Parasolid V36.0

Downward Interfaces

July 2023

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

Trademarks

Table of Contents

Introduction to the Frustrum 1

1.1 Introduction

The Frustrum is a set of functions which must be written by the application programmer. They are called by Parasolid to perform the following tasks:

- Frustrum control
- File (part data) handling
- Memory handling
- Graphical output
- Foreign Geometry support (if required)

Detailed information on the Frustrum's file handling functionality is in Chapter 2, "File Handling".

The standard format of file headers which the Frustrum should read and write is described in Chapter 3, "File Header Structure".

This chapter introduces the Frustrum functions; their interfaces are defined in Appendix A, "Frustrum Functions".

Note: You are strongly advised to look at the following chapters in the Parasolid *Functional Description* before looking at the information in this manual.

- Chapter 5, "Application Design And Architecture"
- Chapter 6, "Supplying A Frustrum"

1.1.1 Dummy frustrum

The code for a dummy frustrum is supplied with the release in the file `frustrum.c`. This example gives a feel for how a Frustrum might be implemented, although the complexity of a Frustrum is dependent on its application. The example Frustrum has three purposes:

- To build and run the Parasolid installation acceptance test program.
- To let you build and run simple prototype applications without having first to write a complete Frustrum.
- To help you write you own frustrum.

Note: This frustrum contains the bare minimum required to be used, in order for it to remain clear and platform independent. Normally a Frustrum is written with a particular application in mind, and may make use of system calls rather than the C run-time library for enhanced performance.

To further help you write your own frustrum, the Parasolid Example Application also contains source code for a frustrum implementation on Windows NT. See the Chapter 6, “Supplying A Frustrum” in the Parasolid *Functional Description* for more details.

1.2 Summary of functions

This table summarizes the frustrum functions which you must provide.

Function			**Description**
FSTART			Initialise the frustrum
these functions require the Frustrum to be initialised:	FMALLO		Allocate a contiguous region of virtual memory
	FMFREE		Free a region of virtual memory (from FMALLO)
	FFOPRD		Open most guises of frustrum file for reading
	FFOPWR		Open most guises of frustrum file for writing
	UCOPRD		Open most guises of frustrum file for reading. The file has a Unicode filename.
	UCOPWR		Open most guises of frustrum file for writing. The file has a Unicode filename.
	these functions require the file to be opened:	FFREAD	Read from a file where permitted
		FFWRIT	Write to a file where permitted
		FFCLOS	Close a Frustrum file
	FABORT		Tidy up/longjump following aborted operation
	FTMKEY		Key name server required by TESTFR
FSTOP			Close down the frustrum

Note: If your application supports 16-bit Unicode filenames, then you must provide UCOPRD and UCOPWR functions in your frustrum in addition to FFOPRD and FFOPWR (which are still required for schemas and journals). If your application does not support Unicode filenames, then you must provide FFOPRD and FFOPWR functions. See Section 2.2, “Unicode filenames”, for more information.

1.2.1 Initialisation

The Frustrum is initialised by calling FSTART and is closed down by FSTOP.

The former is called by PK_SESSION_start and the latter by PK_SESSION_stop. Parasolid does not call any other Frustrum functions before the Frustrum has been initialised nor after it has been closed.

Note: The calls made by Parasolid to the start and stop functions can be nested within each other (e.g. FSTART, FSTART, FMALLO, FMFREE, FSTOP, FSTOP). Only the outermost calls are significant; the innermost calls must be ignored.

FSTART and FSTOP are assumed never to fail.

1.2.2 Memory management

Parasolid needs to be able to allocate and free virtual memory into which to put its model data. The Frustrum interface to this facility is provided by the FMALLO and FMFREE functions, and is similar to the C library functions `malloc` and `free`.

The amount of virtual memory that Parasolid requests depends on the complexity of the modelling operation. By default, the minimum is about 1/8 Mbyte. You can set and enquire the current value of this minimum block of memory using the functions PK_MEMORY_set_block_size and PK_MEMORY_ask_block_size, respectively. For complex cases, or those which require a lot of data storage, Parasolid may request more.

1.3 Abort recovery

Following an interrupt, the application has the option of calling KABORT before allowing Parasolid to continue processing. This causes it to abort the operation which was in progress. The call to KABORT would be made by an interrupt handler provided by the application.

Following such an abort, Parasolid normally returns through the PK in the normal way, giving error PK_ERROR_aborted. Before returning, Parasolid makes a call to Frustrum function FABORT, which gives the application developer an opportunity to do any generic tidying up and/or to do a long-jump to some special recovery-point within his code.

For further information on abort recovery, see Chapter 121, “Error Handling”, of the Parasolid Functional Description manual.

1.4 Frustrum errors

If a Frustrum function detects an error, it returns an error code in its final argument (`ifail`), which otherwise is the code FR_no_errors. The error codes are divided into three categories – **prediction**, **exception** and **illegal call**.

1.4.1 Prediction errors

This category contains those errors which can be expected to occur during the ordinary course of a program run. Parasolid looks for this type of error return code explicitly and takes the appropriate action.

For example, the implementation must always check the key name arguments which are passed to FFOPRD and FFOPWR and the file size argument which is passed to FFOPRB, in case Parasolid is being run with argument validation having been switched off by PK_SESSION_set_check_arguments. These types of errors can be said to be predictable.

The range of prediction error codes which can be returned from a given function are documented as end of line comments to the `ifail` argument.

1.4.2 Exception errors

This category contains codes for unpredictable but plausible errors.

Where some particular remedial action is possible, Parasolid traps these cases explicitly. If no specific course of action is appropriate, Parasolid traps and handles such cases by a default action.

For example, some PK functions need to take special action in the event of running out of disk space whereas others handle such cases by a catch all error trap.

Exception codes are also added as end of line comments to the `ifail` argument in the documentation of each Frustrum function.

1.4.3 Illegal call errors

This category is used to report an erroneous call being made to a Frustrum function, such as trying to write to a file `strid` which has not been opened or to denote that an error has occurred in the Frustrum implementation.

The Frustrum should normally report illegal call errors by outputting a message describing what went wrong, before returning the generic code FR_unspecified.

This code should only be returned when an error has been detected; it should not be set in the normal course of events. For this reason, the ifail code FR_unspecified is not used in the documentation for the Frustrum functions.

If an error occurs which is not the result of an erroneous call being made to the Frustrum or of an internal error in the Frustrum code, the ifail should be set to one the mnemonic codes in the documentation which best describes the result.

Note that the code FR_unspecified is not trapped explicitly by Parasolid, so the resulting `ifail` code returned from the KI may be misleading.

1.5 Validation tests

The test function TESTFR is supplied with Parasolid to enable the customer to check that the behaviour of his implementation of the Frustrum is consistent with the requirements of Parasolid and of file portability.

It is strongly recommended that TESTFR be linked and run every time the Frustrum is changed, and for every new version of Parasolid; Frustrum faults can cause obscure and serious problems in Parasolid.

Although the specification for TESTFR is included with the Frustrum documentation, TESTFR is not itself part of the Frustrum, and as such is not supplied by the customer, but is supplied with Parasolid.

However, the TESTFR function requires a key name server FTMKEY to be provided by the customer implementation, which returns sample names to be used as arguments in the test calls made to FFOPRD and FFOPWR. The key name server is not otherwise used by Parasolid.

1.5.1 TESTFR – invokes the verification tests for the frustrum

```
void TESTFR
(
                              /* received arguments */
int *number,       /* test number */
int *level,        /* trace level */
                              /* returned arguments */
int *code          /* completion code */
)
```

Argument	Synopsis	Values	Description
number	test number	0	run all tests
		n	run test n (n>=1)
level	trace level	0	no tracing
		1	number, purpose, result
		2	+ receive/return args
		3	+ diagnostics
		4	+ debug trace
code	completion code	0	setup_error
		1	test_success
		2	test_failure
		3	test_omitted
		4	test_exceeded
		5	test_warning

This function invokes a set of verification tests for customer implementations of the Frustrum. These are designed to confirm that the behaviour of the Frustrum (with respect to file handling and memory management) is consistent with the requirements of Parasolid and of file portability.

The tests are not foolproof, and in particular, cannot detect cases where the Frustrum writes/reads files in a way not compatible with the C run time library. Nonetheless, the tests can detect many Frustrum faults which might otherwise cause obscure and catastrophic problems in Parasolid. It is therefore strongly recommended that a customer runs the tests after any modification to his Frustrum, and after receiving any new version of Parasolid.

In order to run the tests, the customer must provide a simple driver program to call TESTFR, which he then links with TESTFR and with his Frustrum. Once a test image has been linked, the following test strategy is recommended:

- Delete any back copies of test files which have been produced by earlier runs. The names of these files have been determined by the sample key names which were returned by FTMKEY.

  For example, if FTMKEY is called with guise = FFCJNL, format = FFTEXT and test index 20, it might return the string “TESTFR_20.jnl_txt”. It would be necessary to delete any test files which matched “TESTFR_<index>.<guise>_<format>” (where <index> = 1..20)

  Note that the test files are not necessarily created in the same directory (e.g. a Frustrum implementation might choose to write its journal file to the user’s default directory but to write its schema files to a common system directory).
- Call TESTFR for test 0 at trace level 0. Note the completion code.
- If the completion code is zero (implying a setup error), check that all of the test files which are implied by FTMKEY have been deleted. Check that the test and trace arguments are being passed correctly to TESTFR (by address) and that the values are in range.
- If the completion code is one, all of the tests have succeeded.
- If the completion code is two (failure), delete any test files which have just been produced and call TESTFR for test 0 at trace level 1.

  Note the test number which has failed as N.

  Delete any test files which have just been produced and call TESTFR for test N at trace level 2 (or 3). This traces the function arguments which were used (with diagnostics).

  Correct the Frustrum implementation, making use of the trace messages (and diagnostics), then return to the first step and start the process again.
- If the completion code is three, this implies that no tests in TESTFR are associated with the given test number; it should be incremented (this is to allow particular tests to be commented out later on).
- If the completion code is four, this implies that the given number exceeds the number of tests which have been defined for TESTFR. If running with the test number as zero, this message shows up at the end of the level one trace, all the tests have been tried.
- If the completion code is five, all of the tests have succeeded but a warning has been noted; delete any test files which have just been produced and call TESTFR for test 0 at trace level 3 (writing the trace output to a log file). Search the file for the word ‘Warning’. Check with Parasolid Support if the reason for the warning is not apparent from the trace messages and Frustrum documentation.

The association between a particular check and a test number will not necessarily be maintained between releases of Parasolid and/or releases of the Frustrum interface specification.

File Handling 2

2.1 Introduction

Parts modelled in Parasolid are saved to external storage using functions in your application's Frustrum. Parasolid part files (**transmit** or **XT** files) are intended to fit into an archiving system within your application. This could take the form of a controlled directory structure on the host computer, or some kind of database.

Parasolid also requires facilities to save and retrieve large amounts of data in order to support such operations as saving and restoring snapshots and journalling.

Whilst such facilities could be implemented in a number of ways, they are described here in terms of an implementation based on the use of files provided by the host operating system.

2.1.1 Key names vs. file names

A distinction is made between the name of a key, which is passed by Parasolid as an argument to FFOPRD and FFOPWR (or UCOPRD and UCOPWR for Unicode keys) and the name of a file (which is used internally to identify the Frustrum files to the operating system).

The name of the key which is passed to the Frustrum by Parasolid is exactly the same as the name which has been given in the relevant PK call (PK_PART_transmit, PK_SESSION_transmit, PK_SESSION_start etc.), except in the case of schema files where the key has been generated by Parasolid.

The file name depends entirely on the particular implementation of the Frustrum, but typically this might include the key plus directory prefixes and file extensions as appropriate.

2.1.2 Filename extensions

The following filename extensions are recommended for the Frustrum to generate and use.

	FAT	UNIX/NTFS
part	.X_T	.xmt_txt
mesh	.M_T	.xmm_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

	FAT	UNIX/NTFS
binary	.*_B	.***_bin
debug report	*.xml	*.xml

The Frustrum should 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 can simply be renamed when transferring between the different systems.

Note that the 3 character extensions are shown in the table in upper case for clarity, though the case is ignored.

2.1.3 File guises

Parasolid requires the Frustrum to support different types (or **guises**) of file, represented by six-character integer mnemonic codes.

The following guises are opened for reading by FFOPRD and UCOPRD. The file should already exist, and its contents can be read sequentially by FFREAD.

- FFCSNP snapshot file
- FFCJNL journal file (FFOPRD only)
- FFCXMT part transmit file
- FFCXMM meshes transmit file
- FFCSCH schema file (FFOPRD only)
- FFCLNC licence file
- FFCXMP partition transmit file
- FFCXMD deltas transmit file

The following guises are opened for writing by FFOPWR and UCOPWR. A new file is created, and it can be written to sequentially by FFWRIT:

- FFCSNP snapshot file
- FFCJNL journal file (FFOPWR only)
- FFCXMT part transmit file
- FFCXMM meshes transmit file
- FFCSCH schema file (FFOPWR only)
- FFCXMP partition transmit file
- FFCXMD deltas transmit file
- FFCDBG debug report file (FFOPWR only)

When all of a new file has been written or sufficient of an existing file has been read, the file is closed with FFCLOS. In the case of new files, the call specifies whether or not the new file is to be retained.

2.1.4 File header

There is a standard format for a Frustrum file header, which is described in Chapter 3, “File Header Structure”.

2.1.5 Number of files open concurrently

In normal operation, Parasolid only has a journal file open, which is open for writing.

For short periods, Parasolid may need to have up to two more files open. These are either a snapshot file or a transmit file, possibly with its associated schema file.

The Frustrum implementation must therefore allow for three files to be open at the same time for each Parasolid process.

2.2 Unicode filenames

Parasolid supports the use of 16-bit Unicode filenames in your application, as well as the native character set. This is achieved by using the following frustrum functions instead of FFOPRD and FFOPWR:

Function	Description
UCOPRD	Open various guises of part file for reading. The part file has a filename encoded in Unicode format.
UCOPWR	Open various guises of part file for writing. The part file has a filename encoded in Unicode format.

To use these functions, you must call PK_SESSION_set_unicode before registering the frustrum. Once called, you must then supply valid UCOPRD and UCOPWR functions instead of FFOPRD and FFOPWR functions when registering the frustrum. For more information about registering the frustrum, see Chapter 5, “Registering the Frustrum”.

To find out whether Unicode keys are enabled within a given session, call PK_SESSION_ask_unicode.

2.3 File formats

Parasolid regards a file as being simply a stream of bytes, which is written or read sequentially. It is assumed that the stream of bytes obtained when reading is identical to that which was written (except in the case of a file ported between systems, when the character set may change).

2.3.1 Text and binary

Parasolid distinguishes between **binary** files and **text** files:

- In a binary (FFBNRY) file, the stream of bytes has no inherent meaning or structure. It can contain bytes of any value.
- For a text (FFTEXT) file, the stream of bytes consists of printing characters (as defined by the C library function `isprint`) interspersed with newline characters (corresponding to `\n` in C) such that there are never more than 80 consecutive printing characters.

When writing certain files via the Frustrum, the application needs to specify whether they are to be in text or binary format. A file must be written and read in the same format (thus is Frustrum dependent). When deciding which format to use, consider the following factors:

- Machine specific binary files can only be read back on the same type of machine as that on which they were written.
- Text files (subject to conversion of the contents of the file to the local character set of the target machine) and neutral binary files should be portable between machines.
- Binary, neutral or machine files are normally quicker both to generate and read back than text files, and take up less space. The format of files influences the amount of space required and so affects how space is allocated when the Frustrum is written.
- Whether binary transmit files or neutral transmit files are created is dependent on the option switch passed to PK_PART_transmit and PK_PARTITION_transmit.
- Text files that comply with Parasolid's standard are machine independent and Frustrum independent (the header of the file is in human readable form and may be of interest, but the rest of the file is not intended to be human readable).

Note: 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 Example Frustrum.

2.3.2 Application I/O

There is also a transmit file format called **application i/o**, or **applio**. When this `transmit_format` is selected in PK_PART_transmit and PK_PARTITION_transmit, transmit files are written and read using a suite of functions provided by the application. Using these functions enables the application to do further processing of the output data before storing it.

The applio function interfaces are defined in Appendix E, “Application I/O Functions”.

2.3.3 Indexed I/O

There is also a transmit file format called **indexed i/o**, or **indexio**. When this `transmit_format` is selected in PK_PART_transmit and PK_PARTITION_transmit, transmit files are written using a suite of functions provided by the application. Using these functions enables your application to subsequently receive only specified faces from the parts contained in the transmitted files.

The indexio function interfaces are defined in Appendix F, “Indexed I/O Functions”.

2.3.4 Portability

It is clearly desirable that Parasolid files be portable between different machines and between different systems which use Parasolid (which have different Frustrum implementations). In practice, machine specific binary files cannot be ported from one type of machine to another, so the best that can be achieved is:

- All files shall be portable between different Frustrum implementations on the same machine type.
- Text and neutral binary files shall be portable between different Frustrum implementations on different machine types.

In order that files be portable between different Frustrum implementations, it is necessary to standardize the internal format of the file. For this reason, we recommend that all Frustrum implementations should generate files in the same format as is generated by the appropriate C runtime library functions on the appropriate machine. In the case of text files, this includes correct handling of newline (`\n`) characters.

To ensure portability of text files between different machine types, we require that:

- With the exception of newline characters, text files should only contain printing characters, as defined by the C library function `isprint`. All text data output from Parasolid conforms to this rule; however, the Frustrum must guarantee it for the Frustrum header data.
- Text files should not contain lines of more than 80 characters. To ensure this is the case, Parasolid automatically inserts newline characters (`\n`) into all text data being output. The Frustrum must ensure sufficient newline characters appear in the Frustrum header.

It is also necessary that, when a text file is ported to a different type of machine, it is converted to the local character set and C format for the the target machine.

If a Frustrum implementation does not follow the above recommendations it may prove impossible to read files which have been created by other systems or to forward its own files to other systems (such as when making a fault reporting). Problems of this sort may be revealed by running the Frustrum validation tests, but note that these tests cannot verify that the file format is consistent with the C runtime library, so some additional check is required to confirm that this is the case.

2.4 Characteristics of different file guises

2.4.1 FFCSNP

Snapshot files are created by PK_SESSION_transmit. The data within the snapshot files is schema dependent and Parasolid needs to have access to the corresponding schema file in order to interpret it.

2.4.2 FFCJNL

Journal files are created as a result of calling PK_SESSION_start with journalling switched on. They are always created in text format. The files are used to record the values of arguments which have been passed to and received from PK and KI functions.

Journal files are *reasonably* portable between machines, except where the arguments specify such machine dependent features as file names or database keys or where they refer to parts which were created in an earlier session.

2.4.3 FFCXMT

Transmit files containing parts, created by PK_PART_transmit. The data within the transmit files is schema dependent and Parasolid needs to have access to the corresponding schema file in order to interpret it.

2.4.4 FFCSCH

Schema files are created by Parasolid to have names of the form sch_*n* (where *n* represents an integer value).

The integer value is used internally by Parasolid to identify any changes which have been made to the Parasolid schema.

The schema describes the internal data structure which is used to represent part data within the Parasolid model, such as the ordering of geometric data and the relationships between edges and faces.

The Frustrum should store schema files in a separate directory. The KID Frustrum stores them in a directory which is referenced by the P_SCHEMA environment variable.

Note that the Parasolid release includes a set of schema files for all previous versions of Parasolid and for the current version of the Parasolid modeller. This ensures upgrade compatibility of old part files.

Application writers should include a full set of schema files with their product release, including one for the current version of the Parasolid modeller.

In operating systems which have case specific file names the KID Frustrum chooses to write and read the schema archive file in lower case. Any old schema files that are supplied with a release have lower case file names.

2.4.5 FFCLNC

Licence files may be used in subsequent versions to check that Parasolid is being used in accordance with the licencing agreement. The validation tests for Frustrum implementations require this guise of file to be supported by FFOPRD & FFOPWR, so that the licence checking capability can be introduced in a later release of Parasolid, without the need to alter the Frustrum interface further.

It is intended that licence files are created in text format; consisting of a standard file header followed by one or more lines of licence checking data.

2.4.6 FFCXMP

Transmit files containing a partition, created by PK_PARTITION_transmit and read by PK_PARTITION_receive.

2.4.7 FFCXMD

Transmit files containing deltas, created by PK_PARTITION_transmit if the option to transmit deltas is selected. During the transmit, the partition's deltas are opened, read and output to the transmit file (using the delta handling functions registered with PK_DELTA_register_callbacks).

Delta files can only be received in the version they were transmitted in. They are read by PK_PARTITION_receive_deltas_2, if the option to receive deltas later was selected when the relevant partition was read in by PK_PARTITION_receive.

2.4.8 FFCDBG

Debug report files are created as a result of calling PK_DEBUG_report_start. They are always created in XML format. These files are used to record information such as the values of arguments passed to and returned by Parasolid functions, as well as embedding relevant part files.

2.4.9 FFCXMM

Transmit files containing Persistent Mesh (PSM) mesh data.

2.5 Open modes

There are the basic modes in which files are used by Parasolid.

2.5.1 open_read

This mode is characteristic of receiving a Parasolid transmit file, receiving a snapshot file or reading an archived schema file.

A test is made first for whether there is any header data and whether the ‘skip header’ flag is set. If so, the header data is read and checked as deemed necessary by the implementation. If there is no header data or if the header data is to be left (for checking by the Frustrum validation tests), the file pointer is repositioned back to the start of the file.

The file is read sequentially until sufficient data has been read or until the end of file is reached. The file is then closed (and it is retained).

2.5.2 open_new

This mode is characteristic of recording a new journal or debug report file.

The file is opened for writing and the header data are written to it. Parasolid then writes sequentially to the file, journalling the arguments to each interface function. The new file is closed and is retained.

If the system crashes while the journal or debug report file is still open, the system is left with a (possibly incomplete) file. Even in its incomplete form, this file can be useful for debugging a session.

2.5.3 open_protected

This mode is used for creating new transmit, schema and snapshot files.

The difference between the open_protected and the open_new modes is that the Frustrum helps to prevent Parasolid from accessing incomplete or otherwise erroneous files. If an error is detected by Parasolid when writing to a new file, the call to FFCLOS is made with the status code FFABOR, meaning that the file must be deleted when the strid is closed down.

However, if new files are not closed explicitly (as could occur after a system crash), it is possible that a newly created file is not complete.

The Frustrum can protect Parasolid from incomplete files by creating new transmit, schema and snapshot files with scratch names, only giving them their correct identity after the files have been closed explicitly by a call to FFCLOS.

2.5.4 Summary of open modes

The three basic methods of using files are summarized in the following table:

		open_read	open_new	open_protected
open mode	**FFOPRD**	**Y**		
	FFOPWR		**Y**	**Y**
	UCOPRD	**Y**		
	UCOPWR			**Y**
file guise	**FFCSCH**	**Y**		**Y**
	FFCXMT	**Y**		**Y**
	FFCSNP	**Y**		**Y**
	FFCJNL		**Y**	
	FFCLNC	**Y**		
	FFCDBG		**Y**	
operation	**FFREAD**	**Y**		
	FFWRIT		**Y**	**Y**
close mode	**FFCLOS with action normal**	**Y**	**Y**	**Y**
	FFCLOS with action abort			**Y**

An entry **Y** denotes that Parasolid calls the given function in the way which is described by the comment above the column. The file guises XMP and XMD (transmit files containing partition and delta data) use the same methods as XMT (part transmit) files.

2.6 Explanation of the special characters in a journal file

Journal files contain the following markup characters to assist with interpretation of the contents.

2.6.1 Record and element symbols

Symbol	Description
:	The first and last lines are comment records, which start with a ':' character.
<	The '<' character at the beginning of each journal record is followed by the name of a PK function. This is followed by a sequence of lexical items, being tags, text strings, integers, doubles and punctuation symbols. These are separated by one or more spaces. ■ when journal records are written over more than one line, the continuation lines start with a space ■ nested PK calls are shown with one angle bracket for each nesting depth; if there is an error, and PK_SESSION_tidy is called, the level is reset back to 0
#	A tag is represented by a '#' character, followed by digits.
" "	A string is enclosed in quotation marks (two successive quotation marks imply that the string itself contains a quotation mark).
(-)nnn	Integer values are represented by an (optional) sign, followed by digits.
(-)nn.nnn **5.6e07** **-5.6e-07**	Double values are represented by an (optional) sign, followed by a floating point representation (with a decimal point) or an exponent and mantissa representation (with an 'e' character).
@	Function pointers are journalled as addresses.

2.6.2 Punctuation symbols

Symbol	Description
;	the semi-colon separates received arguments from returned arguments
&	the ampersand concatenates adjacent text strings (the journalling system sometimes needs to split a long text string when writing it to file)
[ ]	square brackets enclose an array of some type (all elements of the array have the same type); if an array or pointer argument is supplied as NULL, this is journalled as the address value @0
( )	round brackets enclose a list of doubles (e.g. vectors)
{ }	curly brackets enclose a list of structure members, e.g. a standard form

File Header Structure 3

3.1 Introduction

There is a standard format for a Frustrum file header, designed to give the following benefits:

- To allow a customer to add his own information to the file without rendering it incompatible with other Parasolid-based systems.
- To provide a standard method for recording in the file such information as when, where, by whom the file was created.

All files must begin with a file header. The header is written and read by the customer’s implementation of the Frustrum, *not* by Parasolid. The reason for standardizing the format of the header is to ensure compatibility between different Frustrum implementations; Parasolid itself never sees the header and therefore has no knowledge of its format.

It is vital that every Frustrum implementation produces file headers which conform to the standard. Therefore, the Frustrum validation tests include specific checks that this is the case.

3.2 Structure of file header

The file header consists of a preamble, three parts of keyword data and a trailer.

- The purpose of the preamble is to identify whether or not a Frustrum file has a header, and also serves to define the character set which is used for writing keyword data.
- The **part 1** keyword definitions describe the file characteristics and the environment in which it was created (e.g. the guise and date of creation).
- The **part 2** keyword definitions provide information about the version of Parasolid which was being used when the file was created.
- The **part 3** keyword definitions provide a method by which the Frustrum developer can attach user specific data to the file.
- The file header is terminated by a trailer record. This provides a check as to whether the three parts of user data have been formatted correctly.

3.2.1 Format of the preamble

The preamble is written as two lines of 80 characters, each terminated by an end of line character. It includes all characters in the ASCII printing set.

```
**ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz**************************
**PARASOLID !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~0123456789**************************
```

The punctuation characters in the preamble are written in the same order as they appear in the ASCII sequence (excluding alphanumeric characters).

3.2.2 Format of part data

Each of the three parts of keyword data starts with the declaration “**PART1;”, “**PART2;” or “**PART3;”. (The quotation marks used here are for documentation purposes and do not appear in the file.)

Each part declaration is followed by a sequence of keyword definitions, each consisting of a keyword name and a keyword value.

The keyword definition sequence for each part is terminated by the start of the declaration for the next part or by the start of the trailer record.

Keyword definitions are written to file so that no output line is more than 80 characters long. The new line characters which are required to achieve this layout have no effect on the meaning of the keyword names or values and can appear anywhere within a keyword definition as it is written to file.

A new line character is written at the end of each keyword definition sequence so that the next part declaration or trailer record starts on a new line.

3.2.2.1 Part 1 data

The part 1 data is standard information which should be accessible to the Frustrum (e.g. by operating system calls). It is the responsibility of the Frustrum to gather the relevant information and to format it as described in this specification. A list of keywords and their meanings is given in a later section.

3.2.2.2 Part 2 data

The part 2 data is again standard information, but this time is information not readily available to the Frustrum (e.g. the Parasolid schema version), and which therefore must be provided from Parasolid. When creating a new file, Parasolid passes a string containing the relevant keywords/values to FFOPWR or UCOPWR, as appropriate. The frustrum must then insert this string into the header in the appropriate place.

The string passed to FFOPWR or UCOPWR does NOT include newline characters or the “**PART2;” prefix; these must be added by the Frustrum. However, the Frustrum should not add escape characters to the string; these have been added by Parasolid, if required.

As an example, the string passed to FFOPWR for the following sample file header would be “SCH=SCH_700084_7007;USFLD_SIZE=0;”.

3.2.2.3 Part 3 data

The part 3 data is non-standard information, which is only comprehensible to the Frustrum which wrote it. However, other Frustrum implementations must be able to parse it (in order to reach the end of the header), and it should therefore conform to the same keyword/ value syntax as for part 1 and part 2 data. However, the choice and interpretation of keywords for the part 3 data is entirely at the discretion of the Frustrum which is writing the header.

3.3 Example of simple file header

```
**ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz**************************
**PARASOLID !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~0123456789**************************
**PART1;MC=hppa;MC_MODEL=9000/710;MC_ID=sdlhpp36.2006892913;OS=HP-
UX;OS_RELEASE=A.09.05;FRU=sdl_parasolid_test_hppa;APPL=unknown;SITE=sdl-
cambridge-
u.k.;USER=ianb;FORMAT=text;GUISE=transmit;KEY=cube;FILE=cube.xmt_txt;DATE
=2-apr-1996; **PART2;SCH=SCH_70084_7007;USFLD_SIZE=0;

**PART3;

**END_OF_HEADER*****************************************************************

T50 : TRANSMIT FILE created by modeller version 70008415 SCH_700084_70070
12 1 6 2 0 0 0 0 0 2 1e3 1e-8 0 3 0 2 1 4 5 6 7 8 9 10 102 4 2 cube13 4 3
0 ... etc ...
```

3.4 Syntax of keyword definitions

All keyword definitions which appear in the three parts of data are written in the form <*name*>=<*value*> e.g. `MC=hppa;MC_MODEL=9000/710`;

where

- <*name*> consists of 1 to 80 uppercase, digit, or underscore characters
- <*value*> consists of 1 or more ASCII printing characters (except space)

3.4.1 Escape sequences

Escape sequences provide a way of being able to use the full (7 bit) set of ASCII printing characters and the new line character within keyword values.

- The header specification does not allow certain characters to be written to file directly; instead, they must be converted to escaped form as they are written to file.

The implementation must also be able to recognise and to convert escaped characters as they are read back from file.

3.4.1.1 New line

The requirement to format the output data into lines of 80 characters or less means that new line characters are ignored as keyword definitions are read back from file (although they are still significant when they are read as part of the preamble or the trailer record).

If new line characters need to be included within a keyword value, they must be written to file in escaped form as “^n” (up_arrow followed by lower case n).

Care is required when reading keyword values from file so that new lines which are part of the keyword value are not confused with file layout new lines.

3.4.1.2 Space

The specification does not allow spaces to be written to file as part of the keyword data. This is because of the danger of losing trailing spaces when porting text files between different systems.

If space characters need to be included within a keyword value, they must be written to file in escaped form as “^_” (up_arrow followed by underscore).

3.4.1.3 Semicolon

The specification uses the semicolon character to mark the end of a keyword value. If semicolon characters need to be included within a keyword value, they must be written to file in escaped form as “^;” (up_arrow followed by semicolon).

Care is required when reading these characters back from file so that the semicolons within a keyword value are not confused with the semicolons which terminate a keyword value.

3.4.1.4 Up arrow

The specification uses the up arrow character as its escape character when writing keyword values to file. When used in a keyword value, the up arrow character is doubled up when it is written to file so as to avoid ambiguity when reading back the data.

3.4.1.5 General points

The two character escape sequences may be split by a new line character as they are written to file. They must not cause any output lines to be longer than 80 characters.

Only those characters which belong to the ASCII (7 bit) printing sequence, plus the new line character, can be included as part of a keyword value.

It is possible that the space, semicolon and up arrow characters are used within keyword values which follow the KEY and FILE keywords in part 1 data; the implementation must be able to decode these, even if it does not need to encode escaped characters when writing its own file headers.

The new line character does not appear as part of a file or key name (in normal operation) as this is rejected by the PK argument checking phase.

It is possible for any of the escaped sequences to be used within the keyword values which are associated with part 3 data.

Note that the preamble and the trailer record are written to file in literal mode, without using escape character sequences.

3.5 Pre-defined keywords

The following keyword names must be present in each file header, in the correct section of part data. The keyword name must be set to one of the associated values which is shown on the right hand side below (such as hppa) or must use the formatting conventions which are given (such as the date consisting of a one or two digit day number, a three letter abbreviation for the month in lower case and a four digit year number).

The pre-defined sets of keyword values are written in lower case rather than in upper case; this is significant.

The spacing and commas which are shown with the lists of pre-defined keywords are for documentation purposes only and must not be used in keyword values.

The sequence “...” is used to represent an arbitrary sequence of one or more characters (for example, the value for the keyword FILE can be cube.xmt_txt). However, all characters which are used in a keyword value must be converted to the escaped form where necessary.

If the Frustrum developer cannot determine which keyword value applies to a particular keyword, in certain cases this can be set as “unknown”. In all other cases, the value should be set to one of the pre-defined values or to use the specified format.

If the range of keyword values which is shown in the Frustrum documentation is not sufficient (e.g. Parasolid is ported to a new machine), a request should be made to Parasolid Support to have the list extended.

3.5.1 Part one data

Keyword	Description	Notes
MC	make of computer e.g. hppa	can be set as “unknown”
MC_MODEL	model of computer e.g. 9000/710	can be set as “unknown”
MC_ID	unique machine identifier ...	can be set as “unknown”
OS	name of operating system e.g. HP-UX	
OS_RELEASE	version of operating system e.g.A.09.05	can be set as “unknown”
FRU	Frustrum supplier and implementation name e.g. sdl_parasolid_customer_support	can be set as “unknown”
APPL	application which is using Parasolid e.g. parasolid_acceptance_tests	can be set as “unknown”
SITE	site at which application is running ...	can be set as “unknown”
USER	login name of user ...	can be set as “unknown”
FORMAT	format of file binary, text	MUST BE SET
GUISE	guise of file snapshot, transmit, schema, journal, licence	MUST BE SET
DATE	dd-mmm-yyyy e.g. 2-apr-1996	can be set as “unknown”

3.5.2 Part two data

Keyword	Description	Notes
SCH	SCH_m_n name of schema key e.g. SCH_700084_7007	MUST BE SET
USFLD_SIZE	length of user fields (0 – 16 integer words) m	MUST BE SET

3.5.3 Part three data

There are no restrictions on the choice of keyword names and values which can be used in the part three data, other than the general rules which have been stated earlier.

Graphical Output 4

4.1 Introduction

When a call is made to the PK rendering functions, the graphical data generated is output through a set of functions known as the GO (Graphical Output) Interface

Note: You should not call heavyweight functions from the GO. See Section 122.2.1, “State of the code in execution”, of the Parasolid *Functional Description* for more information on heavyweight functions.

See the Function Exclusivity list in the *PK Interface Programming Reference Manual* to find out if a function is heavyweight or lightweight.

Warning: This chapter describes the GO interface which you must provide to receive graphical data from Parasolid. The GO Interface consists of a number of functions that must meet the specifications in Appendix B, “Graphical Output Functions”. There are dummy versions of these functions in the Frustrum library which is supplied with Parasolid, so that you can link your programs. They do not do anything, however, so if you want to use any rendering functions you must write your own GO functions, or no graphical data is forthcoming.

4.2 Graphical output functions

Line Data is produced by PK_GEOM_render, PK_TOPOL_render_line and PK_TOPOL_render_facet. It is output through the GO functions GOOPSG, GOSGMT and GOCLSG.

There are two values which GO functions should return in the ifail argument, both defined as tokens in the Parasolid failure codes include file. All GO functions should return one of:

- CONTIN, if the graphical operation is to continue
- ABORT, if there is an error gross enough that the rendering operation should end. In this case the rendering functions return with the error code PK_ERROR_abort_from_go.

4.3 Structure of line data output

Data output through the GO is organised into **segments**, which correspond to identifiable portions of the model (not necessarily entities in the Kernel sense).

4.3.1 Segment hierarchy

There are two classes of segment:

- Hierarchical segments are opened by a call to GOOPSG and remain open until explicitly closed by GOCLSG. Other hierarchical or single-level segments may be produced in between, and these can be regarded as contained in the hierarchical segment. They may contain any number of other segments.
- Single-level segments are opened and closed by a single call to GOSGMT. They do not contain any other segments. The kernel never creates single-level segments on their own, they are always included in a hierarchical segment.

Every segment has a **type,** which governs (among other things) whether it is a hierarchical segment or not.

Single level segments always contain the data for a line to be drawn, as well as information about what kind of line it is.

Hierarchical segments usually contain other segments. Note that graphical data is always output hierarchically: using the `hierarch` option in PK_TOPOL_render_line affects the level of hierarchy.

- When faceting, both the body segment SGTPBY and the face segment SGTPFA are hierarchical, therefore there can be three levels of segment at one time: a body; a face; and another segment.
- When faceting volumetrically, the body segment SGTPBY, the region segment SGTPRE and a shell segment SGTPSH or a frame segment SGTPFR are hierarchical, therefore there can be four levels of segment at one time: a body; a region; a shell or frame and another segment.
- When creating hidden line graphical data the body segment SGTPBY is hierarchical. If the `hierarch` option is used then edge segment SGTPED; silhouette segment SGTPSI; and hatch-lines SGTPPH (planar) SGTPRH (radial) SGTPPL (parametric) are hierarchical.
- For all other options the only type of segment which is hierarchical is the body segment SGTPBY. Therefore as bodies cannot be contained in bodies, there can be at most two levels of segment at one time: a body and another segment.

For example, after a request by the application for view-dependent topology information by a call to PK_TOPOL_render_line, a hierarchical segment is opened for each body. Data for each silhouette line is then given in a single-level segment, and each is contained wholly within the hierarchical segment of the corresponding body.

A single body segment is not guaranteed to contain data for the whole body – for example, sometimes the kernel can output part of a body, close the body segment, open and close another body segment, and then open a new segment for the first body, and output the rest of it.

4.3.2 Graphical data for assemblies

Assemblies constructed with KI routines must be flattened and their constituent body tags and transformation matrices copied into entity arrays before they can be rendered by the PK functions. For further information see Chapter 1, “Parasolid KI Programming Concepts”, of the Parasolid *KI Programming Reference Manual*.

All such body segments have a unique occurrence number when they are rendered – this number is the index of the body (in the entity array) plus 1.

The tags associated with each of these body segments are likely to be different. There are two circumstances when you could receive separate body segments with the same tag:

- a body is instanced more than once in the entity array, and each instance is output in a separate segment (or segments). The occurrence number is different in each case.
- a single body is being output piecemeal, e.g. if the body has been split into two sections in a hidden line drawing because part of it is obscured by another body, as for the block shown in Figure 4–1:

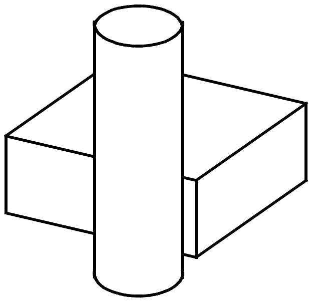

Figure 4–1 *Hidden line drawing of a single body*

In this case, the lines comprising each visible portion of the body might be output in separate body segments.

4.3.3 Notes

You should note that geometrical data output through the segment output functions is:

- in *three dimensions*; it is up to you to project it into two dimensions if required
- relative to *model space* (i.e. the same coordinate system as would be returned by the PK geometry enquiry functions)

The application must transform all of the received GO data by a viewing matrix before projecting the data into two dimensions:

- if the GO data includes view-dependent or hidden line data, this matrix must be the same as the one referenced by the `view_transf` argument when calling PK_TOPOL_render_line
- if the GO data includes no view-dependent or hidden line data, the viewing matrix can be defined wholly by the application

If the viewing matrix specifies a parallel orthographic projection, the 3D GO data can be projected into 2D form simply by replacing the third column of the matrix by a zero vector:

- i.e. using the same notation as used in Chapter 111, “Parasolid View Matrices”, of the Parasolid *Functional Description*, the combined matrix is formed by setting D_x D_y D_z T_z to zero.

4.4 Segment output functions

See Appendix B, “Graphical Output Functions”, for detailed descriptions of each function. The three functions (GOOPSG, GOSGMT, GOCLSG) all have the same arguments, but interpret them in different ways. The arguments are:

```
GO....( segtyp, ntags, tags, ngeom, geom, nlntp, lntp, ifail )
```

The first argument of each function is an integer `segtyp`. Parasolid always sets this to the token representing the **segment type.** These tokens are listed in Appendix I, “Go Tokens and Error Codes”, of this manual. The different types of segment are discussed in a later section.

The values of some of the arguments to the segment functions, and therefore how you should interpret them, vary depending on the segment type. The arguments to which this applies are noted as such, below. The remaining arguments are pairs of integers and arrays.

4.4.1 Tags

The integer `ntags` gives the length of the `tags` array. The contents of this array depend on the segment type. For example, for a hierarchical body segment, `tags` contains the tag of the body; for a single level edge segment, `tags` contains the tag of the edge.

4.4.2 Line type

The integer `nlntp` gives the length of the `lntp` array. This is an array of integers, the first of which is always the **occurrence number** of the segment, and the rest of which are tokens. For hierarchical segments (i.e. in calls to GOOPSG and GOCLSG) the `lntp` array contains only the occurrence number of the segment. The exception is for frame segments as these will also contain the sense of the frame.

Occurrence numbers link the segment to the entity which was passed to the rendering function. You can use them to associate the segments with the Parasolid entities (perhaps using identifiers to identify them). You can then identify what a particular line represents, and the entity it belongs to.

Note: You should use identifiers rather than tags to identify entities.

Identifiers are saved with the part when you archive it and are therefore always the same, whereas the tags of a part and its entities can be different in every Parasolid session.

See also Section 105.3.2, “Occurrence numbers” in the Parasolid *Functional Description*.

Silhouettes are produced by the `silhouette` option in PK_TOPOL_render_line. These automatically have each silhouette on a face labelled with a different integer, i.e. 1, 2, 3, etc.

The remaining array entries, which are given as well as the occurrence number for all single-level segment types, are as follows.

4.4.2.1 Line type

Line type specifies the type of geometry of the curve or lattice element which the segment represents. This is one of:

- Straight line
- Complete circle
- Partial circle
- Complete ellipse
- Partial ellipse
- Poly-line
- Facet vertices
- Facet strip vertices
- Facet vertices + surface normals
- Facet strip vertices + surface normals
- Facet vertices + parameters
- Facet strip vertices + parameters
- Facet vertices + normals + parameters
- Facet strip vertices + normals + parameters
- Non-rational B-curves (Bezier form)
- Rational B-curves (Bezier form)
- Non-rational B-curves (NURBs form)
- Rational B-curves (NURBs form)
- Facet vertices + normals + parameters + 1st derivatives
- Facet strip vertices + normals + parameters + 1st derivatives
- Facet vertices + normals + parameters + all derivatives
- Facet strip vertices + normals + parameters + all derivatives
- Complete sphere
- Complete cylinder
- Complete truncated cone

The different types of line are discussed in detail under Section 4.4.3, “Geometry”, below. They are defined by tokens of the form “L3TP...”

4.4.2.2 Completeness

Completeness codes indicate whether a segment represents a complete item or is part of a larger item. The codes returned may be any of the following:

- CODCOM: the segment is complete and represents a complete item.
- CODINC: the segment is incomplete. An incomplete segment is part of a larger item which might in other circumstances have been output as a single segment.
- CODUNC: it is not known whether the segment is complete or incomplete.
- CODOVP: if viewport clipping is on, the segment is outside of the viewport. Your application may choose not to render such a segment. No visibility code is specified if this completeness code is returned.
- CODCVP: if viewport clipping is on, the segment coincides with a viewport boundary. This information can be useful in helping your application decide how to render segments that lie on a viewport boundary. If, for instance, you are rendering a large body in a number steps using several viewports, this code can help your application ensure that segments on the boundaries are only rendered once.

For more information about viewports, see Section 106.3.23, “Using viewports to render specific entities” in the Parasolid *Functional Description*.

Completeness codes are only calculated in a hidden line drawing, so segments output from a view independent or view dependent wireframe drawing has code CODUNC (unknown completeness).

4.4.2.3 Visibility

Visibility specifies that the line is visible, invisible or of unknown visibility. This value is only relevant to hidden line pictures, so all segments produced in view independent or view dependent wireframe drawings have unknown visibility (CODUNV).

In hidden line drawing:

- Visible segments have visibility code CODVIS.
- Invisible segments have visibility code CODINV. Invisible segments are output only if selected specifically by the `visibility` field of the PK_TOPOL_render_line option structure.
- Drafting segments have visibility code CODDRV.
- Invisible segments which are obscured only by their own body occurrence have visibility code CODISH (*see Note below*). This type of invisible segment is output only if selected specifically by the ‘extended visibility’ and ‘self-hidden’ fields of the PK_TOPOL_render_line option structure.

The effects of the PK_render_vis_... options are as follows:

Option	Description
vis_no_c	no visibility evaluated (topology is output as a view-dependent wireframe drawing)
vis_hid_c	only truly visible lines are output, all tagged CODVIS
vis_inv_c	all lines are output: ■ the truly visible ones tagged CODVIS; ■ the remainder tagged CODINV
vis_inv_draft_c	all lines are output: ■ the truly visible ones tagged CODVIS ■ those obscured by others tagged CODINV ■ the remainder being lines obscured by the body tagged CODDRV
vis_extended_c	lines are output subject to the ‘invisible’, ‘drafting’ and ‘self-hidden’ fields of the PK_TOPOL_render_line option structure

4.4.2.4 Smoothness

Smoothness indicates whether a line is smooth (i.e. the normals of the faces either side of the edge vary smoothly across it). Blend boundaries are smooth by definition. This code is provided because sometimes you may wish to leave smooth edges out of your pictures, to make them look more realistic.

For further information see Section 106.3.24, “Regional data” in the Parasolid *Functional Description*.

CODSMS is a special code which can be returned only from a hidden line drawing. It indicates that an edge is smooth, but that it is also coincident with a silhouette line which is not output. In this case you need to draw the edge even though it is smooth, because otherwise the silhouette is missing, making the picture look wrong.

CODNSS is a special code which can be returned only from a hidden line drawing when rendering sharp mfins. It indicates that the mfin line (which is sharp) is coincident with a silhouette line which is not output.

4.4.2.5 Regional data

If regional data was requested from a hidden line drawing, edge and silhouette segments are output with start and end **point indices**. These are output only when the regional data option is specified, and allow the line segment to be linked correctly with other lines. See Section 4.5, “Interpreting regional data”, for more information.

4.4.2.6 Coincidence with sharp mfins

If sharp mfins were requested then edge segments are output with a code describing whether the segment lies on a chain of connected sharp mfins as follows:

- CODESF: the edge segment is coincident with sharp mfins
- CODNES: the edge segment is not coincident with sharp mfins
- CODUES: the edge segment coincidence with sharp mfins is unknown

These codes are only output for edge segments when the `sharp_mfins` option in PK_TOPOL_render_line is set to PK_render_sharp_mfin_yes.

See Section 4.4.4.24, “Sharp mfin line: SGTPSF” for more information.

4.4.3 Geometry

For hierarchical body segments the `geom` array always contains the model space box of the body, and `ngeom` is 6. See Section 4.4.4, “Segment types” for more details.

For a single level segment the `geom` array is an array of real numbers specifying the geometry of the line it represents. The length and contents of this array depend on the line type, as specified by the second entry of `lntp` (see Section 4.4.2, “Line type”). The length of the array is `ngeom` unless otherwise stated. This concept of a *line* does not correspond exactly to any type of entity at the PK Interface: it is either a set of data describing an analytic curve with a start-point and an end-point, or a poly-line.

The types of line which can be returned are:

4.4.3.1 Straight line: L3TPSL

ngeom = 9	
geom[0...2]	start point
geom[3...5]	end point
geom[6...8]	direction

The explicit direction is generally more accurate than that obtained from the start and end points.

4.4.3.2 Complete circle: L3TPCC

ngeom = 7	
geom[0...2]	center point
geom[3...5]	axis direction
geom[6]	radius

4.4.3.3 Partial circle: L3TPCI

ngeom = 13	
geom[0...2]	center point
geom[3...5]	axis direction
geom[6]	radius
geom[7...9]	start point
geom[10...12]	end point

4.4.3.4 Complete ellipse: L3TPCE

ngeom = 11	
geom[0...2]	center point
geom[3...5]	major axis direction
geom[6...8]	minor axis direction
geom[9]	major radius
geom[10]	minor radius

4.4.3.5 Partial ellipse: L3TPEL

ngeom = 17	
geom[0...2]	center point
geom[3...5]	major axis direction
geom[6...8]	minor axis direction
geom[9]	major radius
geom[10]	minor radius
geom[11...13]	start point
geom[14...16]	end point

4.4.3.6 Poly-line: L3TPPY

ngeom = the number of 3D vectors	
geom[0...2]	start point
geom[3...5]	second point
geom[i...i+2]	nth point, where i = 3(n-1)

Note that the double type array holding a poly-line is of length 3*ngeom.

The poly-line is a chordal approximation to a line which can not be held explicitly within the Kernel. It defines a series of points, each of which lies on the corresponding Parasolid curve. If you join the points of a poly line with straight line segments, this produces an approximation to the curve which is adequate for most viewing purposes. Splining the points produces a more accurate approximation if one is required.

4.4.3.7 For facet vertices

L3TPFV and facet strip vertices – L3TPTS

ngeom = the number of facet vertices	
geom[0...2]	first facet vertex
geom[3...5]	second facet vertex
geom[i...i+2]	last facet vertex, where i = 3(ngeom-1)

4.4.3.8 For facet vertices plus surface normals

L3TPFN; and facet strip vertices plus surface normals – L3TPTN

ngeom = 2 times the number of facet vertices	
geom[0...2]	first facet vertex
geom[3...5]	second facet vertex
geom[i...i+2]	last facet vertex, where i = 3((ngeom/2)-1)
geom[i+3...i+5]	first facet vertex normal
geom[i+6...i+8]	second facet vertex normal
geom[k...k+2]	last facet vertex normal, where k = 3(ngeom-1)

4.4.3.9 For facet vertices plus parameters

L3TPFP; and facet strip vertices plus parameters – L3TPTP

ngeom = 2 times the number of facet vertices	
geom[0...2]	first facet vertex
geom[i...i+2]	last facet vertex, where i = 3 (ngeom/2-1)
geom[i+3...i+5]	first facet vertex (u,v,t) information
geom[k...k+2]	last facet vertex (u,v,t), where k = 3(ngeom-1)

4.4.3.10 For facet vertices plus normals plus parameters

L3TPFI; and facet strip vertices plus normal plus parameters – L3TPTI

ngeom = 3 times the number of facet vertices	
geom[0...2]	first facet vertex
geom[i...i+2]	last facet vertex, where i = ngeom-3
geom[i+3...i+5]	first facet vertex normal

ngeom = 3 times the number of facet vertices	
geom[k...k+2]	last facet vertex normal, where k = 2(ngeom-3)
geom[k+3...k+5]	first facet vertex (u,v,t) information
geom[l...l+2]	last facet vertex (u,v,t), where l = 3(ngeom-1)

4.4.3.11 Non-rational B-curves: L3TPPC

ngeom = the number of Bezier vertices defining the curve	
geom[0...2]	first Bezier vertex
geom[3...5]	second Bezier vertex
geom[i...i+2]	last Bezier vertex, where i = 3(ngeom-1)

4.4.3.12 Rational B-curves: L3TPRC

ngeom = the number of points	
geom[0...2]	first Bezier vertex
geom[3]	first weight
geom[4...6]	second Bezier vertex
geom[7]	second weight
geom[i...i+2]	last Bezier vertex, where i = 4(ngeom-1)
geom[i+3]	last weight, where i = 4(ngeom-1)

4.4.3.13 Non-rational B-curves in NURBs form: L3TPNC

ngeom = 3 (number of b-spline vertices) + number of knots	
geom[0...2]	first b-spline vertex
geom[3...5]	second b-spline vertex
geom[i...i+2]	last b-spline vertex, where i = 3(nvertices-1)
geom[3(nvertices)]	first knot, (3(nvertices) = i+3)
geom[3(nvertices)+1]	second knot
geom[3(nvertices)+nknots -1]	last knot

The number of b-spline vertices is supplied in the 9th element of the integer array and the number of knots is supplied in the 10th element of the integer array.

4.4.3.14 Rational B-curves in NURBs form: L3TPRN

ngeom = 4 (nvertices) + nknots	
geom[0...2]	first b-spline vertex
geom[3]	first weight
geom[4...6]	second b-spline vertex
geom[7]	second weight
geom[i...i+2]	last b-spline vertex, where i = 4(nvertices-1)

ngeom = 4 (nvertices) + nknots	
geom[i+3]	last weight,
geom[4(nvertices)]	first knot, (4(nvertices) = i+4)
geom[4(nvertices)+1]	second knot
geom[4(nvertices)+nknots-1]	last knot

The number of b-spline vertices is supplied in the 9th element of the integer array and the number of knots is supplied in the 10th element of the integer array.

4.4.3.15 For facet vertices + normals + parameters + 1st derivatives

L3TPF1; and facet strip vertices + normals + parameters + 1st derivatives – L3TPT1

ngeom = 5 times the number of facet vertices	
geom[0...2]	first facet vertex
geom[i...i+2]	last facet vertex, where i = ngeom-3
geom[i+3...i+5]	first facet vertex normal
geom[k...k+2]	last facet vertex normal, where k = 3(2ngeom/5-1)
geom[k+3...k+5]	first facet vertex (u, v, t) information
geom[l...l+2]	last facet vertex (u, v, t), where l = 3(3ngeom/5-1)
geom[l+3...l+5]	first dP/du derivative
geom[m...m+2]	last dP/du derivative where m = 3(4ngeom/5-1)
geom[m+3...m+5]	first dP/dv derivative
geom[n...n+2]	last dP/dv derivative where n = 3(ngeom-3)

4.4.3.16 For facet vertices + normals + parameters + all derivatives

L3TPF2; and facet strip vertices + normals + parameters + all derivatives – L3TPT2

ngeom = 8 times the number of facet vertices	
geom[0...2]	first facet vertex
geom[i...i+2]	last facet vertex, where i = 3(ngeom/8-1)
geom[i+3...i+5]	first facet vertex normal
geom[k...k+2]	last facet vertex normal, where k = 3(2ngeom/8-1)
geom[k+3...k+5]	first facet vertex (u, v, t) information
geom[l...l+2]	last facet vertex (u, v, t), where l = 3(3ngeom/8-1)
geom[l+3...l+5]	first dP/du derivative
geom[m...m+2]	last dP/du derivative, where m = 3(ngeom/2-1)
geom[m+3...m+5]	first dP/dv derivative
geom[n...n+2]	last dP/dv derivative, where n = 3(5ngeom/8-1)
geom[n+3...n+5]	first d2P/du2 derivative
geom[p...p+2]	last d2P/du2 derivative, where p = 3(6ngeom/8-1)

ngeom = 8 times the number of facet vertices	
geom[p+3...p+5]	first d2P/dudv derivative
geom[q...q+2]	last d2P/dudv derivative, where q = 3(7ngeom/8-1)
geom[q+3...q+5]	first d2P/dv2 derivative
geom[r...r+2]	last d2P/dv2 derivative, where r = 3(ngeom/8-1)

4.4.3.17 Complete sphere L3TPCS

ngeom = 4	
geom[0...2]	center point
geom[3]	radius

4.4.3.18 Complete cylinder L3TPCY

ngeom = 7	
geom[0...2]	start point
geom[3...5]	end point
geom[6]	radius

4.4.3.19 Complete truncated cone L3TPCN

ngeom = 8	
geom[0...2]	start point
geom[3...5]	end point
geom[6]	start radius
geom[7]	end radius

4.4.4 Segment types

As stated earlier, the first argument of each segment output function is the segment type. A segment of a particular **type** is always of the same class:

- **body** segments are always **hierarchical** (they cannot be output by GOSGMT, only by GOOPSG and GOCLSG)
- **edge** segments, **silhouette** segments, **hatch-line** segments are hierarchical when the `hierarch` option is used (like bodies, they cannot be output by GOSGMT, only by GOOPSG and GOCLSG)
- when the `hierarch` option isn't used they are all **single-level** (and therefore are only output by GOSGMT)

The segment types with their dependent data are as follows.

4.4.4.1 Body: SGTPBY

This type of hierarchical segment corresponds to an occurrence of a body in the model. If an entity within a body is passed to a rendering function, Parasolid still opens the body segment with GOOPSG before outputting the requested entity, and closes the body segment afterwards. This lets you build a graphical data structure and subsequently update it.

- `tags` holds the tag of the body
- `ngeom` is 6 and `geom` holds the model space box of the body, in the order: xmin, ymin, zmin, xmax, ymax, zmax.

There is no geometric data apart from the body box, as all the lines which make up the body in the picture form separate segments within the body segment.

4.4.4.2 Face: SGTPFA

This type of hierarchical segment corresponds to an occurrence of a face in a model. GOOPSG allows you to build a graphical data structure in the same way as for bodies as explained above.

- `tags` holds the tag of the face
- `ngeom` is 6 and `geom` holds the model space box of the face, defined in the same way as the body box, as described above.

  This segment type is only produced by faceting.

4.4.4.3 Region: SGTPRE

This type of hierarchical segment corresponds to an occurrence of a region in a model. GOOPSG allows you to build a graphical data structure in the same way as for bodies as explained above.

- `tags` holds the tag of the region
- `ngeom` is 6 and `geom` holds the model space box of the region, defined in the same way as the body box, as described above.

  This segment type is only produced by PK_TOPOL_render_volume.

4.4.4.4 Shell: SGTPSH

This type of hierarchical segment corresponds to an occurrence of a shell in a model. GOOPSG allows you to build a graphical data structure in the same way as for bodies as explained above.

- `tags` holds the tag of the shell
- `ngeom` is zero and `geom` is empty.

  This segment type is only produced by PK_TOPOL_render_volume.

4.4.4.5 Frame: SGTPFR

This type of hierarchical segment corresponds to an occurrence of a frame in a model. GOOPSG allows you to build a graphical data structure in the same way as for bodies as explained above.

- `tags` holds the tag of the frame
- `ngeom` is zero and `geom` is empty.

  This segment type is only produced by PK_TOPOL_render_volume.

The following are all single-level segment types which may be output by GOSGMT:

4.4.4.6 Edge: SGTPED

These represent edges or portions of edges. They are produced by PK_TOPOL_render_line if you specified the `edge` option. An edge segment may be a complete edge (E) of the model or may be only part of an edge, for example:

- If rendering view independent topology with unfixed blends, where an adjacent edge has a blend attribute, and an effect of the blend is to shorten the edge (E): only the part unaffected by the blend is drawn. If you are drawing the part a few edges at a time, the edge (E) is shortened only if you rendered it in the same call to PK_TOPOL_render_line as the edge which is blended.
- In hidden line drawings when the edge is partly visible and partly not, visible portions of (E) are output in separate calls to GOSGMT. If the `invisible` or `drafting` options are used, the invisible portions of (E) are also output by further calls to GOSGMT.
- In hidden line drawings with the regional data option: if the edge bounds a face being rendered with regional data, or divides such a face into visible and invisible parts, the places where the representation of (E) should meet other lines on the 2-dimensional drawing divide (E) up into separate parts which are output by separate calls to GOSGMT (see Section 4.5).
  - If regional data is not required `tags` contains the tag of the edge in the model.
  - If regional data is required `tags` contains the tag of the edge in the model and two extra tags, identifying the faces either side of the line in the 2-dimensional drawing. Either or both of these face-tags may be PK_ENTITY_null.

If regional data is required `lntp` contains two extra integer values: the indices of the start and end points of the line (see Section 4.5).

4.4.4.7 Silhouette line: SGTPSI

A silhouette is a line on a single face curving away from the eye where its surface changes from visible to hidden. Both view dependent topology and hidden line drawings produce silhouettes. They may be output whole, or cut or shortened in the same way as for edges, see above.

- If regional data is not required `tags` contains the tag of face bearing the silhouette.
- If regional data is required `tags` contains two extra tags, and `lntp` contains point indices, as for edges.

See Section 4.5, “Interpreting regional data”, for information on regional data.

4.4.4.8 Planar hatch-line: SGTPPH

- `tags` contains the tag of the face bearing the hatch-line.

4.4.4.9 Radial hatch-line: SGTPRH

- `tags` contains the tag of the face bearing the hatch-line.

4.4.4.10 Rib line on unfixed blend: SGTPRU

This is a further way of rendering an unfixed blend: adding lines across the blend surface, roughly perpendicular to the original edge. Rib lines can be drawn as well as a blend boundary, but not instead of it. As for blend boundaries, rib lines can only be produced by a view independent drawing.

- `tags` contains the tag of the edge.

4.4.4.11 Blend-boundary line on unfixed blend: SGTPBB

If an edge with an unfixed blend is being rendered as view independent topology with unfixed blends, the blend is rendered instead of the edge. The way in which the blend is rendered depends on the option data provided with the unfixed blend option, or if this is absent, on the attribute data associated with the blend. Unfixed blends are ignored by all the other rendering functions.

See the Section 106.3.19, “Unfixed blends”, in the Parasolid Functional Description manual for further information on rendering unfixed blends.

A blend boundary is the line where the blending surface meets the faces or other blends adjacent to the edge.

- `tags` contains the tag of the blended edge.

4.4.4.12 Parametric Hatch line: SGTPPL

`tags` contains the tag of the face.

4.4.4.13 Facet: SGTPFT

A facet is a planar or near planar polygon. A face rendered by the facet rendering function is approximated by a collection of contiguous facets. The data supplied is dependent on the setting of the rendering options to the faceting function.

See Chapter 108, “Facet Mesh Generation” and Chapter 109, “Faceting Output Via GO” in the Parasolid Functional Description manual for further information on faceting.

- If edge tag data is not required, `tags` contains the tag of the face on which the facet lies.
- If edge tag data is required, `tags` contains the tag of the face on which the facet lies and also contains tags of the model edges from which each facet edge is derived. The number of edge tags equals the number of vertices which define the facet. The null tag is supplied if the facet edge is not derived from a model edge. The first edge tag (tag[1]) is the tag of the model edge from which the first facet edge is derived. The first facet edge ends at the first vertex given in the geom array, see below. The second edge tag is for the facet edge which ends at the second vertex in geom, and so on.
- Extra data is supplied in `lntp` for this segment type:
  - `lntp[2]` contains the number of loops in the facet
  - `lntp[3]` contains the number of vertices in the first loop
  - `lntp[4]` contains the number of vertices in the second loop

  and so on.
- `ngeom` and `geom` depend on the geometry type of this segment as specified in the second element of `lntp`.

If a facet has multiple loops, the outer loop is output first and the inner loops follow. The vertices of the outer loop are ordered counter-clockwise when viewed down the surface normal. The vertices of inner loops are ordered clockwise. Facets are manifold. That is, no vertex coincides with any other in the same facet, nor does it lie in any edge in the same facet.

This type of facet is only produced by the facet drawing function.

4.4.4.14 Facet from volume rendering: SGTPVF

A volume facet is a planar triangle. A face rendered by the volume rendering function PK_TOPOL_render_volume is approximated by a collection of contiguous facets,

See Section 20.10.4, “Imprinting embedded lattices” in the Parasolid *Functional Description* for more information on volume rendering.

- `tags` contains the tag of the shell followed by the tag of the face on which the facet lies.

The vertices of the facet are ordered counter-clockwise when viewed down a normal pointing out of the region.

This type of volume facet is only produced by PK_TOPOL_render_volume.

4.4.4.15 Error Segment: SGTPER

When rendering a list of entities, Parasolid may encounter a body, face or edge which it is unable to render (e.g. a rubber face). In such a case, Parasolid outputs an error segment giving the tag of the bad entity a code indicating why it was not rendered. When the error segment has been output, Parasolid continues to render the remaining entities.

- `tags` holds the tag of the entity which could not be rendered.
- `ngeom` is zero

4.4.4.16 Geometric Segments: SGTPGC, SGTPGS, SGTPGB

Geometric segment types SGTPGC (curves), SGTPGS (surfaces), and SGTPGB (surface boundaries) are used to sketch unchecked parametric curves and surfaces as view independent drawings enabling the relevant curve/surface to be visualised.

4.4.4.17 Mangled Facet: SGTPMF

If during a call to the facet drawing function, user tolerances cannot be matched, or facets are created which self intersect or are severely creased, then geometric data is output as a segment type SGTPMF. In this case, the facets are always triangular.

4.4.4.18 Volume Mangled Facets: SGTPMV

If during a call to PK_TOPOL_render_volume, user tolerances cannot be matched, or volume facets are created which self intersect or are severely creased, then geometric data is output as a segment type SGTPMV.

4.4.4.19 Visibility Segment: SGTPVT

If PK_TOPOL_render_line is used to output data hierarchically from a hidden line drawing (that is, if the `hierarch` option is anything but PK_render_hierarch_no_c), then the single level segments output for each edge, silhouette, or hatch-line are:

- a geometry segment (when `hierarch` is PK_render_hierarch_yes_c or PK_render_hierarch_param_c)
- a visibility segment

If regional data has been requested, `tags` holds the regional information for the segments between the visibility transition points. This means that `ntags` is twice the value of `nlntp`, since `nlntp` represents the number of visibility code *pairs* (see below).

- `tags[0]` is the tag of the face to the left of the segment after the first transition point
- `tags[1]` is the tag of the face to the right of the segment after the first transition point
- `tags[2]` is the tag of the face to the left of the segment after the second transition point
- `tags[2n-2]` is the tag of the face to the left of the segment after the nth transition point
- `tags[2n-1]` is the tag of the face to the right of the segment after the nth transition point

If regional data is not requested, then `ntags` is zero and `tags` contains no regional information.For more information on regional data, see Section 4.5, “Interpreting regional data”.

`geom` holds the visibility transition points, i.e. the vectors in model space where the edge changes visibility or smoothness. `ngeom` holds the number of these visibility transition points.

The `lntp` array is structured as shown in Figure 4–2 and Figure 4–3. The array consists of five blocks of data, as follows:

- The first block, of length `nlntp`, holds the visibility codes for the edge.
- The second block, of length `nlntp`, holds the smoothness codes for the edge. The smoothness property can change along edges which are partially coincident with silhouettes. The smoothness code contained in the geometry segment should be ignored. See Section 4.4.2.4, “Smoothness” for more information.
- The third block, of length `ngeom`, holds the point indices for the visibility transition points if regional data has been requested.
- If the `viewport_clipping` option in PK_TOPOL_render_line_o_t is set to PK_render_viewport_clip_yes_c, then the fourth block, of length `nlntp`, holds the viewport codes describing whether an edge is inside, outside, or coincident with a viewport.
- If the `sharp_mfins` option in PK_TOPOL_render_line is set to PK_render_sharp_mfin_yes_c and this visibility segment is for an edge line, then the fifth block, of length `nlntp`, holds the codes describing whether or not an edge lies on a chain of sharp mfins. The sharp mfins code can change along edges which are partially coincident with sharp mfins. Therefore, the sharp mfins code contained in the geometry segment should be ignored. See Section 4.4.2.6, “Coincidence with sharp mfins” for more information.

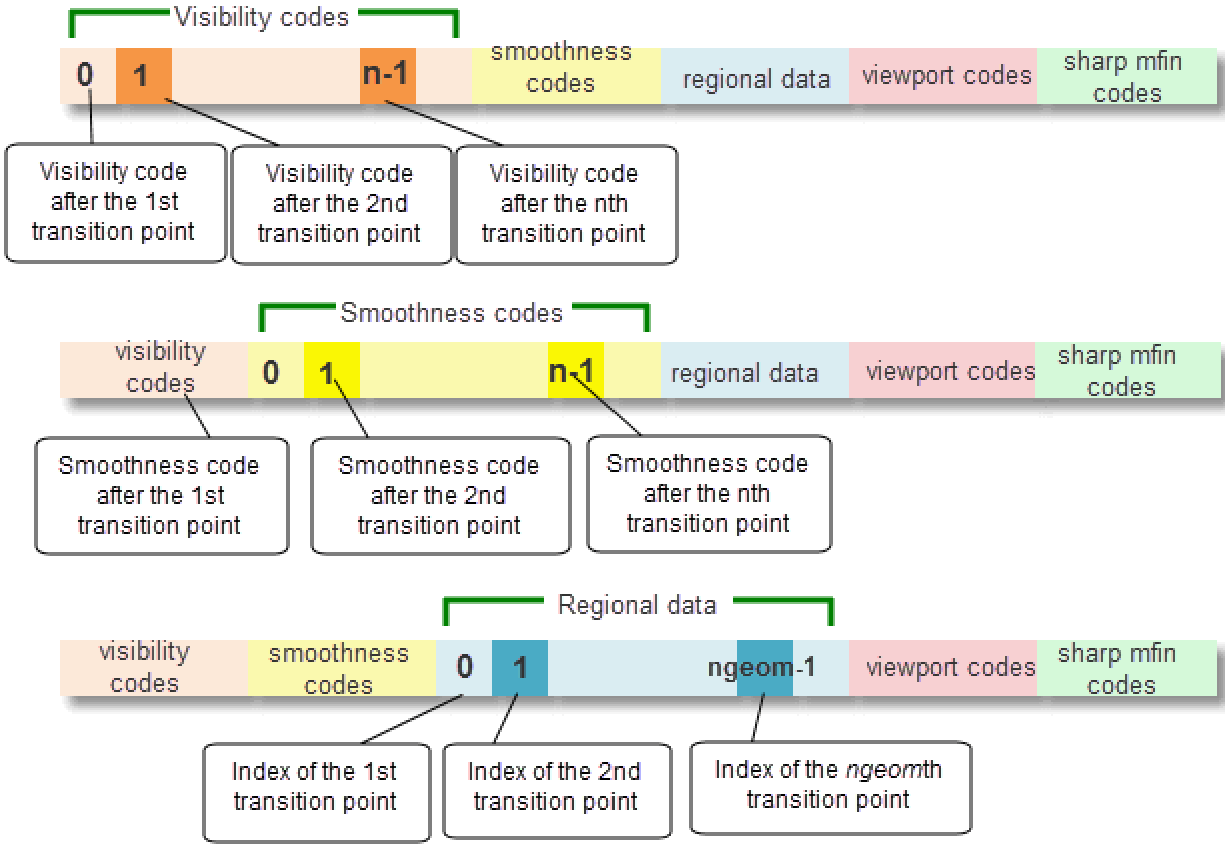
Visibility codes
0
1
n-1
smoothness codes
regional data
viewport codes
sharp mfin codes
Visibility code after the 1st transition point
Visibility code after the 2nd transition point
Visibility code after the nth transition point
Smoothness codes
visibility codes
0
1
n-1
regional data
viewport codes
sharp mfin codes
Smoothness code after the 1st transition point
Smoothness code after the 2nd transition point
Smoothness code after the nth transition point
Regional data
visibility codes
smoothness codes
0
1
ngeom-1
viewport codes
sharp mfin codes
Index of the 1st transition point
Index of the 2nd transition point
Index of the *ngeom*th transition point

***Figure 4–2** Structure of the `lntp` array explaining visibility codes, smoothness codes and regional data*

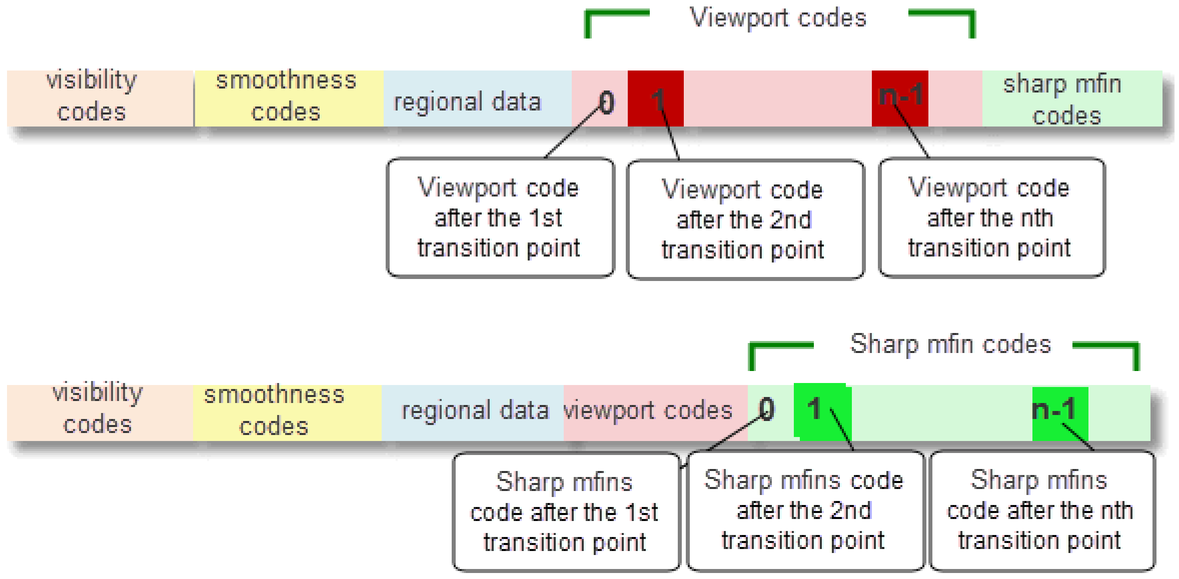
Viewport codes
visibility codes
smoothness codes
regional data
0
1
n-1
sharp mfin codes
Viewport code after the 1st transition point
Viewport code after the 2nd transition point
Viewport code after the nth transition point
Sharp mfin codes
visibility codes
smoothness codes
regional data
viewport codes
0
1
n-1
Sharp mfins code after the 1st transition point
Sharp mfins code after the 2nd transition point
Sharp mfins code after the nth transition point

Figure 4–3 *Structure of the `lntp` array explaining viewport codes and sharp mfin codes*

4.4.4.20 Facet Strip: SGTPTS

Using the 'facet strip' option when outputting data from the facet drawing function results in this data being output in the form of a 'strip' or 'ribbon' consisting of triangular facets.

The number of facets in each strip must be specified in the option data list when using this option.

Triangular facets share vertices between adjacent facets with the geometry specifying the vertices of the triangles in a particular order. For example, in a strip consisting of eight triangular facets the order of the vertices are specified as follows:

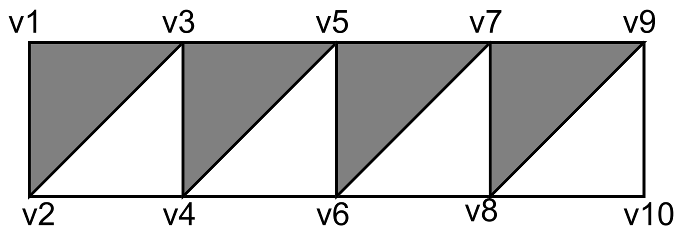

Figure 4–4 *The ordering of vertices in a facet strip*

As can be seen from the above example, triangular strip geometry only stores ‘n + 2’ vertices, whereas an individual-triangle’s geometry stores ‘3n’ vertices.

4.4.4.21 Parametrized Visibility Segment: SGTPVP

If the `hierarch` option is used to output data hierarchically from a hidden line drawing then the single level segments output for each edge, silhouette, or hatch line is very similar to those output when the other hierarchical options are specified, i.e.

- a geometry segment
- a visibility segment

However, when the geometry segment is a polyline, the visibility segment supplied is of type SGTPVP rather than SGTPVT:

- `ntags` and `tags` are zero, as this information is provided by the hierarchical segment.
- `ngeom` holds the number of visibility transition points.
- `geom` holds the visibility transition points. These points are sets of four values, defining both the vector position of the change in visibility and its parameter along the polyline, i.e.
  - geom[0...2] vector position of first change in visibility
  - geom[3] parameter of first change in visibility
  - geom[i...i+2] nth vector position, where i = 4(n-1)
  - geom[i+3] nth parameter
- `nlntp` holds the number of visibility codes.
- `lntp` holds the visibility codes for the edge.

The parameterisation of polylines is pseudo arc-length, normalized so that the parameter interval of any polyline is always [0,1]. That is, the parameter of any point on the polyline is equal to the distance between the point and the start measured along the polyline, divided by the total length of the polyline.

The parameters are supplied in order to help the application locate the chord in the polyline on which the associated visibility transition point lies.

Given a polyline P with N chords, defined by the set of 3-D points:

p_i (where 0 $0 \le i \le N$)

we define the total length of a polyline consisting of N chords as:

$$L(N) = \sum_{0 £ i < N} |p_{i+1} - pi|$$

and define distance D(t) as the length of the polyline at parameter value t measured from the point p_0.

Given a visibility transition point v with parameter value $t \ (0 \le t \le 1)$ we can find the chord $p_n \rightarrow p_{n+1}$ on which the point v lies by finding a point index n such that

$$L(n) \le D(t) < L(n+1)$$

The position v is given by:

$$p_n + (p_{n+1} - p_n)\left(\frac{D(t) - L(n)}{L(n+1) - L(n)}\right)$$

4.4.4.22 Interference curve: SGTPIC

If, in PK_TOPOL_render_line, overlapping bodies are detected during the rendering process by setting the `overlap` option to PK_render_overlap_intersect_c, then this segment type is used for any curves that are generated as a result of intersections between overlapping faces in the bodies. See Section 106.3.20, “Overlapping bodies”, in the Parasolid *Functional Description*, for more information.

4.4.4.23 Clip line: SGTPCL

If, in PK_TOPOL_render_line, 3D viewports are enabled by setting `viewport_type` to PK_render_viewport_type_3D_c and viewport clipping is enabled by setting `viewport_clipping` to PK_render_viewport_clip_yes_c, then any clashes between rendering faces and viewport faces will result in the generation of a clip line. Segments on clip lines are marked with the code SGTPCL. See Section 106.3.23.3, “Clipping entities to viewport boundaries”, in the *Functional Description*, for more information about viewports.

4.4.4.24 Sharp mfin line: SGTPSF

If, in PK_TOPOL_render_line, sharp mfins are requested by setting `sharp_mfins` to PK_render_sharp_mfins_yes_c then polylines that run along chains of connected sharp mfins will be rendered for every face that contains facet geometry. See Section 106.3.4, “Sharp mfins”, in the *Functional Description*, for more information.

4.4.4.25 Lattice geometric segments: SGTPLB, SGTPLR

Lattice segment types SGTPLB (lattice balls), and SGTPLR (lattice rods) are used to render lattice balls as spheres and lattice rods as straight lines between the 2 balls it connects. See Section 106.2.4, “Lattice”, in the *Functional Description* for more information.

4.5 Interpreting regional data

Regional data is produced in a hidden line drawing when you use the `region` option in PK_TOPOL_render_line_o_t. It tells you how to split a hidden line picture into separate 2D regions, as shown in the Figure 4–5.

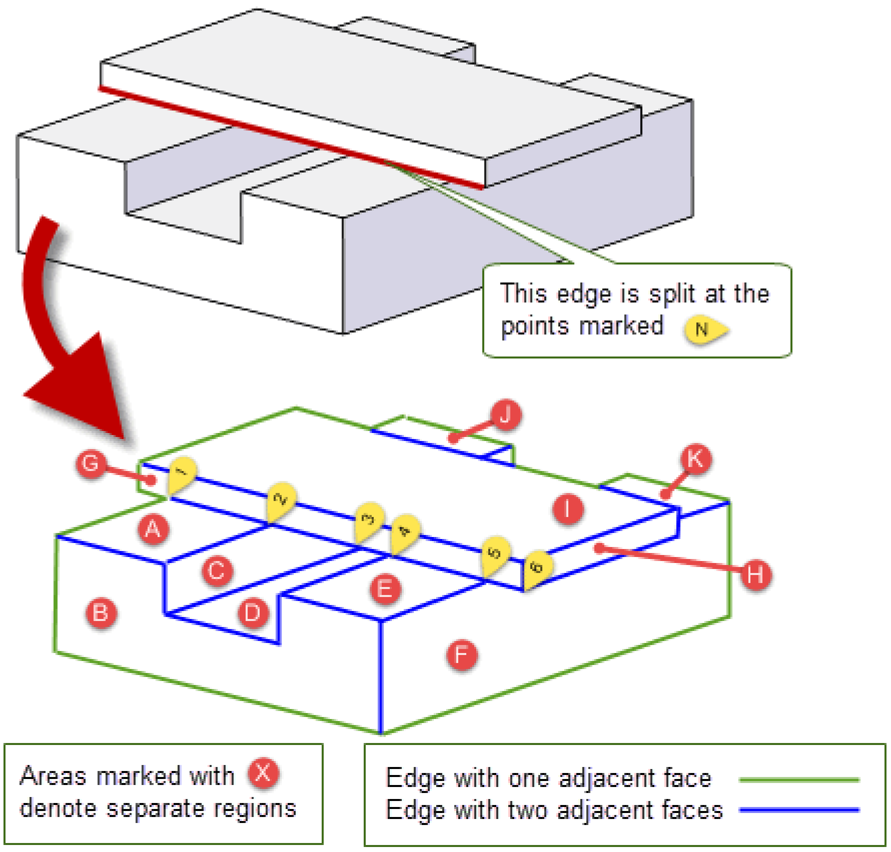
This edge is split at the points marked N
J
G
K
I
A
C
E
H
B
D
F
1
2
3
4
5
6
Areas marked with X denote separate regions
Edge with one adjacent face
Edge with two adjacent faces

Figure 4–5 *Interpreting regional data*

A single edge may bound several regions on the two-dimensional picture (for instance the edge marked in Figure 4–5, bounds regions A, C, D, E, F and G). When this happens it is divided at the intersections, and output as several segments, with the same basic segment data (and completeness code CODINC), but different regional data. The additional data is of two types: adjacent faces, and point indices.

4.5.1 Adjacent faces

The order of faces returned in regional data is based on the orientation of the faces either side of the boundary line in question, relative to the direction of the curve and viewing direction.

When a bounding line is output with regional data, the `tags` array is of length 3 (i.e. `ntags`=3):

- `tags[0]` contains the tag of the original model entity: either an edge (if the bounding line is generated from a model edge) or a face (if the bounding line is generated from another source, such as a silhouette)
- `tags[1]` and `tags[2]` contain either the tags of faces in the model, or PK_ENTITY_null. These indicate which faces are on each side of the line corresponding to the segment in the 2D picture. (The faces may or may not be adjacent to the original edge or silhouette in the 3D model.)

PK_ENTITY_null indicates that the region of the picture on that side of the line is *either* part of a face not tagged for regional data *or* outside the 2D representation of the model (the “outside” of the picture).

Face tags are returned relative to the direction of the line, which is required to interpret the point indices correctly: `tags[1]` is the **left** face, and `tags[2]` the **right**.

Figure 4–6 illustrates how regional data is returned when rendering three cubes, with a view direction such that only one face from each cube is visible, two of which are partially obscured. The illustration shows how regional data is returned for a sample of edges in the rendered image, shown in red. For simplicity, PK_ENTITY_null is shown as 0 in the illustration, and `tags[0]` is not shown.

- For each segment with two adjacent faces, the left face (relative to the direction of the edge) is returned first, and the right face is returned last.
- For each segment with only one adjacent face, PK_ENTITY_null is returned for the side that has no face.
- For each segment that is completely obscured in the rendered image, PK_ENTITY_null is returned for both left and right faces.

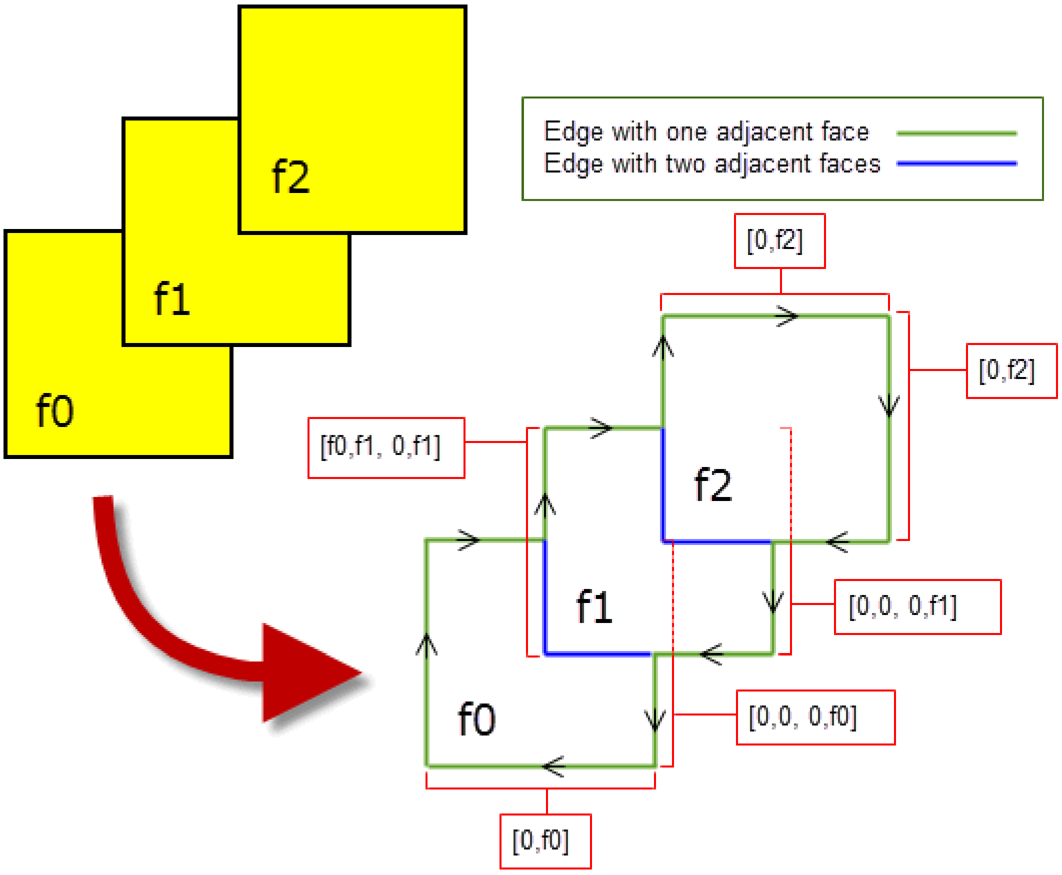

Figure 4–6 *Format of regional data for edges*

4.5.2 Point indices

The `lntp` array is of length 7 for a segment with regional data. Elements `lntp[5]` and `lntp[6]` are the **start index** and **end index** respectively for the line. They are non-zero integer values, and specify which "points" of the two-dimensional picture the segment joins.

Suppose segment A has end index *x*, and segment B has start index *x*. Then the end point of A and the start point of B should be regarded as the same point in the two-dimensional picture, *even if their geometric projections do not exactly coincide*. (This may happen as the result of numerical approximations in rendering.) You will also find that of all the lines sharing a point index, one with it as an end index and one with it as a start index share an adjacent face on the left, and so these two can be linked up as consecutive portions of the boundary of a region; and similarly with faces on the right. The values used for point indices are not in any meaningful order.

4.6 Graphical output of pixel data

Another part of the GO consists of three functions for producing pixel data. These were required to support the KI function RRPIXL. These functions do not need to be implemented to support the PK (they can be supplied in dummy form).

Function	Description
GOOPPX	open output of encoded pixel data
GOPIXL	output encoded pixel data
GOCLPX	close output of encoded pixel data

See Appendix J, “Legacy Functions”, for further information on the interface to these functions.

Registering the Frustrum 5

5.1 Introduction

The application writer has several options for providing the Frustrum functions for Parasolid to use.

5.2 Object-file frustrum

When Parasolid is used as an object-file library, the Frustrum, GO and Foreign Geometry functions must also be compiled and linked with it. These functions are specified in Appendix A, “Frustrum Functions” and Appendix B, “Graphical Output Functions”.

The functions can be split into logical groups, as shown in the following table:

Group	Functions
Control	FSTART FABORT FSTOP
Memory Management	FMALLO FMFREE
File I/O	FFOPRD FFOPWR FFSKXT FFREAD FFWRIT FFCLOS
Graphics	GOOPSG GOSGMT GOCLSG
Foreign Geometry (curves)	FGCRCU FGEVCU FGPRCU
Foreign Geometry (surfaces)	FGCRSU FGEVSU FGPRSU
Rollback (obsolete)	FFOPRB FFSEEK FFTELL
Shaded Images (obsolete)	GOOPPX GOPIXL GOCLPX

Simple functions are provided for initial testing in the files `frustrum.c` and `fg.c` in the Parasolid release area.

5.3 Registered frustrum

Parasolid may be supplied as a shared image as well as an object-file library.

If the following method is used to provide a registered frustrum, the functions do not need to have the 6-letter FORTRAN-style names.

Parasolid still needs to call the Frustrum, GO and Foreign Geometry functions but the image must use a different mechanism to the object-file library. This is done using the PK functions PK_SESSION_register_frustrum or PK_SESSION_register_fru_2. The installed Frustrum can then be identified using the PK functions PK_SESSION_ask_frustrum or PK_SESSION_ask_fru_2.

There is an example of registering the Frustrum using PK_SESSION_register_frustrum included in the file `parasolid_test.c` in the Parasolid release area.

A C structure is defined, with an element for each required function, and the application must register this with Parasolid before starting the modeller. This mechanism can also be used with the supplied object-file library: it is not specific to the shared image implementation. The advantages to the application of using a registered frustrum are that:

- The application-supplied functions no longer need to have the six letter FORTRAN-style names (as specified in the “Frustrum Functions” chapter).
- The application no longer needs to supply all the specified functions: e.g. if the application uses the PK_DELTA_* functions for partitioned rollback, the functions FFOPRB, FFSEEK and FFTELL need not be registered.

All applications must supply equivalents of FMALLO and FMFREE. An application can omit group of functions as required.

If a non-registered function is accessed, Parasolid may fail with PK error code PK_ERROR_fru_missing. As an example, an application that does not make use of Foreign Geometry might receive a part from another application that does.

An application using the shared image can replace the Parasolid image with an updated version without relinking. **It is important that the Parasolid version in the new library is later than the old one.** To guard against an incompatible combination, the application can enquire the version number of the installed Parasolid using PK_SESSION_ask_kernel_version. For example:

```
PK_SESSION_kernel_version_t info;
PK_SESSION_ask_kernel_version (&info);
if (info.major_revision<9)
    {
    fprintf (stderr, "Parasolid v9 is required");
    exit (EXIT_FAILURE);
    }
```

This function may be called at any time, in particular, without starting the modeller, by calling this function in the shared image.

If you wish to transmit and receive embedded mesh data, or optimally transmit and receive mesh data, you need to register the frustrum seek function FFSKXT using PK_SESSION_register_fru_2. This frustrum allows for the optimised transmitting and receiving of binary XT files that contain embedded meshes. You can identify the installed frustrum using PK_SESSION_ask_fru_2.

If you wish to call Parasolid’s rendering and faceting functions concurrently, or to use SMP when generating graphical output, you can indicate that the GO functions, GOSGMT, GOOPSG and GOCLSG, are thread-safe using the `go_thread_safe` option in PK_SESSION_register_fru_2. Where practical, we recommend a thread-safe GO implementation to allow your application to benefit from the full set of performance optimisations available in Parasolid.

Note: We strongly recommend that you register your frustrum using PK_SESSION_register_fru_2

5.4 Application I/O

There is a transmit file format called ‘application i/o’, or ‘applio’. When this format is selected in PK_PART_transmit and PK_PARTITION_transmit, transmit files are written and read using a suite of functions provided by the application. Using these functions enables the application to do further processing of the output data before storing it.

The functions open files, read and write chars, bytes, shorts, ints and doubles to/from these files, and close the files; they are registered using PK_SESSION_register_applio.

Note that the application is responsible for any conversion required between machine types (e.g. for endian byte ordering and floating point representation). The read functions must be handed the correct number of computation-ready data items, as written out by the write functions.

Snapshot files cannot use this format – they must be text or machine-dependent binary.

5.5 Indexed I/O

There is also a transmit file format called **indexed i/o**, or **indexio**. When this `transmit_format` is selected in PK_PART_transmit and PK_PARTITION_transmit, transmit files are written using a suite of functions provided by the application. Using these functions enables your application to subsequently receive only specified faces from the parts contained in the transmitted files.

The functions open files, read from and write to these files in both Unicode and non-Unicode format, and close the files; they are registered using PK_SESSION_register_indexio.

Frustrum Functions A

A.1 List of frustrum functions

This appendix contains the specifications of the Frustrum functions required by the PK functions for file and memory handling.

Function	Description	For more information
`FSTART`	Start up the frustrum	PK_FSTART_f_t
`FABORT`	Called at the end of an aborted kernel operation	PK_FABORT_f_t
`FSTOP`	Shut down the frustrum	PK_FSTOP_f_t
`FMALLO`	Allocate virtual memory	PK_FMALLO_f_t
`FMFREE`	Free virtual memory	PK_FMFREE_f_t
`FFOPRD`	Open all guises of file for reading	PK_FFOPRD_f_t
`FFOPWR`	Open all guises of file for writing	PK_FFOPWR_f_t
`UCOPRD`	Open various guises of file for reading using Unicode key	PK_UCOPRD_f_t
`UCOPWR`	Open various guises of file for writing using Unicode key	PK_UCOPWR_f_t
`FFCLOS`	Close file	PK_FFCLOS_f_t
`FFREAD`	Read from file	PK_FFREAD_f_t
`FFWRIT`	Write to file	PK_FFWRIT_f_t
`FFSKXT`	Seek within the file by resetting the file pointer for C-transmit and partition guises.	PK_FFSKXT_f_t

Graphical Output Functions B

B.1 Introduction

This appendix lists the Graphical Output functions that your application should register with Parasolid using PK_SESSION_register_fru_2; these functions render graphical data generated by the PK line drawing functions:

- PK_GEOM_render
- PK_TOPOL_render_line
- PK_TOPOL_render_facet
- PK_TOPOL_render_volume

Further information on these functions can be found in the following chapters of the Parasolid *Functional Description*:

- Section 20.10.5, “Rendering embedded lattices”
- Chapter 105, “Rendering Functions”
- Chapter 106, “Rendering Option Settings”
- Chapter 108, “Facet Mesh Generation”
- Chapter 109, “Faceting Output Via GO”

B.2 Registering the Graphical Output functions

The Graphical Output functions must be registered with Parasolid by calling the PK function PK_SESSION_register_fru_2 at any time during a Parasolid session.

You can register Graphical Output functions independently of the other frustrum functions or in the same call. You can use PK_SESSION_ask_fru_2 to return the currently registered frustrum functions if necessary.

B.3 Graphical Output functions that can be registered

This section lists the Graphical Output functions that you can register during a Parasolid session.

Note: In the following list, the function names given are purely nominal, as the functions are registered by the call to PK_SESSION_register_fru_2.

Function	Description	For more information
`gosgmt`	Output a non-hierarchical segment	PK_GOSGMT_f_t
`goopsg`	Open a hierarchical segment	PK_GOOPSG_f_t
`goclsg`	Close a hierarchical segment	PK_GOCLSG_f_t

PK_DELTA Functions C

C.1 Introduction

This appendix contains the specifications of the Frustrum functions required for the PK partitioned rollback system.

Partitioned rollback requires six registered frustrum functions. Together these functions provide a virtual file system in which byte streams may be created or read. Byte streams are denoted by PK_DELTA_t values, not filenames, and are referred to as **delta files**. The delta value, which is positive, is assigned by the application Frustrum.

The partition rolling mechanism works by storing the changes between pmarks. These record the entities which need to be created, modified or deleted in order to move from one pmark to an adjacent one (either backwards or forwards). These deltas are written out through the Frustrum interface, stored by the application Frustrum, and read back in during a roll operation.

C.1.1 Example PK_DELTA frustrum code

The file `frustrum_delta.c` in the Parasolid release area lists the code for an example PK_DELTA Frustrum, required for running the partitioned PK rollback system.

The example Frustrum is provided for the following purposes:

- To allow the building and running of the Parasolid installation acceptance test program.
- To allow users to build and run simple prototype applications using rollback without first having to write a complete Frustrum.
- To aid users in writing their own Frustrum.

This Frustrum contains the bare minimum required to be used, in order for it to remain clear and platform independent. Normally a Frustrum is written with a particular application in mind, and may make use of system calls rather than the C run-time library for enhanced performance.

C.1.2 Criteria of use

- Delta files are not read beyond their length.
- Delta files deleted by Parasolid (via delete) are not referred to again.
- The Frustrum must only delete files when told to.
- There may be more than one delta file associated with a given pmark at a given time.
- If a new pmark is created, and the partition is currently at a pmark, a zero-length delta is output to the Frustrum, which must be stored.
- The pmark passed to the `open_for_write` function may sometimes be PK_PMARK_null, in which case the delta does not correspond to a pmark visible to the application. The application should store this delta as usual, and it is deleted by Parasolid when no longer required.

There is further information on the Frustrum requirements of Partitioned Rollback in Chapter 97, “Partitions”, of the Parasolid Functional Description Manual.

C.1.3 Registering the rollback frustrum functions

The partitioned rollback functions must be registered with Parasolid by calling the function PK_DELTA_register_callbacks, before the Parasolid session is started.

C.1.4 Delta functions that can be registered

This section describes the delta functions that you can register during a Parasolid session.

Note: In the following Frustrum function definitions, the function names given are purely nominal, as the functions are registered by the call to PK_DELTA_register_callbacks.

C.3.1.5 open_for_write

```
PK_ERROR_code_t open_for_write
(
PK_PMARK_t      pmark, /* pmark associated with delta */
PK_DELTA_t      delta  /* delta file to open          */
)
```

Opens a new delta file for writing, associated with the given pmark. Returns a PK_DELTA_t value chosen by the Frustrum which Parasolid uses to identify this delta file.

If `pmark` is PK_PMARK_null, the delta file is internal to Parasolid and is deleted when no longer required.

C.3.1.6 open_for_read

```
PK_ERROR_code_t open_for_read
(
PK_DELTA_t      delta /* delta file to open */
)
```

Opens an existing, closed delta file for reading.

C.3.1.7 close

```
PK_ERROR_code_t close
(
PK_DELTA_t      delta /* delta file to close */
)
```

Closes delta file `delta`, which is open. The function `close` is provided as a courtesy to the application and is invoked as early as possible.

C.3.1.8 write

```
PK_ERROR_code_t write
(
PK_DELTA_t      delta,   /* delta file to write      */
int             n_bytes, /* number of bytes to write */
char           *bytes    /* bytes to write           */
)
```

Writes n_bytes to the delta file delta (which is open) from the array bytes. n_bytes may often be as small as 20, so the application may wish to provide a buffering mechanism.

C.3.1.9 read

```
PK_ERROR_code_t read
(
PK_DELTA_t      delta,   /* delta file to
read                 */
int             n_bytes, /* number of bytes to
read            */
char           *bytes    /* array in which to store read
bytes */
)
```

Reads n_bytes from the delta file delta (which is open) to the array bytes. n_bytes may often be as small as 20, so the application may wish to provide a buffering mechanism. Parasolid never requests more bytes (in total) than were written. Parasolid does not guarantee that the sequence of values of n_bytes resembles those given to write.

If bytes is NULL then no data should be written to the array, but the file position should be advanced.

C.3.1.10 delete

```
PK_ERROR_code_t delete
(
PK_DELTA_t      delta /* delta file to delete */
)
```

The function delete is used by Parasolid to indicate that the given delta (which exists and is closed) is not required again. Parasolid performs no further operations (including delete) on delta.

PK_MEMORY Functions D

D.1 Introduction

This appendix contains the specifications of the Frustrum functions required for allocating and freeing memory, used when PK functions return variable length information. These functions are called by the PK functions PK_MEMORY_alloc and PK_MEMORY_free.

The functions should be type compatible with malloc and free in the standard C run-time library.

The section on Section 9.3, “Memory management” in the Parasolid *Functional Description* provides further information on the use of these functions.

The functions are allowed to long-jump out of Parasolid in case of an error, with the same restrictions and requirements as the user-registered error-handling function (see Chapter 121, “Error Handling”, of the Parasolid Functional Description manual).

D.1.1 Registering the memory management functions

The memory management functions must be registered with Parasolid by calling the function PK_MEMORY_register_callbacks. This function can be called before starting a Parasolid modelling session, or during a session whenever Parasolid’s internal PK memory is empty.

If the functions have not been registered, or either of the function pointers given to PK_MEMORY_register_callbacks is NULL, Parasolid defaults to using the appropriate function from the standard C run-time-library when the application calls PK_MEMORY_alloc or PK_MEMORY_free.

You can use PK_MEMORY_ask_callbacks to return pointers to the memory management functions currently registered with Parasolid.

D.1.2 Memory management functions that can be registered

This section lists the memory management functions that you can register during a Parasolid session.

Note: In the following list, the function names given are purely nominal, as the functions are registered by the call to PK_MEMORY_register_callbacks.

Function	Description	For more information
`alloc_fn`	Allocates a specified amount of memory required in bytes.	PK_MEMORY_alloc_f_t
`free_fn`	Frees previously allocated memory.	PK_MEMORY_free_f_t

Application I/O Functions E

E.1 Introduction

This appendix contains the specifications of the application I/O (applio) functions which can be implemented to replace or extend the normal part file handling functions in the Frustrum. These functions are used during input and output of transmit files with ‘application i/o’ format.

There is further information on the use of the application I/O functions in Section 2.3.2, “Application I/O”.

E.1.1 Registering the application I/O functions

The application I/O functions must be registered with Parasolid by calling the PK function PK_SESSION_register_applio_2, before the Parasolid session is started.

E.1.2 Application I/O functions that can be registered

This section lists the application I/O functions that you can register during a Parasolid session.

Note: In the following list, the function names given are purely nominal, as the functions are registered by the call to PK_SESSION_register_applio_2.

Function	Description	For more information
`open_rd`	Opens a file for reading with the given key.	PK_SESSION_applio_t
`open_wr`	Opens a new file for writing with the given key.	PK_SESSION_applio_t
`open_uc_rd`	Opens a file for reading with the given Unicode key.	PK_SESSION_applio_t
`open_uc_wr`	Opens a new file for writing with the given key.	PK_SESSION_applio_t
`open_rd_2, open_wr_2, open_uc_rd_2, open_uc_wr_2`	Equivalent to the above open functions but with additional guise argument to support partitions and deltas explicitly.	PK_SESSION_applio_t
`close`	Closes the given file.	PK_SESSION_applio_t
`rd_**** functions`	Reads one or more items of data from the given file.	PK_SESSION_applio_t
`wr_**** functions`	Writes one or more items of data to the given file.	PK_SESSION_applio_t

Indexed I/O Functions F

F.1 Introduction

This appendix contains the specifications of the indexed I/O (indexio) functions which can be implemented to replace or extend the normal part file handling functions in the frustrum in order to create an indexed frustrum. These functions are used during input and output of transmit files with ‘indexed i/o’ format, to allow specific faces in a part to be received into a Parasolid session, rather than requiring an entire part to be loaded.

There is further information on the use of the indexed i/o functions in Section 2.3.3, “Indexed I/O”.

F.2 Registering the indexed I/O functions

The indexed I/O functions must be registered with Parasolid by calling the PK function PK_SESSION_register_indexio. This function receives a PK_SESSION_indexio_t structure that contains pointers to the functions you want to register.

You can also call PK_SESSION_ask_indexio to return a structure that contains pointers to the functions currently registered.

F.3 Indexed I/O functions that can be registered

This section lists the indexed I/O functions that you can register during a Parasolid session.

Note: In the following list, the function names given are purely nominal, as the functions are registered by the call to PK_SESSION_register_indexio.

Function	Description	For more information
`ffoprd`	Opens the FFBNRY specified file for reading and stores any required information.	PK_SESSION_indexio_t
`ffopwr`	Opens a specified file for writing, and stores any required information.	PK_SESSION_indexio_t
`ucoprd`	Opens the specified FFBNRY file for reading in Unicode mode, and stores any required information.	PK_SESSION_indexio_t
`ucopwr`	Opens a specified file for writing in Unicode format, and stores any required information.	PK_SESSION_indexio_t
`ffread`	Reads from the open file starting at the current position of the file pointer.	PK_SESSION_indexio_t

Function	Description	For more information
`ffwrit`	Writes the given bytes to an open file, starting at the current file position.	PK_SESSION_indexio_t
`ffseek`	Reset file pointer.	PK_SESSION_indexio_t
`ffclos`	Closes a file which has been opened with a call to `ffoprd` or `ucoprd`.	PK_SESSION_indexio_t

Attribute Callback Functions G

G.1 Introduction

This appendix lists the attribute callback functions that can be implemented to replace or extend the normal attribute handling process in Parasolid.

Further information on the use and effects of attribute callback functions can be found in the following chapters of the Parasolid *Functional Description*:

- Chapter 95, “Attribute Definitions”
- Chapter 96, “Attributes”

G.2 Registering the attribute callback functions

The attribute callback functions must be registered with Parasolid by calling the PK function PK_ATTDEF_register_cb, at any time during a Parasolid session.

The functions can then be enabled and disabled during a modelling session by the PK function PK_ATTDEF_set_callback_flags.

G.3 Attribute callback functions that can be registered

This section lists the attribute callback functions that you can register during a Parasolid session. You do not need to register all of these callback functions; if there are any you do not wish to use, then set the relevant field in the `callbacks` structure passed to PK_ATTDEF_register_cb to `null`.

Note: In the following list, the function names given are purely nominal, as the functions are registered by the call to PK_ATTDEF_register_cb.

Function	Description	For more information
`split_fn`	Called after a split event has occurred. There are no attributes on the new entity.	PK_ATTDEF_split_callback_f_t
`merge_fn`	Called as a merge event is about to occur.	PK_ATTDEF_merge_callback_f_t
`delete_fn`	Called as a deletion event is about to occur.	PK_ATTDEF_delete_callback_f_t
`copy_fn`	Called after a copy event has occurred. There are no attributes on the new entity.	PK_ATTDEF_copy_callback_f_t

Function	Description	For more information
`transmit_fn`	Called at the start of a call to a PK function performing a transmit.	PK_ATTDEF_transmit_callback_f_t
`receive_fn`	Called at the end of a call to a PK function performing a receive.	PK_ATTDEF_receive_callback_f_t

Frustrum Tokens and Error Codes H

H.1 Introduction

This appendix lists all the tokens and error codes used by the Frustrum functions. These values are defined in the files ‘frustrum_ifails.h’ and frustrum_tokens.h’ in the Parasolid release area.

H.2 Ifails

FR_no_errors	0	operation was successful
FR_bad_name	1	bad file name
FR_not_found	2	file of given name does not exist
FR_already_exists	3	file of given name already exists
FR_end_of_file	4	file pointer is at end of file
FR_open_fail	10	unspecified open error
FR_disc_full	11	no space available to extend the file
FR_write_fail	12	unspecified write error
FR_read_fail	13	unspecified read error
FR_close_fail	14	unspecified close error
FR_memory_full	15	insufficient contiguous virtual memory
FR_bad_header	16	bad header found opening file for read
FR_rollmark_op_pass	20	rollmark operation within frustrum passed
FR_rollmark_op_fail	21	rollmark operation within frustrum failed
FR_unspecified	99	unspecified error

H.3 File guise tokens

FFCROL	1	rollback file
FFCSNP	2	snapshot file
FFCJNL	3	journal file
FFCXMT	4	transmit file (generated by Parasolid)
FFCXMO	5	transmit file (generated by Romulus)
FFCSCH	6	schema file
FFCLNC	7	licence file
FFCXMP	8	transmit file (partition)
FFCXMD	9	transmit file (delta)

FFCDBG	10	debug report file
FFCXMM	11	transmit file (mesh)

H.4 File format tokens

FFBNRY	1	binary
FFTEXT	2	text
FFAPPL	3	applio
FFXML	4	XML text

H.5 File open mode tokens

FFSKHD	1	skip header after opening file for read (usual case)
FFLVHD	2	leave header after opening file for read (fru tests)

H.6 File close mode tokens

FFNORM	1	normal: default action on file close
FFABOR	2	abort: delete the newly created file

H.7 Foreign geometry ifails

FGOPOK	0	Foreign geometry operation successful
FGOPFA	1	Foreign geometry operation failed
FGEVIN	2	Foreign geometry evaluation incomplete
FGPROP	3	Use default properties for foreign geometry
FGGEOM	4	Foreign geometry not found
FGDATA	5	Foreign geometry data retreive error
FGFILE	6	Foreign geometry data file error
FGRERR	7	Foreign geometry real data error
FGIERR	8	Foreign geometry integer data error

H.8 Foreign geometry operation codes

FGRECU	01	Retreive foreign curve geometry
FGRESU	02	Retreive foreign surface geometry
FGCOCU	11	Copy foreign curve geometry

FGCOSU	12	Copy foreign surface geometry
FGFRCU	21	Free foreign curve geometry
FGFRSU	22	Free foreign surface geometry
FGTXCU	31	Transmitting foreign curve geometry
FGTXSU	32	Transmitting foreign surface geometry

H.9 Foreign geometry evaluator codes

FGEVTR	01	Triangular evaluation matrix required
FGEVSQ	02	Square evaluation matrix required
FGPRBD	01	Geometry parametrisation is bounded
FGPRPE	02	Geometry parametrisation is periodic

H.10 Rollmark operation codes

FRROST	1	Rollback status
FRROSE	2	Set a roll mark
FRROMA	3	Roll to a mark
FRRODT	4	Rollmark is out of date
FRROON	1	Rollback status is ON
FRROFF	0	Rollback status is OFF

Go Tokens and Error Codes I

I.1 Introduction

This appendix lists all the tokens and error codes used by the Graphical Output functions.

I.2 Ifails

CONTIN	0	Continue: no errors
ABORT	-1011	Abort: return control to caller

I.3 Codes

CODCOM	1001	Segment complete
CODINC	1002	Segment incomplete
CODUNC	1003	Segment may or may not be complete
CODOVP	1004	Segment is outside all viewports
CODCVP	1005	Segment coincides with a viewport boundary
CODVIS	1006	Line segment is visible
CODINV	1007	Line segment is invisible
CODUNV	1008	Visibility of line segment is unknown
CODDRV	1009	Line segment is drafting
CODNSS	1013	Sharp mfin segment is not "smooth" but coincident with silhouette
CODSMO	1014	Segment is “smooth”
CODNSM	1015	Segment is not “smooth”
CODUNS	1016	Segment “smoothness” is unknown
CODSMS	1017	Segment is “smooth” but coincident with silhouette
CODINE	1018	Edge is internal
CODNIN	1019	Edge is not internal
CODINU	1020	Not known whether edge is internal
CODINS	1021	Edge is internal, coincides with silhouette
CODISH	1022	Line segment is invisible (hidden by own body occurrence)
CODIGN	1023	Edge lies on the boundary of an ignorable feature
CODESF	1024	Edge is coincident with sharp mfins
CODNES	1025	Edge is not coincident with sharp mfins
CODUES	1026	Edge coincidence with sharp mfins is unknown

I.4 Line types

L3TPSL	3001	Straight line
L3TPCI	3002	Partial circle
L3TPCC	3003	Complete circle
L3TPEL	3004	Partial ellipse
L3TPCE	3005	Complete ellipse
L3TPPY	3006	Poly-line
L3TPFV	3007	Facet vertices
L3TPFN	3008	Facet vertices plus surface normals
L3TPPC	3009	Non-rational B-curve
L3TPRC	3010	Rational B-curve
L3TPTS	3011	Facet strip vertices
L3TPTN	3012	Facet strip vertices plus surface normals
L3TPNC	3013	Non-rational B-curve (nurbs form)
L3TPRN	3014	Rational B-curve (nurbs form)
L3TPFP	3015	Facet vertices plus parameters
L3TPFI	3016	Facet vertices plus normals plus parameters
L3TPTP	3017	Facet strip vertices plus parameters
L3TPTI	3018	Facet strip vertices plus normal plus parameters
L3TPF1	3019	Facet vertices + norms + params + 1st derivs
L3TPF2	3020	Facet vertices + norms + params + all derivs
L3TPT1	3021	Facet strip vertices + norms + params + 1st derivs
L3TPT2	3022	Facet strip vertices + norms + params + all derivs
L3TPFC	3023	Facet vertices + norms +curvatures
L3TPFD	3024	Facet vertices + norms + params + curvatures
L3TPTC	3025	Facet strip vertices + norms + curvatures
L3TPTD	3026	Facet strip vertices + norms + params + curvatures
L3TPCS	3027	Complete sphere
L3TPCY	3028	Complete cylinder
L3TPCN	3029	Complete truncated cone

I.5 Segment types

SGTPBY	2003	Body (hierarchical segment)
SGTPED	2006	Edge
SGTPSI	2007	Silhouette line
SGTPPH	2008	Planar hatch-line
SGTPRH	2009	Radial hatch-line
SGTPRU	2010	Rib line (unfixed blend)

SGTPBB	2011	Blend-boundary line
SGTPPL	2012	Parametric hatch line
SGTPFT	2016	Facet
SGTPFA	2017	Face (hierarchical segment)
SGTPER	2018	Error segment
SGTPGC	2019	Curve geometry segment
SGTPGS	2020	Surface geometry segment
SGTPGB	2021	Surface boundary geometry segment
SGTPMF	2022	Mangled facet
SGTPVT	2023	Visibility transitions
SGTPTS	2024	Facet strip
SGTPVP	2025	Parametrised Visibility segment
SGTPIC	2026	Interference Curve
SGTPCL	2027	Clip Line
SGTPSF	2028	Sharp mfin line
SGTPFI	2029	Additional face faceted by incremental facetting
SGTPLI	2030	Lattice (hierarchical segment)
SGTPLB	2031	Lattice ball
SGTPLR	2032	Lattice rod
SGTPRE	2033	Region
SGTPSH	2034	Shell
SGTPFR	2035	Frame
SGTPVF	2036	Facet from volume rendering
SGTPVM	2037	Volume mangled facet

I.6 Error codes

ERNOGO	4001	Unspecified error
ERRUBB	4002	Rubber entity (no geometry attached)
ERSANG	4003	Surface angular tolerance too small
ERSDIS	4004	Surface distance tolerance too small
ERCANG	4005	Curve angular tolerance too small
ERCDIS	4006	Curve distance tolerance too small
ERCLEN	4007	Curve chord length tolerance too small
ERFWID	4008	Facet width tolerance too small
ERIFMF	4009	Incremental facetting: missing face
ERIFRE	4010	Incremental facetting: refinement required
ERIFER	4011	Incremental facetting: unspecified error
ERIFRM	4012	Incremental facetting: required missing face
ERIFRR	4013	Incremental facetting: required due to refinement

Legacy Functions J

J.1 Introduction

The Frustrum and Graphical Output functions documented in this appendix relate to functionality required only by routines defined in Parasolid’s previous interface, the Kernel Interface (KI).

Note: These functions are documented for legacy purposes only, and should not be implemented for new Parasolid applications.

J.2 Pixel drawing functions

These functions were required to process pixel data generated by the KI function RRPIXL.

Function	Description	For more information
`GOOPPX`	Open output of encoded pixel data	PK_GOOPPX_f_t
`GOPIXL`	Output encoded pixel data	PK_GOPIXL_f_t
`GOCLPX`	Close output of encoded pixel data	PK_GOCLPX_f_t

J.3 Rollback file handling functions

These functions were required to handle files generated by the KI session rollback functionality.

Function	Description	For more information
`FFOPRB`	Open rollback file	PK_FFOPRB_f_t
`FFSEEK`	Reset file pointer	PK_FFSEEK_f_t
`FFTELL`	Output file pointer	PK_FFTELL_f_t

Index In