thread_absolute_c thread format.

This method avoids using a thread per logical processor on machines with HyperThreading active. The best Parasolid performance on machines that support HyperThreading is obtained by turning off HyperThreading in the machine’s BIOS.

5.6.1.9 ARM Linux platform

Use `sysconf(_SC_NPROCESSORS_ONLN)` to get the number of processors. If this is greater than 1, enable SMP with the default option of 1 thread per processor, and Parasolid will apply its own limit.

5.6.1.10 Intel MacOS platform

Use `sysctl()` with `CTL_HW` and `HW_AVAILCPU` to get the number of available processors. If this is greater than 1, enable SMP with the default options of 1 thread per processor; Parasolid will apply its own limit.

Parasolid uses OS-level locking with POSIX threads, since the OS-level locks have much lower overheads than Apple’s pthreads mutexes. Parasolid sets the Quality of Service class for threads that it creates to `QOS_CLASS_USER_INITIATED`, using the function `pthread_attr_set_qos_class_np()`. This assists the macOS scheduler in prioritising threads.

5.6.1.11 ARM MacOS platform

All processors used for this platform are asymmetric. Parasolid SMP works *much* better when it is limited to a number of threads less than or equal to the number of fast processor cores in the machine. You can establish this number as follows:

- On macOS 11, it is four. All the machines capable of running ARM macOS 11.x use the M1 processor, with four fast and four slow cores.
- On macOS 12 and later, you can find the number of processors at performance level zero, the fastest type, by calling `sysctlbyname()` with the argument `¨hw.perflevel0.logicalcpu¨`. See “man sysctlbyname” for more details.

Parasolid uses OS-level locking with POSIX threads, since the OS-level locks have much lower overheads than Apple’s pthreads mutexes. Parasolid sets the Quality of Service class for threads that it creates to `QOS_CLASS_USER_INITIATED`, using the function `pthread_attr_set_qos_class_np()`. This assists the macOS scheduler in prioritising threads.

5.6.2 Downward interfaces

Please note that when you use Parasolid SMP, Parasolid may call any of its downward interfaces from its SMP “worker” threads. This allows SMP threads to work without having to coordinate with the master thread, which improves performance. Similarly, if your application calls Parasolid from multiple threads, then the downward interfaces may be called from any of those threads. For these reasons, the functions you provide for downward interfaces must not make assumptions about which threads will call them, or in which order.

By default, Parasolid will only make one downward interface call at a time, avoiding the need for your functions to be fully thread-safe. If your frustrum is thread-safe, you can allow Parasolid to make multiple simultaneous calls to GO functions with the 'go_thread_safe' option to PK_SESSION_register_fru_2. Other exceptions to the single-call rule are noted in the documentation for individual functions. For more details, see Chapter 114, "Calling Parasolid From Multiple Threads" and Chapter 115, "Symmetric Multi-Processing In Parasolid" in the Functional Description.

The situations where thread safety in your frustrum is important include:

- Freeing and allocating memory in the frustrum, which must continue to work correctly under these conditions.
- The GO interfaces, where thread safety allows data for one body to be output while data for others is still being generated.

5.6.3 Run-time error handling

On platforms where the operating system doesn't do this automatically, Parasolid copies the floating-point trap settings in use by your application into its SMP "worker" threads. See Section 6.4, "Floating-point information", for details of floating-point traps on Parasolid platforms.

On UNIX platforms, code run by SMP "worker" threads generates run-time errors in the same way as code run by the thread you use to call Parasolid. Your application's signal handler should be prepared to deal with this situation.

On Microsoft Windows platforms, Parasolid's thread management catches run-time errors in worker threads and re-creates them in the thread that you used to call Parasolid. This allows you to use either signals or Structured Exception Handling to handle run-time errors.

Note: C++ exceptions thrown by frustrum functions called by Parasolid worker threads will not be caught in the thread that called Parasolid. We recommend that you do not use C++ exceptions in frustrum functions, but instead return a failure status.

Environment 6

6.1 Introduction

This chapter contains information specific to the different environments in which Parasolid is supported:

- Section 6.2, “Build environment”, describes the operating systems, compilers, and compiler options that were used to build Parasolid on each platform.
- Section 6.3, “Supported operating systems and hardware”, describes the environments in which Parasolid runs.
- Section 6.4, “Floating-point information”, describes platform-specific floating point information.
- Section 6.5, “Solutions to platform-specific issues”, describes common platform-specific issues and situations that you should avoid.
- Section 6.6, “Supporting material for Parasolid libraries”, provides additional platform-specific information.
- Section 6.7, “Parasolid version information”, describes how you can extract version information from the Parasolid library on each platform.
- Section 6.8, “.NET binding”, describes build information for the Parasolid C# Binding for .NET.
- Section 6.9, “Linking Windows run-time libraries”, describes how to deal with some special cases in linking on Windows platforms.
- Section 6.10, “Enabling SIGFPE for SSE2 on Intel NT”, describes how to handle SSE2 floating-point errors on 32-bit Windows platforms.

Note: Please ensure you check all the sections for information related to the platforms you use. Relevant platforms are clearly indicated at the start of each section.

6.2 Build environment

This section contains information for all platforms on the compiler and operating system used to build Parasolid. For details of the environments in which Parasolid can be run, see Section 6.3, “Supported operating systems and hardware”.

The sizes of the C types that are used in Parasolid interfaces (e.g., int, double and pointer) are defined by each platform’s compiler. Please see your compiler documentation for details, and remember that the size of pointers depends on your decision to compile for 32-bit or 64-bit code.

6.2.0.1 C and C++

Since its creation, Parasolid has been written in a domain-specific language, which is “transpiled” to C code, and then compiled with each platform’s C compiler. The C compiler and options used are documented, to assist you with achieving compile and link compatibility with Parasolid.

At v34.1, we began to include C++ code within Parasolid. This means that some of its object files are generated by a C++ compiler, and some by a C compiler. Both compilers, and the options used with them, are described below. Note that on many platforms, the C and C++ compilers are different front-ends to a common code generator, and are thus often considered to be the same program.

You will need to ensure that Parasolid-based applications are linked against an appropriate C++ run-time library for each platform. If your applications already use C++, this is already taken care of. If not, linking using the C++ compiler is the best course of action.

6.2.0.2 C++ standards

At v36.0, Parasolid is compiled as C++17.

Since Parasolid's interfaces are, and will remain, pure C, your applications are not required to use the same C++ standard as Parasolid. However, if your application code is complied using a different standard, check that the run-time libraries for that standard are compatible with the C++ standard used by Parasolid,

6.2.0.3 Choice of compilers

Because Parasolid is large, complex, extensively tested, and sensitive to changes in compilers, we take considerable care that all the computers used in developing and producing it have the same version of the compilers used for the relevant Parasolid platform. We very rarely change or update the compiler versions used for a given version and build of Parasolid after its release. We only do this if a compiler has serious bugs or other problems that make it impractical for further use.

For compilers that are frequently updated, such as Microsoft Visual Studio, we much prefer to use the initial release of each version of the compilers, because that provides greater assurance of upwards compatibility with later releases and updates.

6.2.1 X64 WIN

Parasolid was compiled, linked and tested on 64-bit Windows 10, version 22H2. The C compiler used was the Microsoft (R) C/C++ Optimizing Compiler Version 19.30.30705 for x64, from Visual Studio 2022 v17.0.0.

The following compiler options were used:

```
/MD /DWIN32 /DNDEBUG /D_AMD64_ /GF /W3 /O2 /fp:precise /fp:except /GS
/favor:blend /options:strict /D_WIN32_WINNT=_WIN32_WINNT_WIN7
```

The same compiler was used for C++, with the following options:

```
/EHa /MD /DWIN32 /DNDEBUG /D_AMD64_ /GF /W3 /O2 /fp:precise /fp:except /GS
/favor:blend /options:strict /std:c++17 /D_WIN32_WINNT=_WIN32_WINNT_WIN7
```

Option	Description
`/MD`	Link with `MSVCRT.LIB.` Parasolid requires the Visual Studio 2019 or later version of the Microsoft run-time library DLL for modern versions of Visual Studio, `VCRUNTIME140.DLL`.
`/EHa`	(C++ only) Enable C++ error handing, including asynchronous exceptions.
`/std:c++17`	(C++ only) Compile as ISO C++17.
`/options:strict`	Treat unrecognized command line flags as errors.
`/GF`	Pool read-only strings in read-only memory.
`/W3`	Warning level 3.
`/O2`	Optimize for speed.
`/fp:precise`	Use precise floating-point calculations.
`/fp:except`	Use floating-point exceptions.
`/GS`	Use “GuardStack” stack protection, in compliance with the Microsoft SDL. See Section 6.5.8 and Section 6.6.2.
`/favor:blend`	Optimize for several different x64 processors.
`/D_WIN32_WINNT=` `_WIN32_WINNT_WIN7`	Compile for Windows 7 compatibility.

This build of Parasolid is compatible with all Update levels of Visual Studio 2019 and 2022. It requires Visual Studio 2019 or later run-time libraries, since it does not use the `/d2FH4-` option.

6.2.2 ARM WIN

Parasolid was compiled, linked and tested on 64-bit Windows 10 for ARM, version 21H2. The C compiler used was the Microsoft (R) C/C++ Optimizing Compiler Version 19.30.30705 for ARM64 from Visual Studio 2022 v17.0.0.

The following compiler options were used:

```
/MD /fp:precise /Qfast_transcendentals /D_ARM64_ /O2 /GS /Qspectre
/D_WIN32_WINNT=_WIN32_WINNT_WIN7 /options:strict /GF /volatile:iso /W3
```

The same compiler was used for C++, with the following options:

```
/EHa /MD /fp:precise /Qfast_transcendentals /D_ARM64_ /O2 /GS /Qspectre
/D_WIN32_WINNT=_WIN32_WINNT_WIN7 /std:c++17 /options:strict /GF
/volatile:iso /W3
```

Where:

Option	Description
`/MD`	Link with `MSVCRT.LIB.` Parasolid requires the Visual Studio 2019 or later version of the Microsoft run-time library DLL for modern versions of Visual Studio, `VCRUNTIME140.DLL`.
`/EHa`	(C++ only) Enable C++ error handing, including asynchronous exceptions.

Option	Description
`/std:c++17`	(C++ only) Compile as ISO C++17.
`/fp:precise`	Use precise floating-point calculations.
`/Qfast_transcendentals`	Use transcendental processor instructions, rather than library functions, for improved performance.
`/D_WIN32_WINNT=_ _WIN32_WINNT_WIN7`	Limit Parasolid to using Windows 7 APIs.
`/options:strict`	Treat unrecognized command line flags as errors.
`/O2`	Optimize for speed.
`/GS`	Use “GuardStack” stack protection, in compliance with the Microsoft SDL. See Section 6.5.8 and Section 6.6.2.
`/Qspectre`	Use mitigations for the “Spectre” family of security problems.
`/GF`	Pool read-only strings in read-only memory.
`/volatile:iso`	Use ISO C “volatile” semantics, rather than emulating the slower “MS” form.
`/W3`	Warning level 3.

This build of Parasolid requires Visual Studio 2019 or later run-time libraries. It is compatible with all Update levels of Visual Studio 2022, and with Visual Studio 2019 v16.7 and later. However, we recommend that you use Visual Studio 2022 in preference to 2019, since that avoids compiler bugs we encountered with 2019, and the need to compile with a special pragma to avoid the use of “floating point contractions.” *Do not* use Visual Studio 2019 versions before v16.7 with this platform.

6.2.3 Intel NT

This build uses SSE2 registers and instructions for improved performance.

Parasolid was compiled, linked and tested on 64-bit Windows 10 version 22H2. While it is built on a 64-bit operating system, it is 32-bit software for use in 32-bit applications. The C compiler used was the Microsoft (R) C/C++ Optimizing Compiler Version 19.30.30705 for x86, from Visual Studio 2022 v17.0.0.

The following compiler options were used:

```
/MD /Gs /D_X86_ /GF /GS /fp:precise /arch:SSE2 /O2 /options:strict
/Qfast_transcendentals /D_WIN32_WINNT=_WIN32_WINNT_WIN7
```

The same compiler was used for C++, with the following options.

```
/EHa /MD /Gs /D_X86_ /GF /GS /fp:precise /arch:SSE2 /O2 /options:strict
/std:c++17 /Qfast_transcendentals /D_WIN32_WINNT=_WIN32_WINNT_WIN7
```

Option	Description
`/O2`	Optimize for speed.
`/EHa`	(C++ only) Enable C++ error handing, including asynchronous exceptions.
`/std:c++17`	(C++ only) Compile as ISO C++17.

Option	Description
`/options:strict`	Treat unrecognized command line flags as errors.
`/Gs`	Disable stack checking.
`/MD`	Link with `MSVCRT.LIB.` Parasolid requires the Visual Studio 2019 or later version of the Microsoft run-time library DLL for modern versions of Visual Studio, `VCRUNTIME140.DLL`.
`/GF`	Pool literal strings in read-only storage.
`/GS`	Use “GuardStack” stack protection, in compliance with the Microsoft SDL. See Section 6.5.8 and Section 6.6.2.
`/fp:precise`	Use precise floating-point semantics.
`/arch:SSE2`	Compile for processors that support the SSE2 instruction set.
`/Qfast_transcendentals`	Use transcendental processor instructions, rather than library functions, for improved performance.
`/D_WIN32_WINNT= _WIN32_WINNT_WIN7`	Compile for Windows 7 compatibility.

This build of Parasolid is compatible with all Update levels of Visual Studio 2019 and 2022. It requires Visual Studio 2019 or later run-time libraries, since it does not use the `/d2FH4-` option.

6.2.4 Intel Linux

Default build Parasolid was compiled, linked and tested on Red Hat Enterprise Linux 8.8. The C compiler used was gcc version 11.2.1 from Red Hat GCC Toolset 11, with the following options:

```
-m64 -O2 -fno-strict-aliasing -std=c99 -fPIC
-fvisibility=hidden -D_XOPEN_SOURCE=700 -fexceptions
-fasynchronous-unwind-tables
-Wformat -Wformat-security -D_FORTIFY_SOURCE=2
-fstack-protector-strong
```

The same compiler was used for C++, with the `-std` option changed to `-std=c++17` and the addition of `-D_GLIBCXX_ASSERTIONS`.

Option	Description
`-m64`	Compile for 64-bit code.
`-O2`	Optimization level 2.
`-fno-strict-aliasing`	Do not assume there is no aliasing.
`-std=c99`	(C only) Compile code as ISO C99.
`-std=c++17`	(C++ only) Compile code as ISO C++17.
`-fPIC`	Generate position-independent code.
`-fvisibility=hidden`	Hide Parasolid’s internal symbols.
`-D_XOPEN_SOURCE=700`	Use XOPEN facilities.
`-fexceptions`	Allow C++ exceptions to be thrown through Parasolid.

Option	Description
`-fasynchronous-unwind-tables`	Provide precise stack unwinding tables.
`-Wformat` `-Wformat-security`	Warn of security risks with the format strings used by the printf/scanf family of functions.
`-D_FORTIFY_SOURCE=2`	Use more-secure C string functions.
`-D_GLIBCXX_ASSERTIONS`	(C++ only) Trap on invalid C++ string accesses.
`-fstack-protector-strong`	Use improved stack protection. See Section 6.5.8

The Parasolid shared library is linked using additional options for improved security:

Option	Description
`-z relro`	Make the relocation table read-only.
`-z now`	Load the whole library as soon as any part of it is called.
`-z noexecstack`	The library does not require an executable stack.
`-z defs`	Make sure there are no undefined symbols.

More linking options are required to allow C++ exceptions to be reliably thrown through Parasolid’s C code:

Option	Description
`-lshared-glibc`	Tell the linker to use the shared GCC support library, for consistent exception handling across the application.
`-lgcc_s`	Explicitly require the shared GCC support library.

GCC7-compatible extra build Parasolid was compiled, linked and tested on CentOS 7.9. The C compiler used was gcc version 11.2.1 from Red Hat Developer Toolset 11. The same compiler and linker options were used as the default build. This build was done on CentOS 7.9, so that it would be compatible with our previous GCC7 builds.

6.2.5 ARM Linux

Parasolid was compiled, linked and tested on Red Hat Enterprise Linux 8.8. The C compiler used was gcc version 11.2.1 from Red Hat GCC Toolset 11, with the following options:

```
-march=armv8-a -ffp-contract=off -O -std=c99 -fPIC
-fvisibility=hidden -D_XOPEN_SOURCE=700 -fexceptions
-fasynchronous-unwind-tables -D_FORTIFY_SOURCE=2
-Wformat -Wformat-security -fstack-protector-strong
-fno-strict-aliasing
```

The same compiler was used for C++, with the `-std` option changed to `-std=c++17`, and the addition of `-D_GLIBCXX_ASSERTIONS`.

Option	Description
`-march=armv8-a`	Compile for the basic 64-bit ARMv8 architecture.
`-ffp-contract=off`	Avoid the use of “floating-point contractions,” to improve consistency.
`-O`	Optimisation level 1.
`-std=c99`	(C only) Compile code as ISO C99.
`-std=c++17`	(C++ only) Compile code as ISO C++17.
`-fPIC`	Generate position-independent code.
`-fvisibility=hidden`	Hide Parasolid’s internal symbols.
`-D_XOPEN_SOURCE=700`	Use XOPEN facilities.
`-fexceptions`	Allow C++ exceptions to be thrown through Parasolid.
`-fasynchronous-unwind-tables`	Provide precise stack unwinding tables.
`-D_FORTIFY_SOURCE=2`	Use more-secure C string functions.
`-Wformat` `-Wformat-security`	Warn of security risks with the format strings used by the printf/scanf family of functions.
`-D_GLIBCXX_ASSERTIONS`	(C++ only) Trap on invalid C++ string accesses.
`-fstack-protector-strong`	Use improved stack protection. See Section 6.5.8
`-fno-strict-aliasing`	Do not assume there is no aliasing.

The Parasolid shared library is linked using additional options for improved security:

Option	Description
`-z relro`	Make the relocation table read-only.
`-z now`	Load the whole library as soon as any part of it is called.
`-z noexecstack`	The library does not require an executable stack.
`-z defs`	Make sure there are no undefined symbols.

More linking options are required to allow C++ exceptions to be reliably thrown through Parasolid’s C code:

Option	Description
`-lshared-glibc`	Tell the linker to use the shared GCC support library, for consistent exception handling across the application.
`-lgcc_s`	Explicitly require the shared GCC support library.

6.2.6 ARM MacOS

Parasolid was compiled, linked and tested on macOS 11.2. The C compiler used was Apple clang version 12.0.0 (clang-1200.0.32.27) from Xcode 12.2.

The following options were used:

```
-target arm64-apple-darwin -isysroot
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk
-mmacosx-version-min=11.0 -fstack-protector-strong -O2 -fPIC -dynamic -fno-common -fexceptions
-std=c99 -Wpartial-availability -Werror=partial-availability
```

The same compiler was used for C++, with the `-std` option changed to `-std=c++17`.

Option	**Description**
`-target arm64-apple-darwin`	Compile for 64-bit ARM macOS.
`-isysroot...`	Use the macOS SDK headers supplied with Xcode 12.2.
`-mmacosx-version-min...`	Compile and link for macOS 11.0 or later. See Section 6.6.9
`-O2`	Optimisation level 2.
`-fPIC`	Generate position-independent code.
`-dynamic`	Code intended for a dynamic library.
`-fno-common`	Make sure that there are no common blocks.
`-fexceptions`	Allow C++ exceptions to be thrown through Parasolid.
`-std=c99`	(C only) Compile code as ISO C99
`-std=c++17`	(C++ only) Compile code as ISO C++17.
`-fstack-protector-strong`	Use improved stack protection. See Section 6.5.8
`-Wpartial-availability`	Warn when compiling calls to partially available functions.
`-Werror=partial-availability`	Make the partial availability warning into a compilation error. See Section 6.6.9, “MacOS and iOS backwards compatibility”.

6.2.7 Intel MacOS

Parasolid was compiled, linked and tested on macOS 11.4. The C compiler used was Apple clang version 12.0.0 (clang-1200.0.32.27) from Xcode 12.2.

The following options were used:

```
-m64 -isysroot
/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk
-mmacosx-version-min=11.0 -fstack-protector-strong -O2 -fPIC -dynamic -fno-common -fexceptions
-std=c99 -Wpartial-availability -Werror=partial-availability
```

The same compiler was used for C++, with the `-std` option changed to `-std=c++17`.

Option	**Description**
`-m64`	Compile for 64-bit code.
`-isysroot...`	Use the macOS SDK headers supplied with Xcode 12.2.
`-mmacosx-version-min...`	Compile and link for OS X 11.0 or later. See Section 6.6.9
`-O2`	Optimisation level 2.
`-fPIC`	Generate position-independent code.
`-dynamic`	Code intended for a dynamic library.
`-fno-common`	Make sure that there are no common blocks.
`-fexceptions`	Allow C++ exceptions to be thrown through Parasolid.

Option	Description
`-std=c99`	(C only) Compile code as ISO C99.
`-std=c++17`	(C++ only) Compile code as ISO C++17.
`-fstack-protector-strong`	Use improved stack protection. See Section 6.5.8
`-Wpartial-availability`	Warn when compiling calls to partially available functions.
`-Werror=partial-availability`	Make the partial availability warning into a compilation error. See Section 6.6.9, “MacOS and iOS backwards compatibility”.

6.2.8 ARM iOS

Parasolid was compiled, linked and tested on macOS 11.0. The C compiler used was Apple clang version 12.0.0 (clang-1200.0.32.27) from Xcode 12.2. Parasolid was tested on an iPad Pro with 6GB RAM, running iPadOS 13.5.

Please see Section 6.3.6 for expected changes to this platform’s build and test environment.

The following options were used:

```
-arch arm64 -isysroot
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS.sdk
-mios-version-min=13.0 -fstack-protector-strong -O2 -fPIC -dynamic -fno-common -fexceptions
-std=c99 -Wpartial-availability -Werror=partial-availability
```

The same compiler was used for C++, with the `-std` option changed to `-std=c++17`.

Option	Description
`-arch arm64`	Compile 64-bit ARM code.
`-isysroot...`	Use the iOS SDK headers supplied with Xcode 12.2.
`-mios-version-min=13.0`	Compile and link for iOS 13.0 onwards. See Section 6.6.9.
`-fstack-protector-strong`	Use improved stack protection. See Section 6.5.8.
`-O2`	Optimisation level 2.
`-fPIC`	Generate position-independent code.
`-dynamic`	Code intended for a dynamic library. This allows for building one in the future without changing the compiler options.
`-fno-common`	Make sure that there are no common blocks.
`-fexceptions`	Allow C++ exceptions to be thrown through Parasolid.
`-std=c99`	(C only) Compile code as ISO C99.
`-std=c++17`	(C++ only) Compile code as ISO C++17.
`-Wpartial-availability`	Warn when compiling calls to partially available functions.
`-Werror=partial-availability`	Make the partial availability warning into a compilation error. See Section 6.6.9, “MacOS and iOS backwards compatibility”.

6.2.9 Intel iOS

Parasolid was compiled, linked and tested on macOS 11.0. The C compiler used was Apple clang version 12.0.0 (clang-1200.0.32.27) from Xcode 12.2. Parasolid was tested on the iOS Simulator from the same version of Xcode.

Please see Section 6.3.6 for expected changes to this platform's build and test environment.

The following options were used:

```
-arch x86_64 -isysroot
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneS
imulator.sdk -mios-version-min=13.0 -fstack-protector-strong -O2  -fPIC -dynamic -fno-common
-fexceptions -std=c99 -Wpartial-availability -Werror=partial-availability
```

The same compiler was used for C++, with the `-std` option changed to `-std=c++17`.

Option	Description
`-arch x86_64`	Compile 64-bit x86 code.
`-isysroot...`	Use the iOS Simulator SDK headers supplied with Xcode 12.2.
`-ios-version-min=13.0`	Compile and link for iOS 13.0 onwards. See Section 6.6.9.
`-fstack-protector-strong`	Use improved stack protection. See Section 6.5.8.
`-O2`	Optimisation level 2.
`-fPIC`	Generate position-independent code.
`-dynamic`	Code intended for a dynamic library. This allows for building one in the future without changing the compiler options.
`-fno-common`	Make sure that there are no common blocks.
`-fexceptions`	Allow C++ exceptions to be thrown through Parasolid.
`-std=c99`	(C only) Compile code as ISO C99.
`-std=c++17`	(C++ only) Compile code as ISO C++17.
`-Wpartial-availability`	Warn when compiling calls to partially available functions.
`-Werror=partial-availability`	Make the partial availability warning into a compilation error. See Section 6.6.9, “MacOS and iOS backwards compatibility”.

We intend to keep the Intel iOS builds of Parasolid as similar as possible to the ARM iOS builds.

6.2.10 ARM Android

Parasolid was compiled and linked on CentOS 7.9. The C compiler used was Android clang version 12.0.8, from NDK23b for Linux. Parasolid was tested on a Qualcomm Snapdragon® 8 Gen 1 Mobile Hardware Development Kit, running Android 12.0.

The following compiler options were used:

```
--target=aarch64-linux-android28 -O2 -fno-strict-aliasing
-ffp-contract=off -std=c99 -fPIC -fvisibility=hidden
-fexceptions -Wformat -Wformat-security
-fstack-protector-strong -D_FORTIFY_SOURCE=2
```

The same compiler was used for C++, with the `-std` option changed to `-std=c++17`.

Option	Description
`--target=aarch64-linux-android28`	Compile for 64-bit ARM Android, API 28, Android 9.0.
`-O2`	Optimisation level 2.
`-fno-strict-aliasing`	Optimisation should not make assumptions about aliasing.
`-ffp-contract=off`	Avoid the use of “floating-point contractions,” to improve consistency.
`-std=c99`	(C only) Compile code as ISO C99.
`-std=c++17`	(C++ only) Compile code as ISO C++17.
`-fPIC`	Generate position-independent code.
`-fvisibility=hidden`	Hide Parasolid’s internal symbols.
`-fexceptions`	Allow C++ exceptions to be thrown through Parasolid.
`-Wformat` `-Wformat-security`	Warn of security risks with the format strings used by the printf/scanf family of functions.
`-fstack-protector-strong`	Use improved stack protection. See Section 6.5.8.
`-D_FORTIFY_SOURCE=2`	Use more-secure C string functions.

The Parasolid shared library is linked using the following options:

Option	Description
`-z relro`	Make the relocation table read-only.
`-z now`	Load the whole library as soon as any part of it is called.
`-z noexecstack`	The library does not require an executable stack.
`-z defs`	Make sure there are no undefined symbols.
`--no-rosegment`	Work around an Android 9 limitation.

Parasolid is no longer linked with “`--hash-style=both`” because all Android 9 and later implementations support “gnu” symbol table hashes.

Note: You can develop Parasolid-based Android apps on Windows, Linux or macOS. The libraries, headers and so on can be used by Android development tools on any platform. Parasolid is built on Linux simply because that was convenient.

Note: No Intel build of Parasolid is supplied. There is no need for a separate simulation platform, unlike iOS, because Android development toolsets invariably include an emulator that runs on Intel and AMD machines, and executes an ARM-based Android OS and ARM-based apps. An Intel Android build would only be useful in developing for Intel-based Android devices.

6.3 Supported operating systems and hardware

This section describes the environments in which Parasolid can run for each platform. It contains information for all platforms.

6.3.1 Windows platforms

6.3.1.1 Supported versions of Windows

Windows 7. Windows 8.1

Windows 7 is no longer supported by Microsoft, but several customers have asked us to keep Parasolid compatible with it, since they expect demand for it in the future. Therefore:

- We will continue to compile Parasolid for compatibility with Windows 7.
- We will do some basic testing on Windows 7, while this is practical.
- We will not formally support Windows 7, which means that fixing Windows 7-specific problems (which have been extremely rare in practice) will be at our discretion.
- We have not had requests for Windows 8.1 compatibility, but it is provided automatically by Windows 7 compatibility.

We intend to continue this scheme until the end of Microsoft support for Windows Server 2012 R2, described below. However, we cannot commit to this, because there is potential for future changes to Windows, Visual Studio, virtual machine systems or security standards to make it impossible. We will, however, make all reasonable efforts to continue with it.

Windows Server 2012 R2

Windows Server 2012 R2 will cease to be supported by Parasolid on 10-OCT-2023, when Microsoft support for this operating system ends.

Windows 10 and Windows 11

We support Parasolid on the versions of Windows 10 and 11 that are still under support from Microsoft, including their server and embedded versions, where provided. We will cease supporting Parasolid on each version of Windows as and when Microsoft withdraws support for it. The currently supported versions are:

Windows	Version	Codename	Build	Server version	End of support
10	1507	Threshold 1	10240		14-Oct-2025
10	1607	Redstone 1	14393	2016	13-Oct-2026
10	1809	Redstone 5	17763	2019	09-Jan-2029
10	21H2	21H2	19044	2022	12-JAN-2027
10	22H2	22H2	19045		13-MAY-2025
11	21H2	Sun Valley	22000		08-Oct-2024
11	22H2	2022 Update	22621		14-OCT-2025

Note: Windows 10 Versions 1507, 1607, 1809 and 21H2 have long periods of support from Microsoft, 1507 because it was the initial release, and 1607, 1809 and 21H2 because they have server versions. These are all “Long Term Servicing Channel” or “LTSC” versions in current Microsoft terminology.

6.3.1.2 X64 WIN

Parasolid will run on 64-bit supported versions of Windows.

Parasolid will run on any Intel or AMD 64-bit x86 processor.

Parasolid is built and tested for the traditional 64-bit WIN32 desktop programming environment. It is not supported on the WinRT programming environment, or the Universal Windows Platform (UWP) environment, and is known not to work in those environments. It is not officially supported in Windows 10 applications for the Windows Store that make use of the Desktop Conversion Extension, but we would expect it to work in that environment.

See Section 6.4.1, “X64 WIN” for details of the effects of the Visual Studio 2013 and later 64-bit math libraries when run on processors that support the AVX2 instruction set.

6.3.1.3 ARM WIN

Parasolid runs on 64-bit supported versions of Windows for ARM. It is built for the 64-bit WIN32 environment; see above for variant environments.

See Section 5.6.1.5, “Processor affinity” for issues with processor affinity on ARM WIN.

6.3.1.4 Intel NT

Parasolid will run on 32-bit or 64-bit supported versions of Windows, on any Intel or AMD x86 processor that supports the SSE2 instruction set. It is built for the 32-bit WIN32 environment; see above for variant environments.

The Parasolid DLL has additional version information, allowing you to confirm that you are using an SSE2 build via Windows Explorer. See Section 6.7, “Parasolid version information”, for more details.

6.3.2 Intel Linux

Default build Parasolid will run on any Intel or AMD 64-bit x86 processor. Parasolid is supported on Red Hat Enterprise Linux 8 onwards, Alma Linux 8 onwards, Rocky Linux 8 onwards, SUSE Linux Enterprise Desktop/Server version 15sp3 onwards, Amazon Linux 2023 onwards, and Ubuntu LTS 20.04 onwards.

It should run on any x86-64 Linux with glibc 2.28 or later and the GCC 8 or later versions of libgcc_s and libstdc++.

Warning: This default build *will not run* on CentOS 7, Red Hat Enterprise Linux 7 or SUSE Linux Enterprise 12. For those, see the GCC7 compatible build below.

Note: The Red Hat GCC Toolset we use to provide GCC 11.2 statically links any C and C++ language support functions required by Parasolid which are not available in the GCC 8 versions of libgcc_s and libstdc++.so provided with Red Hat Enterprise Linux 8.

GCC7-compatible extra build

This build is compatible with our GCC7 build of Parasolid, provided at Parasolid versions 34.1 to 35.1. It will run on any Intel or AMD 64-bit x86 processor. Parasolid is supported on CentOS 7, SUSE Linux Enterprise Desktop/Server version 12 onwards, Red Hat Enterprise Linux 7 onwards, and Ubuntu LTS 20.04 onwards.

It should run on any x86-64 Linux with glibc 2.14 or later and the GCC 4.8 or later versions of libgcc_s and libstdc++.

Warning: Versions 7.9 of CentOS and Red Hat Enterprise Linux will leave support from Red Hat on 30-Jun-2024. We hope to be able to continue support of this build until 31-Dec-2024, if we can do so while meeting proper security standards. However, we cannot yet commit to this.

Note: The Red Hat Developer Toolset we use to provide GCC 11.2 statically links any C and C++ language support functions required by Parasolid which are not available in the GCC 4.8 versions of libgcc_s and libstdc++.so provided with CentOS 7. This allows this build to run on the same set of Linux distributions as our former GCC 4.8 and GCC 7.3 builds.

6.3.3 ARM Linux

Parasolid will run on any 64-bit ARM processor. Parasolid is supported on Red Hat Enterprise Linux 8 onwards, Alma Linux 8 onwards, Rocky Linux 8 onwards, SUSE Linux Enterprise Desktop/Server version 15sp3 onwards, Amazon Linux 2023 onwards, and Ubuntu LTS 20.04 onwards.

It should run on any ARM64 Linux with glibc 2.28 or later and the GCC 8 or later versions of libgcc_s and libstdc++.

Warning: This build *will not run* on Amazon Linux 2, our former build standard.

6.3.4 ARM MacOS

Parasolid runs on macOS 11.0 or later on any ARM-based “Apple Silicon” Macintosh.

6.3.5 Intel MacOS

Parasolid runs on macOS11.0 or later on any Intel-based Macintosh.

6.3.6 ARM iOS

Apple now provides two separate operating systems in the iOS family. These are iOS, for iPhones (and iPods) and iPadOS, for iPads. Although they use the same software development kits and tools, Apple does not provide any undertakings about compatibility between them. This change took place at iOS 13, which was accompanied by iPadOS 13. We develop and test Parasolid using iPad Pros, because they have more RAM than iPhones, and some of our test cases reach or exceed the RAM capacity of the largest currently available iPad Pros.

- Parasolid runs under iPadOS 13, 14, 15 or 16 on any iPad that is supported by those operating systems.
- We expect Parasolid to work under iOS 13, 14, 15or 16 on any iPhone or iPod that is supported by those operating systems, but it has not been tested and is not formally supported.

Since, by mobile device standards, a fully-featured solid-modeling application may use a lot of processor time and memory, we recommend that your apps require an iPad Pro with at least 4GB RAM.

Parasolid uses the basic ARMv8 64-bit instruction set, rather than any of the later enhanced instruction sets, so as to be compatible with older hardware.

Apple’s tvOS and watchOS are related to iOS, but use separate software development kits. We do not consider either to be relevant for Parasolid.

Warning: Given the pace of Apple’s release of new iOS versions and development tools, it is not always practical for us to maintain a Parasolid release with the same development tools, nor to test it on the same version of iOS, throughout its maintenance life. This is different from our practice with other, non-iOS platforms.

It is possible that during the life of this release, we will switch to building Parasolid with a later version of Xcode, possibly more than once. When we change Xcode versions, subsequent patch releases may no longer run on the minimum version of iOS supported at the First Customer Ship (FCS) release. The FCS release, and patch releases made before the change of Xcode, will remain compatible with the original minimum iOS.

6.3.7 Intel iOS

Parasolid runs on the iOS Simulator for iPadOS 13, 14, 15 or 16, on any 64-bit Intel-based Macintosh with an appropriate version of Xcode installed. The warning for ARM iOS above also applies to this platform. We intend to keep the Intel iOS builds of Parasolid as similar as possible to the ARM iOS builds.

6.3.8 ARM Android

Parasolid runs on Android 9.0 or later on any ARM-based device running a 64-bit version of Android. Since, by mobile device standards, a fully-featured solid-modelling application may use a lot of processor time and memory, we recommend that your apps require a fast device with at least 4GB RAM.

> **Note:** Since Parasolid contains C++ code, you will need to add the C++ run-time library, `libc++_shared.so` from the Android NDK to your app bundle, if it was not already included. Android does not provide a "system" C++ run-time library, but requires each app that uses C++ to include its preferred C++ run-time. This uses slightly more storage, but means that the C++ standard available does not depend on the operating system version.

Parasolid should also be runnable in the Android ARM emulator, although its size demands a powerful host machine, and our efforts have had limited success.

Parasolid uses the basic ARMv8 64-bit instruction set, rather than any of the later enhanced instruction sets, so as to be compatible with older hardware.

See also Section 6.6.10, "Android changes for NDK developers".

6.4 Floating-point information

Where practical, we recommend that you enable floating-point traps (that is, allow the operating system to catch run-time errors arising from floating-point operations).However, this is not practical on many of the supported platforms. See below for details of how to set up the recommended configurations on each platform, or the reasons why this is impractical.

The errors that we recommend that you enable traps for are:

- Invalid operation
- Divide by zero
- Overflow

You should also register a signal handler to handle the signal (SIGFPE) caught by the operating system in these cases; see Chapter 122, "Signal Handling" of the *Functional Description* manual for details.

> **Warning:** You must *not* enable traps for these floating-point errors:
>
> - Underflow
> - Denormal operand
> - Inexact result
>
> Parasolid relies on numbers being rounded naturally, including rounding to zero, and will not operate correctly if the above floating-point errors are trapped by the OS.

For more information about floating-point traps, see Appendix A, "Floating-Point Traps". For more information about floating-point traps and Parasolid SMP, see Section 5.6.3, "Run-time error handling".

6.4.1 X64 WIN

The recommended floating-point traps are disabled by default on this platform. They are enabled by the line:

```
#include <float.h>
_controlfp(_EM_DENORMAL | _EM_UNDERFLOW | _EM_INEXACT, _MCW_EM);
```

Note that this line specifies those floating-point traps to be *disabled*; all other traps are consequently enabled. You should consult the Microsoft documentation for `_controlfp()` before making use of this function.

The use of denormals as zeroes is recommended on this platform and can be enabled with the following additional line:

```
#include <float.h>
_controlfp(_DN_FLUSH, _MCW_DN);
```

6.4.1.1 FMA3 instructions

Parasolid is compiled to use the basic x86-64 instruction set, and does not use AVX, AVX2 or FMA3 instructions. However, Microsoft's 64-bit math library contains code paths that use FMA3 combined floating-point-multiply-add instructions. These paths will be used by default on processors that support them, such as the Intel “Haswell” series, AMD “Piledriver” series and their successors.

Microsoft's objective in this is increased performance, but Parasolid makes sufficiently few calls to the math library that this has no detectable effect on Parasolid's speed in our trials. However, the occasional slight differences in rounding caused by combining a multiply and an add into a single instruction can cause math library functions to produce slightly different results. These differences can cause a Parasolid modeling operation to fail where it would succeed on a processor without FMA3 instructions. In the same way, a Parasolid operation can succeed on a processor with FMA3 instructions when it would fail on a processor without them. These changes do not happen often (less than one in a thousand tests in our trials), but they are common enough to be noticed. Parasolid's algorithms are robust against most instances of such differences, but without human knowledge of what's “right” and what isn't, perfection is unattainable. The differences are similar in number and scope to those caused by a change of compilers, but depend on the hardware in use at run-time.

If you want to turn off the use of FMA3 instructions on processors that support them, you can do so with the line:

```
#include <math.h>
_set_FMA3_enable(0); /* 0 = disable, 1 = enable */
```

Doing this only affects the process that makes the call to `_set_FMA3_enable()`, but it affects all code in that process that was built with the same version of Visual Studio. This happens because it is a setting within an individual run-time library, and different versions of Visual Studio can use different run-time libraries (However, Visual Studio 2015 to 2022 share the same run-time libraries, see Section 6.6.1, “Windows run-time libraries”).

You should consult the Microsoft documentation for `_set_FMA3_enable()` before making use of this function.

If your program contains code built with more than one version of Visual Studio that supports FMA3, then if you change this setting, you should change it in code compiled by *each* version of Visual Studio. If different run-time libraries are using different FMA3 settings, the calculations performed by code using different run-time libraries won't be entirely consistent, which may have unpredictable results.

This issue, as of Visual Studio 2015 to 2022, only affects 64-bit code.

6.4.2 ARM WIN

Floating-point traps are an optional feature of the ARM architecture, and Windows 10 for ARM does not allow them to be enabled.

6.4.3 Intel NT

Parasolid is compiled with the `/fp:precise` option, which is the compiler's default. Intel NT builds of Parasolid on older compilers used `/fp:fast`, which is slightly faster, but provides worse consistency. You can use either setting for your applications' code, but `/fp:precise` is preferable, if its performance is acceptable.

Do not call Parasolid with the x87 floating-point unit set to any evaluation precision other than 53-bit precision (64-bit values), since it will only operate correctly with that setting. All floating-point values passed to, and returned from, Parasolid should be standard 64-bit values.The recommended floating-point traps are disabled by default on this platform. They can be enabled using code such as:

```
#include <float.h>
_controlfp(_EM_DENORMAL | _EM_UNDERFLOW | _EM_INEXACT, _MCW_EM);
```

Note that this line specifies those floating-point traps to be *disabled*; all other traps are consequently enabled. You should consult the Microsoft documentation for `_controlfp()` before making use of this function.

Note that this Parasolid platform uses both x87 and SSE2 registers and instructions, but that Windows does not generate the SIGFPE signal when SSE2 instructions cause traps. It is reasonably simple to implement this yourself with some code in the `main()` function of your program: see Section 6.10, "Enabling SIGFPE for SSE2 on Intel NT". Even if you do not use signals for error handling, see that section for details of the necessary calls to `_fpreset()`.

It is possible to use denormals as zeroes on this platform, and this gives a small improvement in performance. However, you may wish to avoid the practice. The SSE2 floating-point registers support them but the x87 registers do not; Parasolid uses both, with the choice for any individual calculation being made by the compiler. Our trials have been unable to detect any significant differences in Parasolid test results when using denormals as zeroes, but the possibility remains. To use denormals as zeroes, use the following call:

```
#include <float.h>
_controlfp(_DN_FLUSH, _MCW_DN);
```

6.4.4 Intel Linux

This platform disables the recommended floating-point traps by default. You can enable them with the following code:

```
#include <fenv.h>
feenableexcept( FE_OVERFLOW | FE_INVALID | FE_DIVBYZERO);
```

6.4.5 ARM Linux

Floating-point traps are an optional feature of the ARM architecture, and the Linux operating system has to be able to run with or without them, depending on hardware manufacturers' choices. It manages this by always leaving them turned off, and does not supply a way for software to turn them on. The recommended floating-point traps thus cannot be enabled.

6.4.6 ARM MacOS

The recommended floating-point traps are disabled by default on this platform, and should not be enabled. The Clang compiler does not attempt to generate code that can safely be run with floating-point traps enabled. At optimization levels -O1 and -O2, it will sometimes move code that can cause a floating-point trap outside the scope of a test intended to prevent the trap. ARM-based versions of macOS also report floating-point traps as illegal instructions, making them indistinguishable from program corruptions.

6.4.7 Intel MacOS

The recommended floating-point traps are disabled by default on this platform, and should not be enabled. The Clang compiler does not attempt to generate code that can safely be run with floating-point traps enabled. At optimization levels -O1 and -O2, it will sometimes move code that can cause a floating-point trap outside the scope of a test intended to prevent the trap.ARM iOS

The recommended floating-point traps are disabled by default on this platform. They should not be enabled, for the same reasons as ARM MacOS. Intel iOS

The recommended floating-point traps are disabled by default on this platform. They should not be enabled, for the same reason as Intel MacOS.

6.4.8 ARM Android

Floating-point traps are an optional feature of the ARM architecture, and the Android operating system has to be able to run with or without them, depending on device manufacturers' choices. It manages this by always leaving them turned off, and does not supply a way for apps to turn them on. The recommended floating-point traps thus cannot be enabled.

6.5 Solutions to platform-specific issues

This section describes platform-specific issues and situations that you should avoid. It contains information on all relevant platforms.

6.5.1 64-bit and 32-bit code

Applies to: X64 WIN and Intel NT

We recommend that you build your applications as 64-bit code, for Windows 10 and later operating systems. If this is not practical, we recommend that you build as 32-bit code for Windows 10 and later.

Applies to: Intel NT

If you wish to use this platform, note that *all* of your application has to be compiled as 32-bit code. Windows does not support mixing 32-bit and 64-bit code in the same program.

Applies to: ARM Android

Parasolid is only provided for the arm64-v8a ABI. It is not possible to build Parasolid-based apps for any other Android ABI.

6.5.2 Data alignment issues

Applies to: Windows platforms

Do not use the `/Zp` compiler option, which modifies the Microsoft C/C++ compiler's structure packing rules. Parasolid is compiled with the Microsoft C/C++ compiler's default structure packing, and if structures compiled with different packing are passed to it, it will crash.

6.5.3 Using a different compiler from Parasolid

Applies to: Windows platforms

It is possible to build applications for these platforms with a different compiler to that used by Parasolid, although it is not recommended or officially supported. If you do this, any calls to the `matherr()` function made by the run-time library used by Parasolid will not be caught by your application, and FMA3 settings can be inconsistent between different parts of an application. See also Section 6.9, “Linking Windows run-time libraries”.

Applies to: ARM Android

If you need to use a different NDK from Parasolid, note that the minimum Android API standard required by your app will be the highest API standard required by any of its components.

6.5.4 Handling signals in C++ applications

Applies to: Windows platforms

C++ applications built in the Visual Studio environment can’t, by default, handle signals. To do this, some non-default settings and some support code are required. *Chapter 122, “Signal Handling”, in the Functional Description* provides a detailed explanation of signal handling, and Microsoft’s documentation is the definitive reference on use of their tools. The steps needed to tie these things together are:

- Enable your application to catch a “Structured Exception”, by compiling it with the `/EHa` option.
- Register a “translator” function with the C++ runtime, via `_set_se_translator()`. Note that this registration is thread specific, and thus needs to be done for every thread that enters Parasolid.
- Then, when there is a signal from the OS, the C++ runtime catches it and calls the translator function.
- The translator function should call PK_THREAD_is_in_kernel, and then call PK_SESSION_abort if PK_THREAD_is_in_kernel showed that Parasolid was executing.
- Parasolid then calls the application’s registered error handler, and this error handler can then throw a C++ exception, which the application should catch to allow recovery rather than exit.
- If PK_THREAD_is_in_kernel did not indicate that Parasolid was executing in the relevant thread, then the translator function can throw a C++ exception when it sees fit.

See the following sections:

- Section 5.6.3, “Run-time error handling”
- Section 6.4, “Floating-point information”
- Section 6.10, “Enabling SIGFPE for SSE2 on Intel NT”

6.5.5 Memory limitations

Applies to: X64 WIN, ARM WIN, Android, iOS and MacOS

X64 WIN and ARM WIN programs that use Parasolid always require more stack space than the default provided by the platform’s linker. Android, iOS and MacOS require larger than default stack sizes for application threads which will call Parasolid and for Parasolid internal SMP threads (on relevant platforms). See Section 5.5, “Stack sizes”, for more details.

6.5.6 Memory consumption by Parasolid

Applies to: Android, ARM iOS and Intel NT

These platforms impose limitations on the amount of memory that can be used by software:

- Android and ARM iOS are limited to the physical memory of the device they run on, some of which will be reserved for the operating system.
- Intel NT is limited to 4GB of address space.

If you produce software for these platforms, we recommend that you take special care in testing for and handling out-of-memory conditions. This applies to Parasolid memory use in general, but meshes and lattices are especially likely to use large amounts of memory.

Applies to: ARM WIN

While this platform runs on ARM-based devices with no hard disk, Windows requires that such devices be capable of “swapping” data between physical memory and storage. ARM WIN programs are therefore *not* limited to the physical memory of the device they run on.

6.5.7 Posix threads library

Applies to: Linux platforms

You will need to use the `-lpthread` parameter when linking, because the Parasolid library contains references to the POSIX threads library. See Section 5.6, “Symmetric Multi-Processing (SMP)” for details. On macOS, iOS, and Android platforms, the POSIX threads functions are included in the standard system library.

6.5.8 Stack protection

All of the platforms on which Parasolid is supported can use “stack protection”, also known as “buffer overflow protection” or “buffer overrun protection”, provided by the compiler. This is a safety precaution against attempts by computer viruses or other “malware” to take over a program.

A program’s stack is an area of memory, used to hold the local variables of the functions that make up the program. When a function is called, it usually uses stack memory for variables, and when it exits and returns to its caller, the memory used by those variables is freed for use by subsequent functions. The stack is also used to store the “return address” of each function call, the place where the caller function should resume running when the called function terminates.

If a program can be persuaded or tricked into storing more data in a variable on the stack than will fit, then this can overwrite the return address. The simplest way of doing this is to give a function that reads a string, from a file or a network message, a longer string than the variable will hold. If the function doesn’t check the length and stop when it reaches the limit, then it overwrites whatever lies beyond the variable, which is called “overrunning” the variable. Variables that hold strings are sometimes known as “buffers”.

The point of overwriting the return address is that when the function tries to return, it does so to an address controlled by whoever created the data that was read. This is why many security flaws can only be exploited via specially crafted files or network messages.

Stack protection systems work by means of compiler-inserted code at the start and end of vulnerable functions. The code at the start stores a hard-to-predict value (a “cookie” or “canary”) on the stack between the variables and the return address. The code at the end checks to see if the value has changed: if so, it is assumed that some variable has been overrun and the return address has been overwritten, corrupting the stack.

The usual response to a stack corruption is to terminate the program. It is not generally possible to recover from this condition, because there is no way to tell how far the corruption has spread.

Applies to: Windows Platforms

Parasolid is compiled with the `/GS` (GuardStack) option, which implements stack protection, in accordance with the Microsoft Security Development Lifecycle (see Section 6.6.2).

If the protection code detects stack corruption, it generates an exception, which normally goes unhandled, and terminates the program. If you have a debugger installed, Windows will offer you the chance to debug the program, and the debugger will tell you that there has been a stack buffer overrun.

Applies to: Linux Platforms

Parasolid is compiled with the `-fstack-protector-strong` option, which implements stack protection.

If the protection code detects stack corruption, it will output the message “stack smashing detected” to standard error. It will then call the run-time library function `abort()`. This will raise UNIX signal 6, SIGABRT, whose default handler will terminate the program. If you have installed a handler for this signal, it will be called. If your handler returns, or you have requested that SIGABRT be ignored, the program will be terminated.

Applies to: MacOS and iOS

Parasolid is compiled with the `-fstack-protector-strong` option, which implements stack protection.

If the protection code detects stack or heap corruption, your program will be terminated, and the message “Abort trap: 6” sent to standard error. This is done by raising UNIX signal 6, SIGABRT. It is not possible to trap this signal and report it differently, because the protection code disables any SIGABRT handler immediately before raising the signal.

Applies to: Android

Parasolid is compiled with the `-fstack-protector-strong` option, which implements stack protection.

If the protection code detects stack corruption, it will add the message “stack corruption detected” to the Android log, and send it to standard error. It will then call the run-time library function `abort()`. This will raise UNIX signal 6, SIGABRT, whose default handler will terminate the program. If you have installed a handler for this signal, it will be called. If your handler returns, or you have requested that SIGABRT be ignored, the program will be terminated.

6.5.9 iOS and Android message handling

Since Parasolid modeling operations can take significant amounts of time, and their duration can be hard to predict, it is important that you do not call Parasolid from the main thread of your app (on Android, the UI thread). If you do this, the application risks becoming unresponsive to messages from the operating system.

- iOS may then kill it, assuming it has hung.
- Android will display an “Application Not Responding” message and wait for user input.

Create a separate thread to call Parasolid, and make sure it has sufficient stack space (see Section 5.5, “Stack sizes”).

6.6 Supporting material for Parasolid libraries

This section describes additional relevant material that you will find useful.

6.6.1 Windows run-time libraries

Visual Studio 2015 marked an important change in Microsoft's provision of run-time libraries. Before that version, each version of Visual Studio had its own version of the C/C++ run-time library, or "CRT", and building programs that used a mixture of versions of Visual Studio required special care. Since Visual Studio 2015, all versions of Visual Studio have used the same filenames for their CRT:

1. The "Universal CRT", `UCRT.DLL`.
2. The `api-ms-win-crt` family of DLLs, which provide access to the Universal CRT.
3. The compiler-specific support functions, `VCRUNTIME140.DLL`.

Visual Studios 2015 to 2022 are all "the same" compiler in some respects, since they all use the same CRT. Visual Studio 2019 added an additional file, `VCRUNTIME140_1.DLL`, to support improvements to C++.

Run-time library compatibility

Versions of Parasolid built with Visual Studio 2022 make use of the C++ improvements introduced at Visual Studio 2019, and thus require the run-time libraries from Visual Studio 2019 or 2022.

Installing run-time libraries

To be able to run programs that use Parasolid, you will need to install the relevant redistributable CRT. These are supplied with the relevant Visual Studio, and will be found, by default, in the following directories:

Visual Studio 2022

```
c:\program files (x86)\Microsoft Visual Studio 17.0\VC\redist\MSVC\14.30.30704\vc_redist.x64.exe
c:\program files (x86)\Microsoft Visual Studio 17.0\VC\redist\MSVC\14.30.30704\vc_redist.x86.exe
c:\program files (x86)\Microsoft Visual Studio 17.0\VC\redist\MSVC\14.30.30704\vc_redist.arm64.exe
```

Warning: None of the Visual Studio run-time libraries are provided as part of any version of Windows.

Microsoft recommends that you install and use the latest libraries, as they may contain significant bug fixes.

When you install Visual Studio 2022 on a computer that already has Visual Studio 2015, 2017 or 2019 run-time libraries installed, the older Visual Studio run-time libraries will apparently disappear from lists of installed programs. They have been replaced in the list by the Visual Studio 2022 run-time libraries, which will serve programs built with all versions of Visual Studio from 2015 onwards.

See also Section 6.9, "Linking Windows run-time libraries".

6.6.2 Software "hardening" of Parasolid

Microsoft Security Development Lifecycle

Parasolid complies with the Microsoft Security Development Lifecycle (MS-SDL). This is a set of standards intended to reduce the incidence of security vulnerabilities in Microsoft Windows applications. Microsoft recommends that all Windows software producers should use MS-SDL. We have done so because of customer requests. This does not make software perfectly secure, but it makes a would-be intruder's task harder.

Hardening for other Parasolid platforms

Parasolid now has “hardening” broadly equivalent to MS-SDL on the Linux and Android platforms. This is explained below using MS-SDL terminology.

Parasolid on Linux complies with many of Red Hat's recommendations for compiler and linker options. It omits some that are more appropriate for open-source software running as part of the operating system.

6.6.2.1 Data Execution Protection (DEP)

Parasolid DLLs for Windows platforms are linked with the `/NXCompat` linker option, to mark them as capable of running with DEP active. DEP is normally active on all versions of Windows supported by Parasolid, if all EXEs and DLLs used by a program are marked `/NXCompat`. Parasolid operates correctly with Data Execution Protection (DEP) checking active for Parasolid itself and the KID test harness.

All other Parasolid platforms provide equivalents of DEP, which are used automatically.

6.6.2.2 Address Space Layout Randomisation (ASLR)

Parasolid DLLs for Windows platforms are linked with the `/DynamicBase` linker option, to mark them as capable of running with the ASLR security measure, introduced at Windows Vista. This is only used if all EXEs and DLLs used by a program are marked `/DynamicBase`. Parasolid built with Visual Studio 2017 or later allows Windows to use more entropy with ASLR for 64-bit code, strengthening the protection.

ASLR can make it much harder to debug some kinds of code problems, notably bad pointers and memory overwrites. If you suspect that you are experiencing such problems, you can remove the `/DyamicBase` marker with:

```
editbin /DynamicBase:NO pskernel.dll
```

Note that removing this marker will invalidate the Authenticode signature of the DLL. Do not distribute any DLLs that you have changed in this way; reserve them for debugging.

All other Parasolid platforms provide equivalents of ASLR, which are used automatically.

6.6.2.3 Safer Structured Exception Handling

Parasolid DLLs for Windows platforms are linked with the `/SafeSEH` linker option, to ensure that they contain a table of all their Structured Exception Handling (SEH) exception handlers. This can be used by Windows as a countermeasure to new exception handlers being uploaded through security exploits.

Note: If you link your own Parasolid DLL, or statically link Parasolid to your application, we recommend that you use the `/NXcompat`, `/DynamicBase` and `/SafeSEH` linker options. `/SafeSEH` is used automatically when linking 64-bit code and you do not need to specify it explicitly.

Structured Exception Handling is unique to Microsoft Windows. Other platforms have no need to secure a feature they don’t possess.

6.6.2.4 Structured Exception Handler Overwrite Protection (SEHOP)

This is a feature of all versions of Windows supported by Parasolid, which improves on /SafeSEH, but has to be turned on for each application on a desktop Windows system. It isn't possible for Parasolid, which is a library, to turn it on, but we recommend that your products' installers do so. For full details, see the "Let me enable it myself" section of: `https://support.microsoft.com/kb/956607`.

Again, Structured Exception Handling is unique to Windows, so other platforms have no need to secure a feature they don't possess.

6.6.2.5 GuardStack

Parasolid is compiled with the GuardStack compiler option (`/GS` compiler option) so as to provide stack protection and comply with MS-SDL. We recommend that you consider using the `/GS` option for your application code. Code that performs network I/O is best compiled with the source pragma `#pragma strict_gs_check`.

Using GuardStack has increased the size of Parasolid, and reduced its speed, but both losses are less than 2%, after some optimization work. Parasolid Support can provide advice on reducing the performance losses on request.

All other Parasolid platforms are compiled using the GCC or Clang option `-fstack-protector-strong`, which provides equivalent protection. For details of what happens when a stack buffer overflow is detected, see Section 6.5.8, "Stack protection".

6.6.2.6 Secure string handling

Parasolid on Windows uses the secure string functions from ISO/IEC TR24731 (e.g., `strcpy_s()` as required by MS-SDL), and no longer calls the traditional string functions banned by MS-SDL.

If the secure string functions detect a buffer overrun, they will generate an "Invalid Parameter Error". We recommend that your applications should catch and report these errors, rather than simply terminating. See MSDN for details of invalid parameter errors and the function `_set_invalid_parameter_handler()`.

Note: Parasolid still contains calls to the traditional memcpy() function. These calls do not originate from source code, but are inserted by the C/C++ compilers for copying structures, and by the C++ Standard Template Library. As as described in Microsoft's Visual Studio documentation for memcpy, these classes of call to memcpy are permitted in code that complies with MS-SDL.

The Linux and Android platforms are compiled with the `-D_FORTIFY_SOURCE=2` option, which causes the compiler to use versions of the traditional string functions that check the lengths of their arguments. If they detect a buffer overflow, they write the message "*** buffer overflow detected ***" to standard output and call `abort()`. This will raise UNIX signal 6, SIGABRT, whose default handler will terminate the program. If you have installed a handler for this signal, it will be called. If your handler returns, or you have requested that SIGABRT be ignored, the program will be terminated.

If these string handling functions are passed invalid arguments, such as null pointers, they write the message “Fatal exception: a secure string handling function received invalid parameters.” to standard output and call `abort()`, as described above.

Linux platforms are compiled with `-D_GLIBCXX_ASSERTIONS`, which provides length checking for C++ `std::string` objects. These are reported to standard output as normal assertion failures, followed by a call to `abort()`, as described above.

6.6.2.7 Heap metadata protection

The run-time libraries for all versions of Visual Studio that can be used with Parasolid automatically terminate a program when they detect corruption of the memory heap.

The other Plarasolid platforms do not have a precise equivalent of this feature, but do provide heap debugging facilities that partially fill this gap.

- Linux: See `man mcheck`,
- macOS and iOS: see https://developer.apple.com/library/archive/documentation/Performance/Conceptual/ManagingMemory/Articles/MallocDebug.html
- Android: see https://android.googlesource.com/platform/bionic/+/master/libc/malloc_debug/README.md

6.6.2.8 Pointer encoding

MS-SDL recommends, but does not require, that function pointers be passed and stored in an encoded form. This has not been implemented in Parasolid, since it would create a large amount of work for all customers, even those who do not chose to implement MS-SDL, and would make Parasolid completely incompatible with earlier versions.

6.6.2.9 .NET DLLs

.NET DLLs are always built with `/DynamicBase` and `/NXCompat`; they don't contain SEH exception handlers.

6.6.3 Windows icons

The Parasolid DLLs supplied for these platforms contain two Windows icons: a Parasolid “program” icon and a Parasolid “document” icon. They may be used in the user interface of Parasolid-based programs.

6.6.4 Windows digital signatures

The Parasolid DLLs for all the Windows platforms contain Authenticode® digital signatures (see MSDN). These signatures now use SHA-512 (or stronger) cryptographic hashes.

6.6.5 MacOS install names

Parasolid is supplied as a dynamic library, or “dylib”. Its default install name is `./libpskernel.dylib`. Use `otool` with the `-L` option to check the install names in, or used by, a dylib or executable.

You can use the utility `install_name_tool` with the `-id` option to change the Parasolid dylib's install name to the path name and filename where your application will store the Parasolid library: the Parasolid dylib was linked with the `-headerpad_max_install_names` option to allow for this. The reason for doing this is that the linker copies the install name in the Parasolid dylib into executables and dylibs that are linked against it. Therefore, if you change the name in the Parasolid dylib, you get the correct names in your own code automatically.

You can change the install name(s) that an executable (including `kid.exe`) uses for the dylibs it calls with the `-change` option of `install_name_tool`.

There are three special strings that may be useful to you when setting install names. `@executable_path` is replaced with the directory of the main executable of whatever program is loading the library. `@loader_path` is replaced with the directory of the executable or library that is loading the library. `@rpath` is replaced by the string(s) embedded in the application using the `-Wl,-rpath,<string>` option, and these string(s) may use `@loader_path`. Using these strings, it is possible to create applications that do not require installers and can be dragged and dropped into any directory. This follows the example of recent releases of Xcode, which is an exemplar of Apple’s application design. For more details, see “man ld” and “man dyld”.

6.6.6 MacOS memory limits

Most operating systems have a way to limit the amount of virtual memory that can be used by the operating system. macOS does not. It provides a dialog that informs the user of excessive memory usage, but their only course of action is to kill off applications.

Incorrect requests to Parasolid can cause it to request very large quantities of memory from its frustrum. As a safety precaution, you may wish to have your frustrum keep a running total of memory allocated through it and refuse to allocate more than, say, four times the physical memory of the machine.

6.6.7 iOS and Android memory limits

Since an iOS or Android device has no ability to “swap” memory out to its storage when an app tries to use too much memory, it is important that your Parasolid frustrum keeps a running total of memory allocated through it. Your app must set limits to prevent the frustrum allocating more memory than is available.

You may wish to use a similar memory limitation with the iOS Simulator. This is not mandatory, but it does make the simulator behave more like an iOS device.

6.6.8 MacOS security and digital signatures

We presume that Parasolid customers will want to make their applications compatible with Apple’s “Gatekeeper” security software. This requires signing all executables, dylibs, scripts, and other runnable parts of an application with an appropriate Apple signing certificate. Recent versions of Xcode can automate this process, and, by default, will apply the signature to everything signable in the application.

Parasolid family dylibs are not signed, and we do not currently intend to start signing them. This is because changing the install name of a library invalidates any signature. We feel it is important to give our customers the flexibility to set install names for whichever method they prefer their applications to use for locating libraries. Apple appear to regard signing as the final step of building a complete application, and Parasolid is a toolkit, not a complete application at all. See Section 6.6.5 above for details of setting install names.

Customers who download Parasolid from our website directly onto a Macintosh computer will find that the Parasolid dylib extracted from the `.tar.gz` file has a file-system attribute marking it as a quarantined file, created by the web browser and extraction tools. You should remove this attribute with the `xattr` command before building your application, since distributing a dylib with a quarantine attribute already on it is likely to cause confusion. See “man xattr” for details.

6.6.9 MacOS and iOS backwards compatibility

Apple’s system headers are annotated with the versions of macOS and iOS at which functions appear. The compiler uses the `-mmacosx-version-min` and `-mios-version-min` arguments to interpret those annotations.

Since macOS 10.12, and for all iOS and iPadOS versions, Parasolid is compiled with the `-Wpartial-availability` and `-Werror=partial-availability` options. These ensure that if functions were used that were not available at the minimum operating system to be supported, the calls would be treated as compilation errors.

For example, at Parasolid v36.0 the iOS builds of Parasolid are compiled for compatibility with iOS 13, using the iOS 14.2 headers supplied with Xcode 12.2. Using `-Wpartial-availability` and `-Werror=partial-availability` ensures that no calls to functions that aren’t available on iOS 13 can be compiled. This helps ensure that Parasolid is backwards compatible to iOS 13.

6.6.10 Android changes for NDK developers

The Android project maintains a web page of issues that developers who use the Android NDK need to take account of, at:
`https://android.googlesource.com/platform/bionic/+/master/android-changes-for-ndk-developers.md`

Parasolid complies with all of them, as of the production of this document. Specifically, Parasolid:

- Does not load any libraries with `dlopen()`.
- Provides only Gnu-style hashes.
- Uses only public APIs.
- Has full section headers.
- Has no text relocations.
- Uses only SONAMEs in DT_NEEDED entries in its ELF header.
- Has a correct SONAME.
- Has no DT_RUNPATH entry in its ELF header.
- Has no segments that are both writable and executable.
- Has a valid ELF header and section headers.
- Uses thread-local variables, but they all have trivial destructors.
- Does not attempt to hook, or detect hooking of, C library functions.
- Does not rely on the classic Bionic file paths holding regular files.
- Does not attempt to read any execute-only memory.
- Does not use any relative relocations.

6.6.11 No cryptography in Parasolid

Parasolid has no ability to encrypt or decrypt anything. There are no cryptographic algorithms within it. We are documenting this because there are countries that impose restrictions on the import and/or export of cryptographic software.

The code signatures of Windows DLLs and the “Enhanced Strong Names” of the .NET binding DLLs make use of cryptography, but Parasolid does not contain any cryptographic functionality that makes use of them. External cryptographic software, normally provided with Windows, is required to validate those signatures.

6.7 Parasolid version information

Version information is embedded into Parasolid DLLs (for Windows platforms), shared libraries (for UNIX and Android platforms) and archive libraries (for UNIX, Android and iOS platforms). You can use this information to find out exactly which version of Parasolid you are using, for example, when reporting problems to Parasolid Support. This section explains what information is available, and how you can view it on all platforms on which Parasolid is supported.

6.7.1 Viewing version information

To view the version information on Windows platforms, navigate to the Parasolid DLL using Windows Explorer. Right-click on the DLL file and select **Properties**, then click the **Version** tab.

Note: You can also hover the mouse pointer over the DLL file in Windows Explorer to display a subset of the available version information in a tool-tip popup window.

On UNIX platforms and Android, use `what libpskernel.so` (or whatever the shared library is called on your platform: see Section 4.1.2). If your operating system lacks a `what` command, there are basic UNIX commands that can substitute:

```
    strings libpskernel.so | grep '@(#)'
```

For iOS archive libraries, use `what libpskernel_archive.a`, in the Terminal application of your development Mac.

6.7.2 Version information details

The data items that are used are defined by the Windows VERSIONINFO resource, described in detail in the Microsoft Developer Network documentation library. Parasolid's usage of Windows resources is described below; all those not mentioned are used in the same way on all platforms.

Resource	Description
FILEVERSION, PRODUCTVERSION	Binary resources on Windows platforms, but strings on UNIX, Android and iOS platforms.
FILEOS	Binary on Windows platforms, "Darwin" on iOS, "Android" on Android, and the output of `uname` on UNIX platforms.
FILETYPE	Binary on Windows platforms, "Dynamic Library" on MacOS, and either "Sharable Library" or "Sharable Object" on UNIX and Android platforms, according to the particular platform's conventions. UNIX, Android and iOS archive libraries have "Archive Library".
LegalCopyright	Strings on Windows platforms use the © character, while strings on UNIX, Android and iOS platforms spell it out as "(C)".
LegalTrademarks	Strings on Windows platforms use the ™ and ® characters, while strings on UNIX, Android and iOS platforms spell them out as "(TM)" and "(R)".
SpecialBuild Description	This is used to label Intel NT builds of Parasolid as requiring SSE2 instructions (in addition to x87 instructions). It may be used for additional purposes in the future.

6.7.3 Version information files

The source file used to generate the version information for Windows DLLs and UNIX and Android shared libraries is supplied in the relevant Parasolid release kits. For UNIX and Android platforms, it is `libpskernel_resources.c`, which defines a set of C strings to be found by the `what` command. For Windows platforms, it is `pskernel.rc`, a source file for the Microsoft Resource Compiler. Both of these files contain conditional compilation commands, to allow you to use them, if you need to create your own DLL or shared library, while allowing them to be distinguished from the libraries found in a Parasolid release kit.

6.7.4 Inconsistencies with the kernel version

You should not regard the information from version resources as described here as the definitive source of Parasolid version information. If it is inconsistent with the values returned by the function PK_SESSION_ask_kernel_version, you should always assume the function to be correct.

6.8 .NET binding

Applies to: Intel NT and X64 WIN

Bindings for using Parasolid in .NET programs are supplied with the Intel NT platform (32-bit .NET), and the X64 WIN platform (64-bit .NET). For details of the binding see Chapter 12, “Calling Parasolid From .NET Code” in the Parasolid *Functional Description* manual.

The bindings were compiled with the Microsoft (R) Visual C# Compiler, as supplied with Visual Studio and updated by Windows security updates. The bindings for Visual Studio 2022 builds require version 4.8 of the .NET framework.

These bindings will not work with any earlier version of C# or the .NET Framework.

The .NET Framework is not yet available for ARM WIN, and thus no binding is supplied.

6.9 Linking Windows run-time libraries

This section explains various issues and restrictions that can apply to your use of the Microsoft C/C++ run-time library (“CRT”).

6.9.1 Recommended configuration

Parasolid is compiled with the `/MD` option, so that it uses the multi-threaded DLL version of the CRT. We recommend that you use the supplied Parasolid DLL, `pskernel.dll`, by linking its import library, `pskernel.lib`, rather than statically linking Parasolid.

It is simplest and best if your application is compiled using the same version of Visual Studio as Parasolid. The shipped version of your application should be compiled with the `/MD` option; this ensures Parasolid and your application are using the same copy of the CRT.

You will probably wish to create debug versions of your application for use during development, using the `/MDd` option. This makes your application use the debug multi-threaded DLL version of the CRT, which means that Parasolid and your application are linked against different copies of the CRT. This works correctly with Parasolid, because its frustrum architecture means that for many purposes (especially memory and file management) it is using your application's copy of the CRT, via function pointers provided by your application. Two caveats apply:

- Any calls to the `matherr()` function generated by Parasolid code will be made by Parasolid's copy of the CRT, and will not reach your application.
- If you alter the FMA3 setting on X64 WIN in such a program, you will need to do so for all the CRTs involved. See Section 6.4.1, “X64 WIN” for details.

6.9.2 Other configurations

Please review the material on the `/MD`, `/MDd`, `/MT` and `/MTd` C/C++ compiler options in the MSDN documentation library before attempting to use any of the configurations described in this section.

6.9.2.1 Statically linking Parasolid

If you wish to statically link Parasolid to your application, use the `pskernel_archive.lib` library. This only makes sense if all calls to Parasolid and all functions supplied to Parasolid from your application come from a single EXE or DLL, and your application is compiled with `/MD`. Trying to statically link Parasolid to code compiled with `/MDd` will not work, as the link will fail. You cannot statically link Parasolid to code compiled with `/MT` or `/MTd`: again, the link will fail.

6.9.2.2 Static linking to the CRT

If you need to link your application statically to the CRT, compile it with `/MT` or `/MTd` and link against the DLL Parasolid using `pskernel.lib`. This configuration is not recommended; removing the need for static linking of the CRT is preferable.

6.9.3 Using a different compiler from Parasolid

It is possible to use a different version of Visual Studio for your application than the version that was used to build Parasolid. This is not supported, but has been successfully accomplished by several Parasolid customers. You should definitely use the Parasolid DLL, and remember that Parasolid and your application will be linked against different CRTs, as described above. The Parasolid frustrum architecture allows this to work, under normal circumstances, but care and testing will be required, and the caveats above apply.

Note: Visual Studio 2015 to 2022 all use the *same* CRT, as described in Section 6.6.1, “Windows run-time libraries”.This means that, for the purposes of this section, they can be treated as being the same version of Visual Studio. We do not know if this situation will continue for future versions of Visual Studio.

6.10 Enabling SIGFPE for SSE2 on Intel NT

Modern processors for the Intel NT platform have two separate sets of floating-point registers and instructions: the x87 set and the SSE2 set. Compilers may generate code that utilizes both sets of registers. The complexity of using both sets of registers, and the limited support of 32-bit Windows for SSE2 floating-point traps, means that some extra code is required for the traps to be handled well.

6.10.1 Enabling floating-point traps

The `_controlfp()` function provided in the Visual Studio run-time library enables and disables floating-point traps for both the x87 and the SSE2 registers and instructions. You should consult the Microsoft documentation for `_controlfp()` before making use of this function.

6.10.2 SSE2 and SIGFPE

32-bit versions of Windows (including the 32-bit emulation of Windows inside 64-bit Windows) do not generate SIGFPE, or any other signal, in response to SSE2 traps. They may produce a dialog box, or other error notification, claiming to have detected “multiple floating-point traps”.

This is produced by the Structured Exception Handling system (SEH), the underlying mechanism used by Windows for handling all hardware traps including access violations, integer divide-by-zero errors and floating-point traps. SEH is described in the MSDN documentation library; it works somewhat like C++ exceptions, but is quite separate from them.

You can convert SEH exceptions into signals by putting a SEH `__try`/`__except` block in the `main()` function of a C or C++ program:

```
#include <windows.h>// For GetExceptionCode() and
                    // EXCEPTION_CONTINUE_SEARCH

extern void generate_signals ( int );

int main( int argc, char *argv[])
{
  __try
  {
    // Do the actual work of the program
  }
  __except(  generate_signals ( (int)GetExceptionCode()  ),
             EXCEPTION_CONTINUE_SEARCH )
  {};
}
```

This is an add-on to the normal method Windows uses for generating signals from hardware traps. If the program causes a hardware trap within the `__try{}` block, the operating system takes control of the thread that caused the trap and uses it to start executing SEH support code. This searches through the program’s stack looking for `__except(){}` blocks. This is not the same as C++ stack unwinding: while it is similar in that the stack is searched in the same manner, functions are not exited, destructors are not called, and so on. The search changes nothing itself: it merely searches.

Once the search has reached the `main()` function, the `__except` shown above gets control, and evaluates its “filter” expression: the part in `()` brackets. That calls the support function `generate_signals()`, which will be described below. If that function returns – indicating that it hasn’t handled the SEH exception – then the value EXCEPTION_CONTINUE_SEARCH is returned from the filter expression. The search then continues above `main()`, moving into the code in the C/C++ run-time library that calls `main()` to start the program. That has a `__try`/`__except` block that generates the default signals provides by the Microsoft run-time library.

Note: This code, and the code in the following section, is not required in 64-bit Windows applications.

6.10.3 The generate_signals() function

This accepts the output of the special function `GetExceptionCode()`, which can only be called in the filter expression of an `__except` block, and which provides a value that specifies which SEH exception has happened. It can then simply call your program's signal handler with an appropriate signal code.

```
#include <windows.h>// For STATUS_... codes
#include <signal.h> // For SIG... codes

void generate_signals( int seh_code)
{
  switch( seh_code)
  {
    case STATUS_FLOAT_MULTIPLE_TRAPS: // Generated by 32-bit
    case STATUS_FLOAT_MULTIPLE_FAULTS:// Windows for SSE2 traps
      my_signal_handler( SIGFPE);
      break;

    default:
      break;
    }
}
```

This makes a direct call to the signal handler function that you registered via the `signal()` function, and completes the necessary action. This example code works in the same way as the code in the Microsoft C/C++ run-time library. The source code for that library is supplied with all versions of Visual Studio – although installing it is optional – and the files in it that implement signals are `winxfltr.c` and `crtexe.c`.

Your signal handler should call the Microsoft C/C++ run-time library function `_fpreset()`when it handles a SIGFPE generated by the Microsoft run-time library. Note that threads can cause traps and signals independently of each other, and there is nothing in the C or C++ run-time libraries that automatically tidies up other threads when one hits a trap. Parasolid multi-threading handles this: when a thread causes a signal to be generated, and the signal handler calls PK_SESSION_abort, the thread that caused the signal (and is now running the internal error handler) waits until all the other threads have either finished their work, or taken the chance to abort themselves, and then continues with aborting the Parasolid call.

SEH works with C++ code as well as C code, although it is best to compile C++ code with the `/EHa` option, to make sure that the optimizer does not discard code that could be needed under SEH.

6.10.4 SSE2 traps and the Visual Studio debugger

If your version of Visual Studio does not catch SSE2 floating-point traps from 32-bit code, they can be added to it as follows:

- Look in your Windows header files for the hexadecimal values for `STATUS_FLOAT_MULTIPLE_FAULTS` and `STATUS_FLOAT_MULTIPLE_TRAPS`.
- Within Visual Studio, find the exceptions controls, as described in MSDN.
- Add `STATUS_FLOAT_MULTIPLE_FAULTS` and `STATUS_FLOAT_MULTIPLE_TRAPS` as Win32 Exceptions, with their own names, and the hexadecimal codes from the Windows headers.

Floating-Point Traps A

A.5 Introduction

Floating-point run-time errors are a sign that your program is processing invalid numerical values; these can come from programming errors or invalid input data. Some invalid values depend on the operation; for example, you are allowed to add zero to a number or multiply it by zero but division by zero is not permitted, hence zero is an invalid number for division.

Floating-point traps are a mechanism by which the hardware detects floating-point run-time errors. Unlike other run-time errors, you can decide whether the operating system reports these errors (via a signal) or ignores them: the right course of action depends on your application. Enabling a floating-point trap for a particular floating-point error ensures that the operating system raises the appropriate signal when that error occurs; disabling a floating-point trap means that no signal is raised, allowing processing to continue.

Note: Floating-point traps are sometimes called “interrupts” or “exceptions”; they must not be confused with C++ exceptions, however.

A.6 Types of floating-point trap

IEEE standard 754-1985, which is supported by all computers that can run Parasolid, defines the range and precision for floating-point numbers, various special values and conditions, and six floating-point traps. Of these six, three are recommended for use in Parasolid-based programs. The other three are not useful for modeling work; their applications are generally confined to pure mathematics.

Recommended floating point traps	Disabled result
Floating-point invalid operation	NaN
Floating-point divide-by-zero	NaN
Floating-point overflow	Infinity

Traps that should not be used with Parasolid	Disabled result
Floating-point underflow	Denormal value
Floating-point denormal operand	Denormal value
Floating-point inexact result	Rounded value

For more details of the exact meaning of IEEE floating-point traps, consult the processor manuals for your computer, which are usually available for download.

A.7 Enabling and disabling floating-point traps

The IEEE definitions of the floating-point traps allow for any combination of them to be enabled or disabled. Some processors have a more limited set of options, concentrating on the useful combinations, and some processors do not provide floating-point traps at all. Enabling or disabling floating-point traps is usually done by means of calls to the C run-time library: there are some examples in Chapter 6, “Environment”.

When a floating-point trap is enabled and an instruction executed that creates the appropriate error condition, the trap is activated and generates the signal SIGFPE. When a floating-point trap is disabled and an instruction executed that creates the appropriate error condition, the trap is not activated and no signal is generated. Instead, the instruction produces a special result, listed in the tables in Section A.6, “Types of floating-point trap” for each kind of trap and explained in Section A.8, “Special floating-point values”. Execution of the program then continues with the next instruction.

A.8 Special floating-point values

These are defined by the IEEE 754-1985 standard. While it may seem that floating-point numbers should “just work” and not have all these complexities, ideal mathematical numbers are not easy for computers to represent, and the best current approaches to the problem are hundreds to thousands of times slower than IEEE floating-point.

A.8.1 NaN (“Not A Number”)

This is a special value that essentially tells the processor: “This is not a number and cannot be used in mathematical calculations.” If the floating-point traps that can produce NaN are disabled, then once a single NaN is produced it will spread to any other numbers that are calculated using it. For example, any arithmetic operation involving a NaN will produce a result of NaN, irrespective of the other operands used.

A.8.2 Infinity

IEEE floating-point numbers have a limited range of possible values. An operation that produces a value outside this range has “overflowed” the range. If the corresponding floating-point trap is disabled, a special value that says “this is infinity” is produced.

A.8.3 Denormal value

As well as a limited range of values, IEEE floating-point numbers have limited precision. *Denormal* values are those too small to be represented with full accuracy, but which can still be distinguished from zero. Parasolid expects that these will be generated from time to time, and works happily with them. Enabling the floating-point traps that detect and raise signals for denormal values will prevent Parasolid from working properly.

A.8.4 Rounded value

A third limitation of IEEE floating-point numbers is that they cannot represent some values exactly. For example, ordinary decimal numbers cannot represent the fraction 1/3 exactly:

its value is 0.333333... recurring. This is normally dealt with by allowing the processor to round off the problematic value to the closest value that it can represent exactly. This is done automatically if the floating-point trap for an inexact value is disabled, and works very well. Enabling the floating-point traps that detect rounded values will prevent Parasolid from working properly.

A.9 Strategies for using floating-point traps

On most current operating systems, the default is that all floating-point traps are disabled. If floating-point traps are enabled, and one goes off, the operating system will generate the signal SIGPFE; control is then passed to a registered signal handler for SIGFPE if one exists, or the application is shut down if there is no handler for SIGFPE.

For guidance with platforms where the default configuration for floating-point traps is not the recommended one for use with Parasolid, see Section 6.4, "Floating-point information".

Some possible strategies for managing floating-point traps are given below.

A.1.9.1 Disable all floating-point traps

This works well for simple programs where it is acceptable for Not A Number or Infinity to propagate through the data the program produces. For larger programs, it may not be satisfactory, but it is the only option if the platform cannot generate floating-point traps, or if the compiler assumes that they will never be turned on, and takes advantage of this in its optimizer.

A.1.9.2 Enable recommended floating-point traps but do not handle the resulting SIGFPE

Chapter 6, "Environment" shows you how to enable floating-point traps on each platform. Programs do not handle signals by default; to do so, they must register signal handlers.

This approach makes sure that floating-point traps are detected, since the operating system will terminate any program that generates (and does not handle) a floating-point trap. However, it is not suitable for programs that are used interactively.

A.1.9.3 Enable recommended floating-point traps and handle the resulting SIGFPE

For most applications, this is the best strategy. For information about how to register a signal handler and how Parasolid interacts with one, see Chapter 122, "Signal Handling" of the *Functional Description* manual. Where a platform requires special treatment for generating or handling SIGFPE, this is described in Chapter 6, "Environment", of this manual.

A.10 Denormals as zeroes

Some processors have an optional way of treating denormal numbers: rather than generating them or loading them from memory, they will round them off to zero. This allows processes to run more quickly: managing all the details of denormal numbers makes programs run slower on many platforms.

Parasolid runs happily with denormals treated as zeroes, and if this mode of processing is acceptable to your application, you should consider using it. Where a platform requires special instructions for treating denormals as zeroes, this is specified in Chapter 6, “Environment”.