OCTA

Integrated simulation system for soft materials

PLATFORM INTERFACE LIBRARY

libplatform

REFFERENCE

Version 2.1

OCTA User 's Group

DEC. 25 2002

Authors of the Manual

Yuzo Nishio The Japan Research Institute, Limited

Program Developers

Yuzo Nishio The Japan Research Institute, Limited
Masahiro Nishimoto The Japan Research Institute, Limited
Astuko Shono The Japan Research Institute, Limited

Acknowledgment

This work is supported by the national project, which has been entrusted to the Japan Chemical Innovation Institute (JCII) by the New Energy and Industrial Technology Development Organization (NEDO) under METI's Program for the Scientific Technology Development for Industries that Creates New Industries.

Copyright ©2000-2002 OCTA Licensing Committee All rights reserved.

Contents

1 What is libplatform **3**

2 Restrictions **5**

3 Main functions **7**

4 Package structure **9**

5 Package interface **13**
- 5.1 UDFhandle 13
- 5.2 Pfinterface 16
- 5.3 UDFstream 17
- 5.4 ErrorHandling 18
- 5.5 UDFObject 19
- 5.6 IObject 22
- 5.7 UDFManager 22
- 5.8 UObject 26
- 5.9 Location 27

6 Building application **29**

7 Automatic generation tool for interface classes **31**
- 7.1 What is makeinterface tool 31
- 7.2 Using makeinterface tool 31
- 7.3 Generation rule 32
  - 7.3.1 Control macro 32
  - 7.3.2 Global macro 33
  - 7.3.3 Class macro 34
  - 7.3.4 Attribute macro 34
- 7.4 Script file 37
- 7.5 Restrictions 38

8 Bugs in this release **39**

9 Error messages **41**
- 9.1 PFException 41
- 9.2 LocationException relation 41
- 9.3 UDFhandleException relation 41
- 9.4 UDFstreamException relation 41
- 9.5 UDFStreamException relation 42
- 9.6 UDFManagerException relation 42
- 9.7 UDFObjectException relation 42
- 9.8 UDF parser relation 42
- 9.9 an internal error 42

A Tutorial: makeinterface 43
A.1 Overview of makeinterface tool . . . 43
A.2 Automatic generation of an simple class definition . . . 43
A.2.1 Making a class UDF information . . . 43
A.2.2 Generating a class by makeinterface tool . . . 44
A.2.3 Check IUdfInformation.h . . . 45
A.2.4 Check udfinfo_test.cpp. . . . 46
A.2.5 Build and execute the generated program . . . 46
A.3 Generating UObject interface . . . 47
A.3.1 Check uobject_template.txt . . . 47
A.3.2 Check uobject_test.cpp . . . 48
A.3.3 Build and execute the generated program . . . 50
A.3.4 UDFManager class methods . . . 50
A.4 generation of an IObject interface . . . 50
A.4.1 Check template.txt . . . 50
A.4.2 Check script.txt . . . 51
A.4.3 Check iobject_test.cpp . . . 51
A.4.4 Build and execute our program . . . 53
A.4.5 UDFObject class methods . . . 53
A.5 What now . . . 53

List of Figures

3.1 Use-case of OCTA system 8

4.1 Relations between packages 11

List of Tables

5.1 UDFhandle method list . . . 13
5.2 Pfinterface package fuction list . . . 16
5.3 UDFstream method list . . . 17
5.4 example of UDF symbol pattern . . . 18
5.5 UDFObject method list . . . 19
5.6 UDFManager method list . . . 22
5.7 example of UDFManager symbol assignation . . . 26
5.8 INDEX method list . . . 27
5.9 Location method list . . . 27

7.1 makeinterface macro rule list . . . 32
7.2 makeinterface macro definition list (1) . . . 33
7.3 makeinterface macro definition list (2) . . . 34
7.4 makeinterface macro definition list (3) . . . 34
7.5 Example template file for makeinterface . . . 35
7.6 makeinterface script list . . . 37
7.7 Example script file for makeinterface . . . 37

A.1 Example script for simple class ganaration . . . 44
A.2 Example of ganerated simple class by makeinterface . . . 45
A.3 Example of using ganerated class . . . 46
A.4 Example of using UDFManager interface classes . . . 48
A.5 Example of using UDFObject interface classes . . . 51

Chapter 1

What is libplatform

"libplatform" is a library which is packed the interface functions of UDF file and analytic-engine I/O, and it is used for it, linking it with an analytic engine. The input/output interface is encapsulated using the function of C++ language, and an analytic-engine designer can prepare an interface class based on a UDF definition, can read UDF data into the object in stream form, and can write out similarly.

After V2.0 or later, in order to mainly realize improvement in the speed, it is the library for stand-alones which does not use a server machine. In connection with this, the UDF file in which an analytic engine carries out I/O turned into a text file which exists locally. Since the cache of the opened contents of a UDF file is carried out on the back memory by which syntactic analysis was carried out, they have realized the high-speed data access.

The function which controls the analytic engine which works by the remote server from GOURMET by V3.0, and supervises the operation situation of an analytic engine was added.

Moreover, the write-in function of the global data which are the new functions of UDF was supported.

As an I/O method of UDF data, two kinds of interfaces, a UDFManager interface and an IObject interface, are offered.

The late binding type which solves a data name at the time of execution is used for a UDFManager interface, it performs direct I/O to a cache memory, and offers the operation equivalent to the Python script used by GOURMET. For this reason, it is suitable for exploitation with the tools treating the UDF data of the admission to UDF, or easy structure.

The IObject interface adopt the early binding type which generates automatically and uses the binding-head class to C++, perform I/O in stream form, and are suitable for exploitation by the analytic engine treating the UDF data of complicated structure.

Chapter 2

Restrictions

- Direct interfaces for C and FORTRAN are not suported. You can make the conversion filter by file convert tool. Please reffer "GOURMET Operations Manual".
- It is premising using C++ stream to read/write the UDF data.
- The interface with the data structure in C++ language is premised. Specifically, the method which maps a Component type data structure in an interface class is supported.
- The environment where it uses for compile link is Microsoft Visual C++. (r) (later than 6.0 SP3) It is referred to as Linux (2.2.9 or more kernel(s)) and Cygnus (later than b20.1). 2.95.2 or more gcc(s), 2.1.2 or more glibc(s), and 2.5 or more Flex(es) are required for the compile link in Linux and Cygnus.

Chapter 3

Main functions

The main functions of the OCTA system is shown in Figure 3.1. The ranges described by this specification are the interface portions of an engine and a system, and are the portions shown within the limit.

The definition of each function is shown below.

- **UDF header information acquisition and updating** The header information of UDF specified by engine program is acquired or updated.
- **input file reading**

  The value of a UDF symbol with engine program differential out of the record of specified UDF is read.
- **output-file new creation**

  engine program specifies the existing structure definition file, and newly creates UDF.
- **output-file writing**

  engine program adds or updates the value of the UDF symbol specified into the record of specified UDF.
- **engine control information acquisition**

  engine program acquires the engine control information which a user directs.

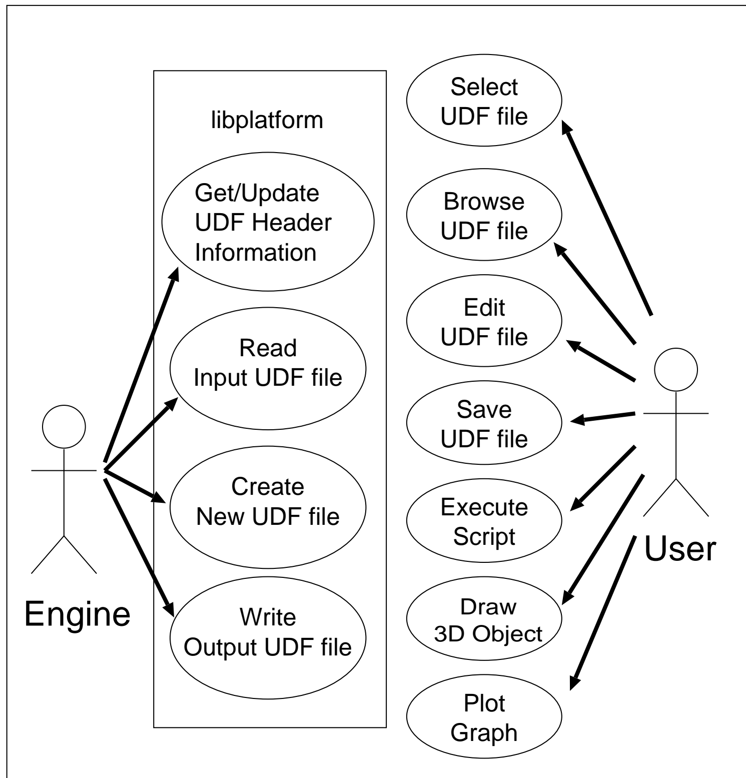

Figure 3.1: Use-case of OCTA system

Chapter 4

Package structure

A platform library consists of the following packages.

- **STL**

  The par C++ template library offered from compiler vender. The class currently used is as follows.

  string, iostream, fstream, strstream, list, vector, map, and the algorithm

  platform library, the STL library is used with each package.

- **UDFhandle**

  The Basic Rate Interface for reading and engine writing UDF data. An interface definition is udfhandle.h.

- **Pfinterface**

  The interface for concealing UDFhandle. It is preparing for the correspondence extension to prospective C and a FORTRAN engine.

  An interface definition is pfinterface.h.

- **UDFstream**

  The package for concealing the data-exchange type between UDF and an engine as a stream of C++ language.

  An interface definition is udfstream.h.

- **ErrorHandling**

  The package which performs exception handling.

  An interface definition is pfexception.h .

- **UDFObject**

  The package of the Basic Rate Interface class definition for mapping the data structure of UDF in the class of C++ language.

  An interface definition is udfobject.h.

- **IObject**

  The package which realizes the UDF interface of engine. UDFObject is inherited and the interface object used with each engine is mounted.

  an interface definition – ICognacin.h and IMuffin_solid.h Etc.

  The tool "makeinterface" which analyzes a UDF file and generates this package automatically is offered. Refer to Chapter 7 "an interface class automatic generation tool" about the method of this tool. The simple description for admission is summarized in Chapter 11 "makeinterface Tutorial."

- **UDFManager**

  The package which realizes a UDF interface by the I/O type (late binding type) different from UDFObject and IObject. This package is for using UDF simple for the shallow user of experience of C++ programming. There is no want that the user of UDFManager knows the contents of other packages.

  The simple description for admission is summarized in another manual "UDF/PF Tutorial." In addition, since the formation of an across the board library is difficult for the source code created using this package, it recommends limiting to exploitation by slight programs, such as tools.

  An interface definition is udfmanager.h.

- **UObject**

  The package which realizes the UDF interface for treating structured-type data simple using UDFManager.

  an interface definition – UCognacin.h and UMuffin_solid.h Etc.

  The tool "makeinterface" which analyzes a UDF file and generates this package automatically is offered. Refer to Chapter 7 "an interface class automatic generation tool" about the method of this tool. The simple description for admission is summarized in Chapter 11 "makeinterface Tutorial."

- **Location**

  The utilities package for specifying a UDF symbol simple using UDFManager.

  An interface definition and udflocation.h

A package related figure is shown in drawing Fig. 4. The dotted-line arrow shows the dependency of a package. The direction of an arrow is dependency (including the file).

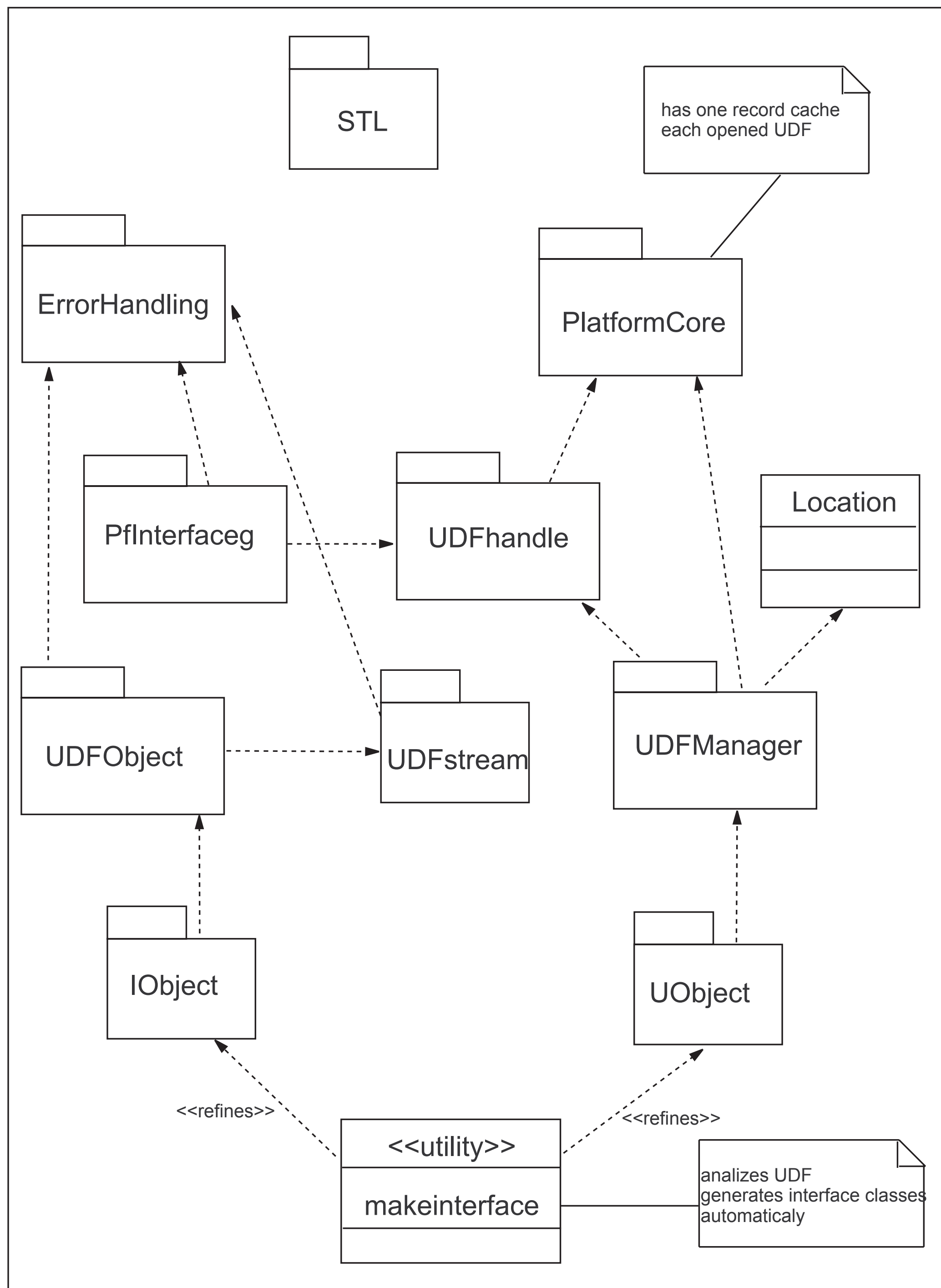

Figure 4.1: Relations between packages

Chapter 5

Package interface

The interface of each package is shown below.

5.1 UDFhandle

The Basic Rate Interface for reading and an engine writing UDF data. Usually, using directly is also possible although used from a PFinterface package.

An interface definition is udfhandle.h.

A method signature and function list is shown in Table 5.1.

Table 5.1: UDFhandle method list

UDFhandle();
Default constructor

UDFhandle(const string &udfname, const int recno =COMMON_RECORD);
constructor. An instance is built and open() is called by the UDF file name and record number which were specified.

UDFhandle (const string &udfname, const string &recname);
constructor. An instance is built and open() is called by the UDF file name and record name which were specified.

ŨDFhandle();
destructor. If the number of times of reference becomes zero, a UDF cache will be deleted and UDFhandle will be rejected to a mint condition.

bool open (const string &udfname, const int recno=COMMON_RECORD);
A UDF cache is built by the UDF file name and record number of which assignation was done.

bool open (const string &udfname, const string &recname);
A UDF cache is built with the UDF file name and record label of which assignation was done.

void close();
UDF cache is deleted and UDFhandle is rejected to a mint condition. In the case of the handle for a UDF output, the contents of a UDF cache are outputted to a UDF file.

int removeRefference();
The decrement of the number of times of reference is carried out.

bool isOpen();
true will be returned if UDF cache is built.

bool inRecord();
true will be returned if it becomes during an output about record data.

void setRecordName (const string &recname);
record label is set up.

int getCurrentRecord();
current-record number is returned.

int getTotalRecord();
All the record counts of present are returned.

const char* newRecord(const char *prefix=NULL);
A record is created newly and a record label is returned. If a name is specified, a # record number will be added and a record label will be set up.

void endRecord();
record data output is ended.

bool getIdList (const string &class_name, vector<int> &list) const;
The list of ID data of a UDF class name of which assignation was done is created. true will be returned if it succeeds.

const char* getLocation (const string &class_name, int id);
A UDF location character string with the UDF class name of which assignation was done, and ID data is returned.

bool getKeyList(const string &class_name, vector<string> &list) const;
The list of Key data of a UDF class name of which assignation was done is created. true will be returned if it succeeds.

const char* getLocation (const string &class_name, const string &key);
A UDF location character string with the UDF class name of which assignation was done, and Key data is returned.

getProjectName() const;
ProjectName of UDF header information is returned.

const string& getIOType() const;
IOType of UDF header information is returned.

const string& getEngineName() const;
EngineType of UDF header information is returned.

const string& getEngineVersion() const;
EngineVersion of UDF header information is returned.

const string& getComment() const;
Comment of UDF header information is returned.

const string& getAction() const;
Action FilePath of UDF header information is returned.

bool isCompatible(const string& version ="", const string& engine ="") const;
It does as EngineVersion of UDF header information, and if there is no version of which assignation was done, it will return false. false will be returned, if engine is specified and EngineVersion is not the same.

void setProjectName(const string& text);
ProjectName of UDF header information is set up.

void setIOType(const string& text);
IOType of UDF header information is set up.

void setEngineName (const string& text);
EngineType of UDF header information is set up.

void setEngineVersion (const string& text);
EngineVersion of UDF header information is set up.

void setComment (const string& text);
Comment of UDF header information is set up.

void setAction (const string& text);
Action FilePath of UDF header information is set up.

5.2 Pfinterface

The interface for concealing UDFhandle. The Pfinterface package serves as a set of functions, cooperates with UDFstream, and outputs and inputs UDF data so that it can use from a C interface.

An interface definition is pfinterface.h.

A method signature and function list is shown in Table 5.2.

Table 5.2: Pfinterface package fuction list

UDFhandle* OpenUDF (const char *udfname, int record_no=COMMON_RECORD);
assignation The UDF file carried out, A record number is opened and UDFhandle is returned.

UDFhandle* CreateUDF (const char *udfname, const char *filename, bool def_copy=true);
A UDF file with the structure of the UDF definition file of which assignation was done is newly created, and UDFhandle is returned. When def_copy is false, the UDF file which carries out the include assignation of the structure definition is newly created.

UDFhandle* CreateUDF (const char *udfname, const char **filenames, int nfiles, bool def_copy=true);

A UDF file with the structure of the UDF definition file which carried out the number assignation (nfiles) in array (filenames) is newly created, and UDFhandle is returned. When def_copy is false, the UDF file which carries out the include assignation of the structure definition is newly created.

void CloseUDF (UDFhandle *fd);
UDFhandle of which assignation was done is closed. Nothing is performed when this UDFhandle is referred to by others.

void AppendUDF (UDFhandle *fd, int record_no=COMMON_RECORD);
A record is added to UDFhandle of which assignation was done. The record number to add must be the positive integer which increases from 0 sequentially.

void SetStepName (UDFhandle *fd, const char *record_name);
The record label of UDFhandle of which assignation was done is set up. All the record counts of UDFhandle of which the size_t GetTotalStepNumber (UDFhandle *fd) assignation was done are returned. An engine control file path is passed to the engine started from void SetPFControl (const char *statusFile) GOURMET as a command line parameter. The passed engine control file is set up. Control information acquisition function after this functional call PFControl() It can be used. Generally, it is used with the main function of an analytic engine.

E_ENGINE_STATUS_TYPE PFControl (void);
engine control information is acquired. Processing to which it is used and an engine generally corresponds according to the acquired information within the analysis loop of an analytic engine will be performed. E_ENGINE_STATUS returned is e_ENGINE_RUN(execution) / e_ENGINE_RESUME(resumption) / e_ENGINE_STOP(discontinuation) / e_ENGINE_ERROR (engine control error). It is either. The sleep of the e_ENGINE_SUSPEND (halt) is carried out until it is not returned and other control assignations are performed within this function.

5.3 UDFstream

The package for concealing the data-exchange type between UDF and an engine as a stream of C++ language. It cooperates with UDFhandle and the I/O stream of UDF data is realized.

A method signature and function list is shown in Table 5.3.

Table 5.3: UDFstream method list

UDFistream::UDFistream();
: default constructor

UDFistream:: UDFistream(UDFhandle* fd, const char *udfsymbol=NULL);
: constructor. The input stream of specified UDFhandle is built. If the UDF symbol pattern is specified, an input stream is opened simultaneously.

UDFistream::ŨDFistream();
: destructor

UDFistream::open (const char *udfsymbol);
: assignation was done is opened

unsigned int UDFistream::number ();
: returns the number of UDF objects on the input stream. This method will be used in order to judge whether you read the input stream or not.

void UDFistream::close();
: close input stream. A UDF stream is reusable. That is, a different UDF symbol pattern can be specified and opened to the same UDFistream.

UDFostream::UDFostream();
: default constructor

UDFostream::UDFostream (UDFhandle* fd, const char *udfsymbol=NULL);
: constructor. The output stream of specified UDFhandle is built. If the UDF symbol is specified, an output stream is opened simultaneously. The UDF symbol to specify must be a top object.

UDFostream::ŨDFostream();
: destructor

int UDFostream::open(const char *udfsymbol=NULL);
: The output stream of the UDF symbol specified is opened. The UDF symbol to specify must be a top object.

void UDFostream::close();
: output stream is closed. A UDF stream is reusable. That is, a different UDF symbol pattern can be specified and opened to the same UDFostream.

Note:

A UDF symbol pattern is the character string which specifies a set of the "variable name" in UDF specification, and can specify the data host in UDF by the unique name. It specifies using this character string and writing the differential data host in UDF as a stream. If [] of a tail is not specified when a UDF variable is an array, it can specify inputting the array object itself.

The example of a UDF symbol pattern is shown in Table 5.3.

UDF symbol pattern	Specified UDF data stream	the number of data
Zero length string	All objects in UDF record	number of top object
molecules	whole of molecules Object	1
molecules.bond[]	All elements of molecules.bond[] array	number of elements
molecules.bond	array that has all elements of molecules.bond[]	1
molecules.bond[].atom1.posx	atom1.posx Object in the all elements of molecules.bond[] array	number of elements

Table 5.4: example of UDF symbol pattern

5.4 ErrorHandling

The package which performs exception handling. An interface definition is pfexception.h.

class ExceptionBase is a abstract class for encapsulating exception handling. In this release, three kinds of following exception-handling classes are offered.

class PFException;
The error of a server module is encapsulated.

class UDFStreamException;
The error about UDFstream is encapsulated.

class UDFObjectException;
The error about UDFObject is encapsulated.

Each class has two kinds of following diagnostic methods.

ERROR_CODE WhatsWrong();
The interior error code is returned.

virtual void Explain(DiagOutput & diag);
error information is outputted to a diagnostic-information output place.

Moreover, the following function are defined in order to carry out across the board exploitation with a C interface.

const char *GetErrorMessage(ERROR_CODE code);
The diagnostic information corresponding to the error code is returned.

class DiagOutput is a abstract class for outputting diagnostic information. In this release, three kinds of following diagnostic-information output class is offered.

class DiagOutConsole;
Diagnostic information is outputted to standard output.

class DiagOutFile;
Diagnostic information is outputted to a log file.

each class has the message output method of the following in consideration of the output place.

virtual void DisplayMsg(const char *msg, DiagLevel level);

5.5 UDFObject

The package of the Basic Rate Interface class definition for mapping the data structure of UDF in the class of C++ language. An interface definition is udfobject.h.

A method signature and function list is shown in Table 5.5.

Table 5.5: UDFObject method list

UDFObject(); default constructor

UDFObject(const UDFObject& other);
copy constructor

ŨDFObject();
destructor

virtual const char *GetName();
UDF variable name acquisition. This method will be overrided by the inherited class.

virtual void _Input(istream& is);
stream input stub. This method will be overrided by the inherited class.

virtual void _Output(ostream& os)
stream I/O stub. This method will be overrided by the inherited class.

UDFshort::UDFshort
binded to C++ foundations type short

UDFint::UDFint
binded to C++ foundations type int

UDFlong::UDFlong
binded to C++ foundations type long

UDFfloat::UDFfloat
binded to C++ foundations type float

UDFsingle::UDFsingle
binded to C++ foundations type float

UDFdouble::UDFdouble
binded to C++ foundations type double

UDFstring::UDFstring
string type in UDF

UDFstring::GetValue()
value acquisition

operator UDFstring::string()
Type-conversion operator to a C++ string object

void UDFstring::SetValue(string val)
Value setting

template<class T>class UDFarray::UDFarray()
UDFarray type template

size_t UDFarray::size()
STL::<list> same name method

size_t UDFarray::max_size()
STL::<list> same name method

void UDFarray::push_back(T &x)
STL::<list> same name method

void UDFarray::clear()
STL::<list> same name method

void UDFarray::pop_back()
STL::<list> same name method

bool UDFarray::empty()
STL::<list> same name method

void UDFarray::reserve(size_t n)
STL::<vector> same name method

size_t UDFarray::capacity()
STL::<vector> same name method

T UDFarray::operator[(size_t pos)]
STL::<vector> same name method

const T &UDFarray::operator[(size_t pos)]
STL::<vector> same name method

vector<T>::iterator UDFarray::begin()
STL::<vector> same name method

vector<T>::iterator UDFarray::end()
STL::<vector> same name method

vector<T>::iterator UDFarray::rbegin()
STL::<vector> same name method

vector<T>::iterator UDFarray::rend()
STL::<vector> same name method

void UDFarray::resize(size_t n, T x = T())
STL::<vector> same name method

vector<T>::iterator UDFarray::erase(vector<T>::iterator)
STL::<vector> same name method

vector<T>::iterator UDFarray::erase(vector<T>::iterator, vector<T>::iterator)
STL::<vector> same name method

void UDFarray::assign(vector<T>:: const_iterator, vector<T>::const_iterator)
STL::<vector> same name method

void UDFarray::assign(size_t n, const T& x = T())
STL::<vector> same name method

vector<T>::iterator UDFarray::insert(vector<T>::iterator, const T& x = T())
STL::<vector> same name method

void UDFarray::insert(vector<T>::iterator it, size_t n, const T& x)
STL::<vector> same name method

void UDFarray::insert(vector<T>:: iterator, vector<T>::const_iterator, vector<T>::const_iterator)

STL::<vector> same name method

bool operator==(const UDFObject& x, const UDFObject& y)
equivalence judging operator

bool operator<(const UDFObject& x, const UDFObject& y)
size judging operator

istream& operator >>(istream& is, UDFObject& object)
text-input stream operator

ostream& operator <<(ostream& os, UDFObject& object)
text output stream operator

UDFObjectException (int err, const char *info = NULL)
constructor. UDFObjectException of an assignation error number and additional information is generated.

void UDFObjectException::Explain(DiagOutput & diag);
A message is displayed on DiagOutput.

const char * UDFObjectException::ErrorMessage(int err);
The message character string of assignation error number is returned.

5.6 IObject

The package which realizes the UDF interface of an engine. UDFObject is inherited and the interface object used with each engine is mounted.

An interface definition uses the interface name to which I, such as ICognacin.h and IMuffin_solid.h, was attached to the head. Although an interface name is usually made the same as a UDF name, a user can also decide.

The tool ”makeinterface” which analyzes a UDF file and generates this package automatically is offered. Refer to Chapter 7 ”an interface class automatic generation tool” and Chapter 11 ”makeinterface Tutorial” about this tool.

5.7 UDFManager

The package which realizes a UDF interface by different from UDFObject and IObject. There is no want that the user of UDFManager knows the contents of other packages. The simple description for admission is summarized in another manual ”UDF/PF Tutorial.”

An interface definition is udfmanager.h.

A method signature and function list is shown in Table 5.6.

Table 5.6: UDFManager method list

UDFManager(const string &udfname,int mode = READ)
constructor. An instance is built and the UDF file which specified is opened. READ or WRITE is specified by mode.

UDFManager(const string &udfname, const string &deffilename, bool def_copy = true)
constructor. An instance is built and a UDF file is newly created using the UDF definition file which specified. When def_copy is false, it becomes the assignation which includes a definition file.

ŨDFManager()
destructor

const string& getProjectName() const;
ProjectName of UDF header information is returned.

const string& getIOType() const;
IOType of UDF header information is returned.

const string& getEngineName() const;
EngineType of UDF header information is returned.

const string& getEngineVersion() const;
EngineVersion of UDF header information is returned.

const string& getComment() const;
Comment of UDF header information is returned.

const string& getAction() const;
Action FilePath of UDF header information is returned.

bool isCompatible (const string& version = ””, const string& engine = ””) const;
It does as EngineVersion of UDF header information, and if there is no version of which assignation was done, it will return false. false will be returned, if engine is specified and EngineVersion is not the same.

void setProjectName(const string& text);
ProjectName of UDF header information is set up.

void setIOType (const string& text);
IOType of UDF header information is set up.

void setEngineName (const string& text);
EngineType of UDF header information is set up.

void setEngineVersion (const string& text);
EngineVersion of UDF header information is set up.

void setComment (const string& text);
Comment of UDF header information is set up.

void setAction (const string& text);
Action FilePath of UDF header information is set up.

size_t totalRecord()
record count is acquired.

bool jump(size_t record_no)
It moves to a assignation record.

bool jump (const string &record_name)
It moves to a assignation record label.

bool nextRecord()
It moves to a next record.

const string newRecord (const string prefix)
a record is created newly and a record label is returned If a name is specified, a # record number will be added and a record label will be set up.

bool write ()
save to the UDF file which opens all record data.

bool write (const string &udfname)
save as the UDF file which specified all record data.

bool seek (const string &location)
current position is moved to a designating-symbol position.

bool seek (const Location &location)
current position is moved to an assignation location.

const string tell()
present symbol position is acquired.

string type()
The symbol type of the current position is acquired.

bool next()
It moves to next data.

size_t size()
The number of array elements of the symbol of the current position is acquired.

size_t size (const string &location)
The number of array elements of the symbol of the current position is acquired.

size_t size (const string &location, const INDEX & index=INDEX())
The number of array elements of a designating symbol and INDEX is acquired.

bool get (const Location &location, UDF_TYPE &value)
The data of a location are acquired.

bool get (const string &location, UDF_TYPE &value, const INDEX & index=INDEX())
The data of a designating symbol and INDEX are acquired.

bool getArray (const Location &location, vector< UDF_TYPE > &array)
The data array of a assignation location is acquired.

bool getArray(const string&location,vector<UDF_TYPE>&array, const INDEX&index=INDEX())

The data array of a designating symbol and INDEX is acquired.

bool get (const string &location, UDF_TYPE *value, const INDEX & index=INDEX())
The data of a designating symbol and INDEX are acquired.

short shortValue (const string &location)
designating symbol are acquired.

int intValue (const string &location)
The data of a designating symbol are acquired.

long longValue (const string &location)
The data of a designating symbol are acquired

float floatValue (const string &location)
The data of a designating symbol are acquired

double doubleValue (const string &location)
The data of a designating symbol are acquired

string stringValue (const string &location)
The data of a designating symbol are acquired

vector<short> & shortArray (const string &location)
The array data of a designating symbol are acquired

vector<int> & intArray (const string &location)
The array data of a designating symbol are acquired

vector<long> & longArray (const string &location)
The array data of a designating symbol are acquired

vector<float> & floatArray (const string &location)
The array data of a designating symbol are acquired

vector<double> & doubleArray (const string &location)
The array data of a designating symbol are acquired

vector<string> & stringArray (const string &location)
The array data of a designating symbol are acquired

int i(const string &location)
Alias of intValue

double d (const string &location)
Alias of doubleValue

string s(const string &location)
Alias of stringValue

vector<int> & iarray(const string &location)
Alias of intArray

vector<double> & darray(const string &location)
Alias of doubleArray

vector<string> & sarray(const string &location)
Alias of stringArray

bool put (const Location &location, UDF_TYPE &value)
assignation location are set up.

bool put (const string &location, UDF_TYPE &value, const INDEX& index=INDEX())
The data of a designating symbol and INDEX are set up.

bool putArray (const Location &location, vector<UDF_TYPE>&array)
The data array of a assignation location is set up.

bool putArray(const string&location,vector<UDF_TYPE>&array,const INDEX&index=INDEX())

The data array of a designating symbol and INDEX is set up.

vector<int> getIdList (const string &class_name)
The list of ID data of a UDF class name of which the function assignation was done is acquired.

const Location &getLocation (const string &class_name, int id)
The location of a symbol with the UDF class name of which the assignation was done, and ID data is acquired.

vector<string> getKeyList (const string &class_name)
The list of Key data of a UDF class name of which the assignation was done is acquired.

const Location &getLocation (const string &class_name, const string &key)
The location of a symbol with the UDF class name Key data of which the assignation was done is acquired.

UDFManager::UDFManagerException
a certain error within UDFManager

UDFManager::ObjectNotFoundException
applicable data are not found

UDFManager::SymbolNotFoundException
a designating symbol is not found

UDFManager::SymbolNotDeterminedException
designating symbol cannot be determined:

UDFManager::KeyNotFoundException
designating key is not found

UDFManager::IndexNotFoundException
index is not found

UDFManager::ParseException
UDF Perth error

UDFManager::FileIOException
IOerror occured in assignation UDF file (open error is also included)

UDFManager::SystemException
Undefined error occured in a library

UDFManager::NotImplementedException
called the function which is not implemanted

Note: When there are not especially explanation, the method which returns bool will return false, if an operation is successful, and it will true and will fail. UDF_TYPE shows short, int, long, float, double, or string.

The notation of a "symbol" expresses a UDF variable character string, and can use the character string which specifies a UDF variable strictly as shown in Table 5.7. When it expresses an array, it is cautious of "[]" being required. It is a helper class for INDEX specifying the subscript of an array in repeat processing, and the constructor to five dimensions is prepared.

Symbol	INDEX	Contents
molecules.bond[]	none	molecules.bond Array
molecules.bond[0]	none	first element of molecules.bond Array
molecules.bond[]	INDEX(0)	same as the above
molecules.bond[1].atom[0].posx	none	first posx object in second element Array of molecules.bond Array
molecules.bond[].atom[].posx	INDEX(1,0)	same as the above

Table 5.7: example of UDFManager symbol assignation

A "location" is the instance of the helper class Location which mounts above UDF array character string, INDEX, and can specify a UDF variable strictly independently. Refer to 5.9 Location about the method of a Location class.

5.8 UObject

The package which realizes the UDF interface for treating structured-type data simple using UDFManager.

an interface definition – UCognacin.h and UMuffin_solid.h etc. – U uses the interface name attached to the head Although an interface name is usually made the same as a UDF name, a user can also decide.

The tool "makeinterface" which analyzes a UDF file and generates this package automatically is offered. Refer to Chapter 7 "an interface class automatic generation tool" about the method of this tool. The simple description for admission is summarized in Chapter 11 "makeinterface Tutorial."

5.9 Location

The utilities package for specifying a UDF symbol simple using UDFManager. An interface definition and udflocation.h

INDEX class: The constructor of the subscript information for an array data access
A method signature and function list is shown in Table 5.8.

Table 5.8: INDEX method list

INDEX()
default constructor

INDEX(const UINT idx1)
1 demensional constructor

INDEX(const UINT idx1, const UINT idx2)
2 demensional constructor

INDEX(const UINT idx1, const UINT idx2, const UINT idx3))
3 demensional constructor

INDEX(const UINT idx1, const UINT idx2, const UINT idx3, const UINT idx4))
4 demensional constructor

INDEX(const UINT idx1,const UINT idx2,const UINT idx3,const UINT idx4,const UINT idx5))

5 demensional constructor

A method signature and function list is shown in Table 5.9.

Table 5.9: Location method list

Location()
default constructor

Location (const Location & loc)
copy constructor

Location (const string &path)
constructor.

Location (const char *path)
constructor which analyzes the path character string containing the subscript.

Location (const string &path, const INDEX &index)
constructor which analyzes the path character string containing the subscript. analyze the path character string and an assignation index

const string & str()
return path character string

const Location & seek(const string &path)
it moves to the assignation location

const Location & root()
which moves to a top location

const Location & up()
moves to upper location of the hierarchy

const Location & down ()
moves to lower location of the hierarchy

const Location & seek(const string &path)
moves to the location with path

const Location & down(const string &element)
moves to the location of the element

const Location sub(const string &element) const
returns the location of the element without move the current location

const Location & prev()
moves to the previous element of the current array

const Location & next()
moves to the next element of the current array

Location array() const
moves to the current array

const string & getPath()
returns the symbol path

const INDEX & getIndex()
returns the index

Location::LocationException
illegal usage of Location

Chapter 6

Building application

The environment where it uses for a compile link is required.

- Win32
  Microsoft Visual C++(R) 6.0 SP3 or later
- Linux, Cygwin
  Linux (kernel 2.2.9 or lator) and Cygnus (b20.1 or later)
  gcc 2.95.2, glibc 2.1.2, and Flex 2.5

The include file interface library-related [required for application creation] is summarized to the directory $PF_FILES/include/ ,and a library $PF_FILES/lib/$TARGET_ENV/ is prepared.

In this release, the compile macro of environmental dependence is not using it.

- Win32 version
  The example for VC++ construction of a project file is in %PF_FILES%/tutorial/makeinterface/
- Linux, Cygwin version
  The example of Makefile is in $PF_FILES/tutorial/makeinterface/

Chapter 7

Automatic generation tool for interface classes

7.1 What is makeinterface tool

"makeinterface" is a tool which generates the interface class definition for analyzing the definition information on the specified UDF file, and outputting and inputting the same UDF object as the contents of a definition.

If the top object (data with which the name is described in the UDF data section) of UDF is specified by using the class definition generated with this tool, all the structures of the object can be outputted and inputted at once certainly.

"makeinterface" can generate the header file of both of an IObject interface class and a UObject interface by specifying the template file by which par offer is made.

Moreover, it is the script file which specifies the generation method, and it is possible to select the UDF object treated as a binding head in a user definition class or an KEY/ID class.

7.2 Using makeinterface tool

- **Usage:**

```
makeinterface -Dudfdef_file [-Ninterface_name ] [-Itemplate_file]
            [-Ooutput_file] [-Cclass_prefix] [-Aattribute_prefix]
            [-Sscript_file]
```

- **Parameters:**

udfdef_file:
The information of the def section of the UDF file
by which the UDF definition file name assignation was carried out is analyzed, and an interface class is created. If the UDF definition file to specify may also contain **include** and does not exist in a current directory in that case, it looks for the directory specified by environment variable **UDF_DEF_PATH**.

interface_name:
Interface name
include identification Macro name – it is used This is usually used for include inspection of a header file. A default is letter except the extension of class_prefix+ (udfdef_file).

template_file:
The macro portion of the template file by which the template file pathname
assignation was carried out is replaced, and an interface class is created. The contents of generation can be changed by editing template_file. Refer to 7.3 "a generation template file" for details. If a default does not have template.txttemplate_file in an execution-time current directory, it will look for a directory with a makeinterface module.

output_file:
The header file pathname
default to output is interface_name+".h".

class_prefix:
The prefix of class
default attached to the interface class name to generate generates an IObject interface by "I." An assignation of "U" generates a UObject interface.

attribute_prefix:
The prefix of attribute
default attached to the member attribute name to generate makes, and is.

script_file:
script_file in which generation rules are specified
The generation rule can be changed by editing generation method specification-file script_file. Refer to 7.4 "a generation script file" for details. If a default does not have script.txt script_file in an execution-time current directory, it will look for a directory with a makeinterface module.

7.3 Generation rule

The generation rules can be changed by editing template_file.

In template_file, it is transposed to the UDF name with which the contents by which macro definition was carried out were analyzed, and is repeatedly outputted according to a macro rule.

7.3.1 Control macro

A macro rule is the character string surrounded by $, it has the role which performs a control assignation and an output is not performed.

You can use the rule macros showing in Table 7.1.

Table 7.1: makeinterface macro rule list

$CLASS_TEMPLATE_BEGIN$
Opening of class definition rules.

$CLASS_TEMPLATE_END$
Closure of class definition rules. The contents from the following row to the last row of are repeated only for the tale of generation classes.

$ATTRIBUTE_TEMPLATE_BEGIN$
Opening of attribute definition rules for each class.

$ATTRIBUTE_TEMPLATE_END$
The contents from the following row to the last row of are repeated only for the number of attributes of a class during generation.

IF_BASE_TYPE
Opening of an attribute definition of each of a closure conditional-branching rule. When the classification of an attribute is a foundations type during generation, the contents from the following row to the last row of following conditional-branching rule or IF_TYPE_END are generated.

IF_USER_TYPE
Opening of a conditional-branching rule. When the classification of an attribute is a user definition class during generation, the contents from the following row to the last row of following conditional-branching rule or IF_TYPE_END are generated.

IF_ARRAY_TYPE
Opening of a conditional-branching rule. When the classification of an attribute is a foundations type array during generation, the contents from the following row to the last row of following conditional-branching rule or IF_TYPE_END are generated.

$IF_USER_ARRAY_TYPE$

Opening of a conditional-branching rule. When the classification of an attribute is a user definition class array during generation, the contents from the following row to the last row of following conditional-branching rule or IF_TYPE_END are generated.

IF_MAP_TYPE

Opening of a conditional-branching rule. The contents from the following row to the last row of IF_TYPE_END are generated for the classification of a class the case of KEY or ID during generation.

IF_TYPE_END

Closure of a conditional-branching rule

Macro definition is the character string surrounded by %, and the character string replaced according to the contents of processing is outputted. Moreover, the macro definition which finishes it as _LIST is a row macro, and only the line count corresponding to a list in from the head of the sentence to the end of the sentence is outputted. Therefore, a row macro is not made to a nest.

7.3.2 Global macro

The macro definition of Table 7.2 can be used anywhere in a template file.

Table 7.2: makeinterface macro definition list (1)

%GENERATED_DATE%

The date and time which performed automatic generation.

%GENERATED_VERSION%

The version of makeinterface which performed automatic generation.

%INTERFACE_NAME%

interface name.

%INCLUDE_LIST%

The list of files which carried out the include assignation with the script. It uses for an include-file assignation (include"xxxx") etc.

%DEFINITION%

%CLASS_LIST%

The list of all classes of which %UDF definition file name generation is done. It uses for a prototype declaration (class xxxx;) etc.

%KEY_CLASS_LIST%

The interface class name list of objects specified key with the script.

%ID_CLASS_LIST%

The interface class name list of objects specified id with the script.

%OBJECT_NAME%

The UDF variable name under generating.

%KEY_OBJECT_LIST%

The UDF variable name list of objects specified key with the script.

%ID_OBJECT_LIST%

The UDF variable name list of objects specified id with the script.

7.3.3 Class macro

The macro definition of Table 7.3 can be used only between macro rule $CLASS_TEMPLATE_BEGIN$ and $CLASS_TEMPLATE_END$.

Table 7.3: makeinterface macro definition list (2)

%CLASS_NAME%
Class name under generating

%ATTRIBUTE_PARAMETER_LIST%
Parameter list used by the constructor.

%ATTRIBUTE_INITIALIZE_LIST%
The initialization declaration list of attributes used by the constructor.

%ATTIBUTE_DEFINITION_LIST%
attribute definition list.

%ATTRIBUTE_INPUT_LIST%
the input-variable list of all attributes

Usage:
is >> %ATTRIBUTE_INPUT_LIST%;

%ATTRIBUTE_OUTPUT_LIST%
– output-variable list of all attributes

Usage::
os << %ATTRIBUTE_OUTPUT_LIST% <<"";

%POSTFIX%
Format front-end character string

%PREFIX%
above-mentioned output Format postposing character string at the time of the above-mentioned output.

Usage:
os << %PREFIX% %ATTRIBUTE_OUTPUT_LIST% %POSTFIX% <<"";

7.3.4 Attribute macro

The macro definition of Table 7.4 can be used only between macro rule $ATTRIBUTE_TEMPLATE_BEGIN$ and $ATTRIBUTE_TEMPLATE_END$.

Table 7.4: makeinterface macro definition list (3)

%ATTRIBUTE_NAME%
Attribute name under generating

%ATTRIBUTE_SYMBOL%
The UDF variable name corresponding to the above.

%ATTRIBUTE_PARAMETER_LIST%
The parameter list used by the constructor.

The example of a template file which used these macroes is shown Table 7.5. template.txt (par template file which generates an IObject interface)

Table 7.5: Example template file for makeinterface

```
// This file is ganerated by makeinterface tool in Platform utility.
// You may edit this file to add any class or method, and to change output format etc.
//        template version V2.0
//                        Generated by: %GENERATED_VERSION%
//                        Generated date: %GENERATED_DATE%

#ifndef _%INTERFACE_NAME%_H_
#define _%INTERFACE_NAME%_H_

#include "udfobject.h"
#include %INCLUDE_LIST%

extern TAB tab;        // tab formatting object for ostream

class IUdfInformation {
public:
        IUdfInformation(const string& file="%DEFINITION%") : deffile(file) {}

        // UDF File Information
        const string& getDefinition() { return deffile; }

private:
        string deffile;
};

class %CLASS_LIST%;

$CLASS_TEMPLATE_BEGIN$
class %CLASS_NAME% : public UDFObject {
public:
        %CLASS_NAME%() : UDFObject() {}
        virtual ~%CLASS_NAME%() {}
        virtual const char* GetName() const { return "%OBJECT_NAME%"; }
        %ATTRIBUTE_DEFINITION_LIST%

public:
        virtual void _Input(istream& is) {
                is >> %ATTRIBUTE_INPUT_LIST%;
$IF_MAP_TYPE$
#if defined(USE_COMMON_SUPORT)
                if (getRoot())        getRoot()->addMAP(this);
#endif
$IF_TYPE_END$
        }
        virtual void _Output(ostream& os) {
                os << tab << "{";
                tab.enter();
                os << %ATTRIBUTE_OUTPUT_LIST% << " ";
                tab.exit();
                os << tab << "}\n";
        }
};
$CLASS_TEMPLATE_END$
```

```
#endif
```

7.4 Script file

The generation method can be changed by the script assignation by script_file. The contents of substitution of the macro definition of template_file are changed by the method specified with the script.

A script consists of an identifier and a blank and divided parameter. You can use script commands shown in Table 7.6.

Table 7.6: makeinterface script list

bind
: interface class name, C++ user-class name
1) %CLASS_LIST%, it is replaced by the C++ user-class name.
2) Generation by the macro rule is not performed.

include
: Include file required for the compile after a binding head
%INCLUDE_LIST%, it is replaced by the specification-file name.

top
: Top object name of UDF data division
Assignation object names are enumerated by %TOP_OBJECT_LIST%.

key
: KEY class name of a UDF definition part
Assignation class names are enumerated by %KEY_OBJECT_LIST%.

id
: ID class name of a UDF definition part
Assignation class names are enumerated by %ID_OBJECT_LIST%.

simple
: Class name which performs differential script processing
Assignation class names are enumerated by %SIMPLE_OBJECT_LIST%.

The example of a script file which used these scripts is shown Table 7.7. common_udf_script.txt (script which generates across the board UDF)

Table 7.7: Example script file for makeinterface

```
// This file is default script for makeinterface tool in Platform utility.
// To Do:        bind IObject to user class
bind IVector3D Vector3d

// To Do:   include user class header
include "Vector3d.h"

// To Do:        sujestion to makeinterface tool
// top objects
top     parameter
top     mesh
top     field
top     dynamics_manager
```

```
// keymap objects
key     PartialRegion
key     PartialRegionCondition
key     ScalarField
key     VectorField
key     TensorField
key     Parameter
key     Procedure
// idmap objects
id      Vertex
id      Edge
id      Face
id      Cell
// other objects
simple  Mesh
simple  StructuredMesh
simple  UnStructuredMesh
```

When the binding-head assignation of the pointer to a user definition class is carried out at a

7.5 Restrictions

- interface class, it cannot respond to the situation that the object interface class serves as an element of UDFArray.

In this case, when reading the correspondence object in a UDF file using the generated interface class, a constructor ends in ABNORMAL STATUS.

Chapter 8

Bugs in this release

makeinterface tool When the binding-head assignation of the pointer to a user definition class is carried out and the correspondence object in a UDF file is read into an interface class in the situation that the object interface class serves as an element of UDFArray, with using the generated interface class, a constructor terminates abnormally.

Chapter 9

Error messages

9.1 PFException

An error message when PFException is thrown is shown below.

- out of memory ...

  Memory is insufficient.

- illegal use of OpenUDF.

  The call sequence of OpenUDF or the parameter assignation is wrong.

- illegal use of CreateUDF.

  The call sequence of CteateUDF or the parameter assignation is wrong.

- illegal use of AppendUDF.

  The call sequence of AppendUDF or the parameter assignation is wrong.

- file not found.

  The specified file is not found. When a file is a UDF definition file, the assignation of environment variable UDF_DEF_PATH may be wrong.

9.2 LocationException relation

An error message when LocationException is Throw(ed) is shown below.

- illegal use of Location

  The method of a Location class is wrong.

9.3 UDFhandleException relation

An error message when UDFhandleException is Throw(ed) is shown below.

- FATAL ERROR: UDFistream read error occured.

9.4 UDFstreamException relation

An error message when UDFstreamException is Throw(ed) is shown below.

- FATAL ERROR: It cannot read in order of a convention of the data host specified from UDFistream read error occured.

- UDFistream. The mistake of an interface object header file or the possibility of a program mistake is high.

9.5 UDFStreamException relation

An error message when UDFStreamException is Throw(ed) is shown below.

- UDFistream read error occured.

  It cannot read in order of a convention of the data host specified from UDFistream. When the header file of an IObject interface does not have a UDF definition and adjustment, it is displayed in many cases.

- UDFostream write error occured.

  The data host specified to UDFostream cannot be written out. When the header file of an IObject interface does not have a UDF definition and adjustment, it is displayed in many cases.

9.6 UDFManagerException relation

An error message when UDFManagerException is Throw(ed) is shown below.

- illegal use of Location

  The method of a Location class is wrong.

9.7 UDFObjectException relation

An error message when UDFObjectException is Throw(ed) is shown below.

- UDFObject array size error occured.

  The number in the case of reading a UDFarray object in an internal stream is amusing. When the header file of an IObject interface does not have a UDF definition and adjustment, it is displayed in many cases.

9.8 UDF parser relation

A Parse Error message has the possibility of the mistake of UDF format or data. Refer to a "highly efficient material design platform UDF grammar" for details.

9.9 an internal error

Appendix A

Tutorial: makeinterface

A.1 Overview of makeinterface tool

Makeinterface tool is a tool which generates automatically the interface class definition which outputs and inputs the data of the specified UDF file.

When the data structure of UDF becomes complicated, it will become troublesome to create the program which outputs and inputs by C++. If a makeinterface tool is used, these interface programs can be automatically generaed by a little sudjestions.

The contents of the tutorial performed here are as follows.

Automatic generation of an symple class definition

The IUdfInformation class which outputs the header information of specified UDF is generated automatically, and it actually moves. You will understand the basic usage of makeinterface tool.

Generating UObject interface

Use the general-purpose template (uobject_template.txt) for generating automatically the interface class which uses UDFManager, and it actually moves. After ending this tutorial, you will understand the usage of UDFManager class.

Generating IObject interface

Use the general-purpose template (template.txt) for generating automatically the interface class which uses PFinterface, and it actually moves.

After ending this tutorial, you will understand the usage of PFinterface class that are mainly used with OCTA engines, and can understand the method for mounting I/O of a UDF file.

Moreover, he can also understand the method for directing mapping to a C++ variable from UDF data. All source files of this tutorial are prepared in directory: $PF_HOME/tutorial/makeinterface.

Let's start.

A.2 Automatic generation of an simple class definition

A.2.1 Making a class UDF information

We will generate automatically the IUdfInformation class which outputs the header information of specified UDF file. You will understand the basic usage of makeinterface tool.

Please check the contents of info_template.txt.

This outputs the header information of specified UDF. It is a template for generating an interface class header.

The following symbols are macroscopic and are replaced at the time of automatic generation.

%GENERATED_DATE% The date and time

%INTERFACE_NAME% which were generated automatically Interface class name

%DEFINITION% UDF definition file name

Table A.1: Example script for simple class ganaration

```
// This file is ganerated by makeinterface tool in Platform utility.
// You may edit this file to add any class or method, and to change output format etc.
//            Generated date: %GENERATED_DATE%
#ifndef _%INTERFACE_NAME%_H_
#define _%INTERFACE_NAME%_H_

#include "udfobject.h"
#include "pfinterface.h"

class %INTERFACE_NAME% {
public:
    %INTERFACE_NAME%(const string& file="%DEFINITION%") : deffile("%DEFINITION%")
        {
            UDFhandle *udfin = OpenUDF(file.c_str());
            recnum = udfin->getTotalRecord();
            project = udfin->getProjectName();
            engine = udfin->getEngineName();
            version = udfin->getEngineVersion();
            iotype = udfin->getIOType();
            comment = udfin->getComment();
            CloseUDF(udfin);
        }

    // UDF File Information Access method
    const string& getDefinition() { return deffile; }
    const int getRecordNum() { return recnum; }
    const string& getProjectName() { return project; }
    const string& getEngineName() { return engine; }
    const string& getEngineVersion() { return version; }
    const string& getIOType() { return iotype; }
    const string& getComment() { return comment; }

private:
    string deffile;
    size_t recnum;
    string project;
    string engine;
    string version;
    string iotype;
    string comment;
};
#endif
```

A.2.2 Generating a class by makeinterface tool

Execute the following command in a shell or a command window.
makeinterface -D myudf.udf -I info_template.txt -N IUdfInformation

Here, the meaning of a parameter is as follows.

-D UDF file name

-I Template file name

-N Interface name

A.2.3 Check IUdfInformation.h

Please check that macroscopic contents are replaced.

Table A.2: Example of ganerated simple class by makeinterface

```
// This file is ganerated by makeinterface tool in Platform utility.
// You may edit this file to add any class or method, and to change output format etc.
//            Generated date: Wed May 02 16:47:29 2001

#ifndef _IUdfInformation_H_
#define _IUdfInformation_H_

#include "udfobject.h"
#include "pfinterface.h"

class IUdfInformation {
public:
    IUdfInformation(const string& file="myudf.udf") : deffile("myudf.udf")
        {
            UDFhandle *udfin = OpenUDF(file.c_str());
            recnum = udfin->getTotalRecord();
            project = udfin->getProjectName();
            engine = udfin->getEngineName();
            version = udfin->getEngineVersion();
            iotype = udfin->getIOType();
            comment = udfin->getComment();
            CloseUDF(udfin);
        }

    // UDF File Information Access method
    const string& getDefinition() { return deffile; }
    const int getRecordNum() { return recnum; }
    const string& getProjectName() { return project; }
    const string& getEngineName() { return engine; }
    const string& getEngineVersion() { return version; }
    const string& getIOType() { return iotype; }
    const string& getComment() { return comment; }

private:
    string deffile;
    size_t recnum;
    string project;
    string engine;
    string version;
    string iotype;
    string comment;
};
#endif
```

A.2.4 Check udfinfo_test.cpp.

This program is a test program of using IUdfInformation class. The header file which generated automatically by makeinterface is included, the instance of a UdfInformation class is built, and the header information of UDF is asked.

Table A.3: Example of using ganerated class

```
/*
 *
 *  Material Modeling Platform Package
 *
 *  Copyright(c) 2000, The Japan Research Institute, Ltd.
 *  All rights reserved.
 *
 *  $Id: chap11_eng.tex,v 1.1 2002/02/21 12:58:40 nishio Exp $
 *
 */
#if defined(WIN32)
#pragma warning(disable:4786)
#endif
// Include header file genarated by makeinterface tool
#include "IUdfInformation.h"

void UDFInfoTest(const char *input_udfpath)
{
    // Construct the instance of UDF information class.
    IUdfInformation info(input_udfpath);

    cout << "====== " << input_udfpath << " =======" << endl;
    cout << "UDF definition:\t" << info.getDefinition() << endl;
    cout << "total record number:\t" << info.getRecordNum() << endl;
    cout << "Project name:\t" << info.getProjectName() << endl;
    cout << "Engine name:\t" << info.getEngineName() << endl;
    cout << "Engine version:\t" << info.getEngineVersion() << endl;
    cout << "IN/OUT type:\t" << info.getIOType() << endl;
    cout << "Comment:\t" << info.getComment() << endl;
    cout << "=============" << endl;
}
```

A.2.5 Build and execute the generated program

Build our test program by following make command.

make testinterface

Execute the program by the following command.

testinterface -T1 -I myudf.udf

The header information of myudf.udf is displayed on standard output. Here, -T1 is an option for specifying a tutorial 1.

```
====== myudf.udf =======
UDF definition: myudf.udf
total record number:    0
Project name:   makeinterface tutorial
Engine name:    myEngine
Engine version: TestVersion
IN/OUT type:    IN
Comment:        This file is sample UDF file for makeinterface tutorial.
=============
```

I think you thought "What is differ from programming directly by myself?". That's right. However, there is a big difference. A template is programmed in makeinterface. Thereby, it is wide opened from repeat programming from which a variable name differs.

As mentioned above, the procedure which uses a makeinterface tool becomes as follows.

1. Creating UDF file.
2. Programming the template file.
3. Generating the interface class header.
   The header file of an interface class is generated automatically using makeinterface tool.
4. Building program.

Next, the method which uses the general template of makeinterface by which uses and par offer is made is explained.

A.3 Generating UObject interface

Use the general-purpose template (uobject_template.txt) for generating automatically the interface class which uses UDFManager, and it actually moves. He can understand the method which uses UDFManager for altitude more.

Since it can output and input by specifying a UDF variable name, although it can be used simple in UDFManager, since it is necessary to define a corresponding C++ class in the case of data with complicated structure, the amount of programs increases very much.

Since the data structure is also defined as the UDF file, it is possible to generate a C++ class automatically using this information. The general-purpose template (uobject_template.txt) for performing this is prepared.

A.3.1 Check uobject_template.txt

Although it is not necessary to all understand the contents of this template, it explains briefly. The macro used as a nucleus is to $CLASS_TEMPLATE_BEGIN$ to $CLASS_TEMPLATE_END$, and the definition of the C++ interface class corresponding to the data structure defined in the UDF file is generated.

In this, it is

%CLASS_NAME% . Interface class name

%ATTRIBUTE_DEFINITION_LIST% All attribute definition

(s) etc. are replaced.

If you are interested in other macro descriptions, please read Chapter 7.

A.3.2 Check uobject_test.cpp

This program outputs to another UDF file which specified only the record count which read the data of the UDF file which used and specified UDFManager, and specified the same contents.

The header file and IUdfInformation.h which have been included at the head Header file Umyudf.h generated by the tutorial 1 It is the header file of the interface class generated corresponding to the definition of myudf.udf.

Umolecule is an interface class corresponding to the data of a UDF file, and is this instance. To molecule UDF molecule data are read.

Although an output performs the loop of the specified record time and the contents of an interface object instance are outputted as it is, the processing which sets a calculation result to an interface object will enter in fact in the meantime.

Table A.4: Example of using UDFManager interface classes

```
/*
 *
 *  Material Modeling Platform Package
 *
 *  Copyright(c) 2000, The Japan Research Institute, Ltd.
 *  All rights reserved.
 *
 *  $Id: chap11_eng.tex,v 1.1 2002/02/21 12:58:40 nishio Exp $
 *
 */
#if defined(WIN32)
#pragma warning(disable:4786)
#endif
// UDF I/O coding sample using UDFManager interface
//     2001/1/10  JRI Y.Nishio
#include "IUdfInformation.h"
#include "Umyudf.h"

//*************************************/
// To do:
// Declare interface objects to I/O.
//*************************************/
static Umolecule molecule;
//*************************************/

// test for input from UDF
static void ValueInput(UDFManager & uf)
{
//*************************************/
// To do:
// Select UDF object path to input.
//*************************************/
    molecule.get(uf, Location("molecule"));

    cout << molecule;
}

// sample coding to output UDF object as XML file.
void UObjectXMLOutputTest(const char *filepath)
{
    ofstream fout(filepath);
```

```
    fout << "<molecule>" << endl;
    fout << molecule;
    fout << "</molecule>" << endl;
    fout.close();
}

// entry routine to input UDF file
void UObjectInputTest(const char *udfpath, const int restart_step)
{
    try {
        UDFManager uf(udfpath, UDFManager::READ);

        // get the number of record in the UDF
        UINT recnum = uf.totalRecord();

        cout << "record number:" << recnum << endl;
        if (restart_step >= 0 && restart_step < (int)recnum )
            uf.jump(restart_step);

        ValueInput(uf);
    } catch (Location::LocationException &e) {
        cerr << e.what() << endl;
    } catch (UDFManager::UDFManagerException &e) {
        cerr << e.what() << endl;
    }
}

// sample coding to output culculated results.
static void SimpleOut(const UDFManager & uf)
{
    try {
        // Select UDF object path to output.
        molecule.put(uf, Location("molecule"));
    } catch (Location::LocationException &e) {
        cerr << e.what() << endl;
    } catch (UDFManager::UDFManagerException &e) {
        cerr << e.what() << endl;
    }
}

// entry routine to output UDF file
void UObjectOutputTest(const char *udfpath, const int end_step)
{
    try {
        IUdfInformation info;
        // You need to select the UDF definition file to create new UDF file.
        // interface object knows the UDF definition file.
        UDFManager uf(udfpath, info.getDefinition());

        for(int i=0; i <= end_step; i++) {
            string recname = uf.newRecord("record");
            cout << recname << " Output" << endl;
            SimpleOut(uf);
        }
        uf.write(udfpath);
    } catch (UDFManager::UDFManagerException &e) {
        cerr << e.what() << endl;
    }
}
```

A.3.3 Build and execute the generated program

Build out test program by the following command.

make testinterface

Since the generation processing by makeinterface is described by Makefile, if myudf.udf is corrected to it, it will be on an automatic target. Umyudf.h is updated.

Execute the program by the following command.

testinterface -T2 -I myudf.udf -O testout2.udf -e2

The contents of data of myudf.udf are displayed on standard output. Here, -T2 is an option for specifying a tutorial 2. Moreover, data are outputted even to the record specified to be the file specified by -O by -e.

Please check testout2.udf that was created by the generated program.

Thus, if makeinterface is used, the want of programming about I/O of UDF data will almost be lost.

A.3.4 UDFManager class methods

It is also assistance [knowledge] to change the contents of processing of uobject_test.cpp using many methods currently prepared for UDFManager.

About method specification, reffer Section 5.7 and 5.8.

Now, let's begin the last tutorial.

In the interface library, the IObject package other than a UDFManager package is offered. With this package, I/O of UDF data can be treated as a stream and the output to other data format can also be performed simple. Moreover, mapping to a C++ variable from UDF data can be directed.

A.4 generation of an IObject interface

Use the general-purpose template (template.txt) for generating automatically the interface class which uses PFinterface, and it actually moves.

He uses the PFinterface interface currently used with material engines, and can understand the method for mounting I/O of a UDF file. Automatic generation of a C++ class is the same as a tutorial 2. The general-purpose template (template.txt) for performing this is prepared.

Moreover, he can also understand the method for directing mapping to a C++ variable from UDF data. In order to direct this, a script file (script.txt) is used.

A.4.1 Check template.txt

Although it is not necessary to all understand the contents of this template, it explains briefly. First, the IUdfInformation class for making a UDF definition file name memorize is defined.

This class is using the simplex thing here, although it is also possible to transpose to what was explained by the tutorial 1.

the macro used as a nucleus is to *tutorial2−−similarly*)CLASS_TEMPLATE_BEGIN$ to $CLASS_TEMPLATE_END$, and the definition of the C++ interface class corresponding to the data structure defined in the UDF file is generated

In this, it is

%CLASS_NAME% . Interface class name

%ATTRIBUTE_DEFINITION_LIST% All attribute definition

(s) etc. are replaced.

About other macroscopic symbol descriptions, if you interested in this topc, please read Chapter 7 ”an interface class automatic generation tool.”

A.4.2 Check script.txt

This file directs mapping to a C++ variable from UDF data. bind IVector3D Vector3d is pointing saying "When you map Vector3D of a UDF definition in an interface object, declare as Vector3d class."

This is enabled to use the across the board library currently built in advance, and it can carry out direct exploitation of the instance of a Vector3d class within a program.

include"Vector3d.h" The include file for using the above-mentioned library is directed. Macro %INCLUDE_LIST% of a template file is replaced by these directions.

A.4.3 Check iobject_test.cpp

This is the program outputted to another UDF file which specified only the record count which read the data of the UDF file which used and specified PFinterface, and specified the same contents. (It is completely functionally the same as a tutorial 2)

The header file and Imyudf.h which have been included at the head It is the header file of the interface class generated corresponding to the definition of myudf.udf.

Imolecule is an interface class corresponding to the data of a UDF file, and is this instance. To molecule UDF molecule data are read. With a PFinterface package, in order to output and input in stream form, it is necessary to judge whether the object specified before the input exists.

Although an output performs the loop of the specified record time and the contents of an interface object instance are outputted as it is, the processing which sets a calculation result to an interface object will enter in fact in the meantime.

Table A.5: Example of using UDFObject interface classes

```
/*
 *
 *  Material Modeling Platform Package
 *
 *  Copyright(c) 2000, The Japan Research Institute, Ltd.
 *  All rights reserved.
 *
 *  $Id: chap11_eng.tex,v 1.1 2002/02/21 12:58:40 nishio Exp $
 *
 */
#if defined(WIN32)
#pragma warning(disable:4786)
#endif
// UDF I/O coding sample using IObject interface
//     2000/10/27  JRI Y.Nishio
#include "pfinterface.h"
#include "Imyudf.h"

//*************************************/
// To do:
// Declare interface objects to I/O.
//*************************************/
static Imolecule molecule;
//*************************************/

// test for input from UDF
static void ValueInput(UDFhandle * udfin)
{
    UDFistream is(udfin);
//*************************************/
// To do:
// Select UDF object path to input.
```

```
//*************************************/
    is.open("molecule");

    if (is.number()>0)
        is >> molecule;
    is.close();

    cout << molecule;
}

// sample coding to output UDF object as XML file.
void IObjectXMLOutputTest(const char *filepath)
{
    Vector3d::xmlOn();
    ofstream fout(filepath);

    fout << "<molecule>" << endl;
    fout << molecule;
    fout << "</molecule>" << endl;
    fout.close();
}

// entry routine to input UDF file
void IObjectInputTest(const char *udfpath, const int restart_step)
{
    UDFhandle *udfin = OpenUDF(udfpath, restart_step);

    // get the number of record in the UDF
    size_t recnum = udfin->getTotalRecord();
    cout << "input file total record number:" << recnum << endl;

    ValueInput(udfin);

    CloseUDF(udfin);
}

// sample coding to output culculated results.
static void RecordOut(UDFhandle * udfout)
{
    UDFostream os(udfout);
//*************************************/
// To do:
// Select UDF object path to output.
//*************************************/
    os.open("molecule");
    os << molecule;
    os.close();
}

// entry routine to output UDF file
void IObjectOutputTest(const char *udfpath, const int end_step)
{
    IUdfInformation udfinfo;

    // You need to select the UDF definition file to create new UDF file.
    // interface object knows the UDF definition file.
    UDFhandle *udfout = CreateUDF(udfpath, udfinfo.getDefinition().c_str());

    AppendUDF(udfout);

    for(int step= 0; step <= end_step; step++) {
        AppendUDF(udfout, step);
```

```
        cout << "==========" << udfpath << ":" << step << endl;
        RecordOut(udfout);
    }
    CloseUDF(udfout);
}
```

A.4.4 Build and execute our program

Build out test program by the following command.

make testinterface

Since the generation processing by makeinterface is described by Makefile, if myudf.udf is corrected, Imyudf.h will be automatically updated by it.

Execute the program by the following command.

testinterface -T3 -I myudf.udf -O testout2.udf -e2

It specified. The contents of data of myudf.udf are displayed on standard output. Here, -T3 is an option for specifying a tutorial 3. Moreover, data are outputted even to the record specified to be the file specified by -O by -e. Please check outputted testout3.udf.

If makeinterface is used, it will become unnecessary thus, to almost program about I/O of UDF data.

A.4.5 UDFObject class methods

It is also assistance [knowledge] to change the contents of processing of iobject_test.cpp using many methods currently prepared for the UDFObject package. About method specification, reffer Section 5.5 and 5.6.

A.5 What now

You have finished "makeinterface tutorial"!

Not only I/O of UDF but makeinterface can be generated automatically for a format-conversion program and an ultimate program.

Please utilize it by all means.