Import Data from File (IDF) utilities for programming data input to scientific codes

A. Yu. Pigarov and I. V. Saltanova

January 2000

Plasma Science and Fusion Center, Massachusetts Institute of Technology

167 Albany Street, Cambridge, MA 02139, U.S.A.

Abstract

IDF, Import Data from File, is a time-saving utilities for programming data input from an external text file into scientific codes written in C, C++, and FORTRAN languages.

The IDF package offers a user access to various data sets stored in file by their symbolic names and a simple syntax for data representation in a file. The data can then be placed in a file in arbitrary order, supported with comments, and imported into the user program in an order specified by a user. IDF is able to transfer input data to the aggregate targets (such as a structure or common block). IDF supports the pointer targets as well as the dynamic allocation of memory.

This manual presents the complete description of: the structure of the data source file required for IDF; the syntax for data representation in this file; the set of data types supported by IDF; the built-in calculator; the specification of targets in the user code to which the data should be imported from file, as well as the description of functions (utilities) to be called by a user in order to handle the data import with IDF.

E-mails: apigarov@psfc.mit.edu, apigarov@pppl.gov, apigarov@rex.pfc.mit.edu

I. Introduction

As a rule, the large scientific computer programs require numerous input data. It is a common approach to store the input data in a separate file which the user's program reads when it is running. This kind of input allows a user to change the input data and re-run his program without re-compiling the whole source code. The more sophisticated is the program input, the more time consuming and even the more hardworking is the programming of file input in C or FORTRAN languages[1]. In addition, the input file may contain numerous data units looking alike, so that it is hard to find and correct the selected unit amongst the other data units stored in this file.

IDF, Import Data from File, is a time-saving utilities for programming data input from an external text file for scientific codes. The IDF data source file is an ordinary text file which contains input data in a well understandable form. Each item in the IDF data file can be easily identified, modified or corrected. The input data can be supplied with the user's comments to make

[1] In C language, the programming of data input from a text file is based on the usage of standard functions (for example, the *fgetc* and *fscanf* functions) from C run-time library. When programming the data input in FORTRAN, a user manipulates with the special statements which are built-in this language (for example, the *READ* statement). These utilities (statements or functions) are the bricks from which a user can build a very sophisticated input for his code. The complex input programming requires time as well as the detailed knowledge of programming language (and, sometimes, the operational system). The program author must take care to read and transfer data in the proper order, in an order as the data stored in a file. It is not possible for a user to change without modifying the source code of user program the location of data set stored in a file.

the input clear to other users. In comments you can give instructions about the usage of input parameters in your program or any other important information concerning your code instead of referring a user to the documentation (if any) or to the source code of your program. To create[2] and modify[3] this file you can invoke your favorable editor.

The IDF package offers user an access to various data sets stored in a file by their symbolic names. The IDF data source file is then a collection of the *named data sets*. The data sets can be written in arbitrary order. The location of data sets in a file does not affect the result of data import. At the same time, the data can be imported into the user's program in an order specified by a user.

Each data set must have the *syntax* described in this manual. The syntax is very simple and resembles main features of C and FORTRAN languages. The IDF package has its own *built-in calculator* that allows a user to represent the 'arithmetic' type data stored in input file in terms of mathematical formula. IDF offers additional predefined set of *mathematical and physical constants* which are widely used in scientific codes. If necessary, each data unit (or any sequence of consecutive data units) can be *implicitly repeated* by specifying the repetition number instead of consequently repeating this unit many times in a file.

IDF is able to transfer data to the sophisticated *aggregate targets* created

[2]The **file name** can be represented by any combination of letters, digits, and underscore symbols, but must always begin with a letter.

[3]You can use the text output from scientific codes and modify it following the IDF syntax.

by a user, such as structures or common blocks. IDF supports the pointer targets as well as the dynamic allocation of memory.

The IDF package provides a complete set of functions (utilities) necessary to handle the data input from a text file. In this sense, IDF is an alternative to the direct programming of data input in C, C++, and FORTRAN languages. The package includes functions: to open, read, parse, and close the input text data file; to search for and to reference by symbolic name the data set given by a user in the input file; to convert the text data stored in file into binary values of the specified data type; to perform the mathematical calculations; to manipulate with a set of predefined physical constants; and to assign various targets (variables, arrays, structures, pointers, or common blocks) declared elsewhere in the user program, their values taken from the input file. All these functions are written in C language following the ANSI C standards that makes the IDF package portable across many computers and operational systems[4]. You can use the IDF functions in your code almost in the same way as the functions from C run-time library.

II. Structure of IDF data source file

The IDF data source file consists of one or more *data sets*. Each data set has its own name (i.e the set is named). This symbolic name is used by IDF to find the position of data in a file.

[4]At the same time, the IDF utilities are compatible at the source level, i.e they must be recompiled to run under new operational system or processor.

A data set is represented in a file in the form of an abstract assignment statement[5]. Consider the following three examples to illustrate the general structure of a data set:

*<keywords> data_name <subscripts> = <repetition_number < * > > data_unit ;*

<keywords> data_name <subscripts> : <repetition_number > data_block ;*

<keywords> data_name <subscripts> :

<repetition_number< * >><data_unit> <, > ...

<repetition_number * > <data_block> <, > ... ;

Any item in these statements can be optionally omitted, if it is shown within the angled bracket delimiters, <>.

The data name constitutes the left hand side of these statements, the *name field*. Optionally, the name can be preceded by the sequence of keywords. The keywords and the name must be separated by, at least, a blank space. The name can be followed by the subscripts.

The equals mark (=) and the colon mark (:) are used to separate the name from the data assigned to it.

The *data field* constitutes the right hand side of these assignment statements. The data field must end with the semicolon (;) symbol. This field must contain at least one *data unit* or *data block*. The *data block* is a sequence of data units enclosed within curly braces. Each data unit or block can be implicity repeated by specifying its *repetition number*. If data field consists of many data units or data blocks, each item should be properly separated

[5]although the name of data set does not represent directly the target, this form is similar to the assignment statement typical for the most programming languages.

with a space or comma.

There is no other restrictions on how the elements of named data set can be written in a file, other than the correct syntax for this set. On the one hand, each data set (as well as name field, data field, data block, and even some data units) may ocupy one or more lines in a file, and several named data sets can be placed simulteneously in one line in the data file, on the other hand. However, the structure and content of data input to a scientific code will be more clear to the users of this code in the case when each input parameter is separated and supported with comments, as it is illustated by the following example:

```
// IDF data file "couette.dat"
/*
-------------------------------------------
This file contains input data for the XXX code.
The XXX code simulates the 1D2V Couette flow problem.
The input data are written in the IDF language.
-------------------------------------------
*/
/* Physical input parameters */
Lz     = 1;    // A fixed plate separation distance, in cm
Knu  = 0.025;// Knudsen number
Tw  = 0.1;    // Constant plate temperature, eV
Uw  = 0.01; // Constant relative plate velocity, U/Vt
                // Vt is the thermal velocity, sqrt(2RTw)
Pr     = $1/3$, $2/3$ , 1; // The set of Prandtl numbers
/* Computational algorithm input parameters */
Nz = 1000; // number of nodes in the uniform mesh in z direction
```

Nv = 31,31; // number of nodes in the 2D velocity space

rather than in the case:

Tw=0.1;Uw=0.01;
Nz=1000;
L
=
1;
Knu= 0.025;Pr=0.33333, 0.66667,1.0;
Nv=31,31;

where the same input parameters are given in a structureless manner or chaotically in the data file.

III. Name of data set

The name of data set in a file may be represented by any combination of letters (in upper and lower cases), digits, and underscore symbols. But, at the same time, the name must begin with a letter.

Usually the data set name corresponds to the name of a variable declared in the user's code, although an exact coinsidence of these names is not necessary.

A data name itself must not be:

1) the IDF keyword (*char, character, int, integer, long, float, double, void, text, string, complex, static, extern, local*),

2) the name of standard mathematical function used in the built-in calculator (see the list of names given in APPENDIX III), or

3) the name from the IDF list of mathematical and physical constants (see APPENDIX I and II, respectively).

The name assigned to a data set must not match these intrinsic names reserved for special purposes. The IDF keywords can be written either in the lower or in the upper case letters. The names of standard constants must be given in the upper case letters. Any letter in the name of a built-in mathematical function can be written in the lower as well as in the upper cases.

Anonymous ('unnamed') data sets which appear without or to the left of the assignment mark (i.e the data sets '$aaa = 2;;123;$' and '$123 = aaa;$' are illegal) as well as the 'empty' names which do not followed by the data (i.e the following data sets '$aaa =;$' and '$aaa = /* \ aaa\_is\_a\_name \ */;$' are incorrect), will cause a fatal parse error.

IV. Scope and type of named data

The IDF language includes two special keywords: *static* and *extern*, in order to specify the scope (and storage) class for a named data set. One of these keywords can be optionally added to the data name at the left hand side.

The IDF considers all data names related to the *static* class as an internal IDF variables and allocates memory for these variables according to the specified data type (or, by the default, according to the type of a data unit). The whole file is the scope for these variables. The static variable exists for a time the source file is open. The static names cannot be referenced directly

from the users code. At the same time, these names can be used inside the file to create specific data units. (see a description for the *formula data unit* given further in this manual).

The data name supported with the *extern* keyword indicates that correspondent data set should be imported into the user's code on the user request. Such a name implicitly associates particular data set with a target defined outside the IDF. In other words, the name classified as *extern* refers data to one or more objects declared in the user's code. Using this name as an argument for the special function of IDF package, a user has an access to the data set stored in file.

The default class for data set names is *extern*, a user can always omit this keyword in writing data names in the source file.

The IDF generates a fatal error message in the case: when the static and extern keywords appear together in the name field, or when there are two or more data sets with the same name in file, but these names have different scope class.

Along with keywords specifying the data scope, the name field can also contain keywords which specify the type of data stored in the given data set. The type specifier keywords are: *char, string, text, int, long, float, double, void*, and *complex*. The following combinations of keywords are allowed: *complex float* and *complex double*, because they have a certain specific meaning.

For any internal variable (i.e for a variable associated with the data set of *static* class), the type specifier keyword is considered as the declaration for

this variable. The memory allocation, the initialization, and (optionally) the data type conversion occur for an internal variable according to the specified type. The default data type is *void*. The void variables receive the type of the corresponding data unit (see the section "data units" of this manual). The keyword *void* can be omitted for convenience.

For data sets specified as *extern*, the non-conflicting type specifier keywords are simply ignored.

V. Subscripts for data name

The data name in a file can be expanded by using subsripts. These subscripts are required to specify optionally either the array dimensions, or the particular element in an array.

The array subsripts can be prescribed to the name of data set in two ways.

First, you can specify the array dimensions in the same way as they are specified in C language. In the C-style case, the dataname is followed with a list of non-negative integer numbers and each number is enclosed within the pair of square brackets, i.e within [and].

Consider the following example of named data sets:
static double array1D[3] = 7,77,777;
static int array2D[2][3] : 1,2,3,4,5,6;

Here, the first data set named '$array1D$' is declared as the double precision, floating point, one dimensional array of three elements. In parsing

the file, the corresponding static target will be initialized by filling the array elements consequently with 7,77, and 777. The name '$array2D$' introduces the data set as the multi-dimensional array, declared as an "array of arrays". This array consists from six integer-number elements. The elements are stored in memory in the row-major order, i.e the last dimension index varies faster than the previest dimension index. That is, the first element will receive the value of the first data unit, $array2D[0][0] = 1$, the second element will get the value of the second data unit $array2D[0][1] = 2$, and $array2D[0][2] = 3$, and $array2D[1][0] = 4$, and etc.

Second, you can prescribe dimensions to the data set name using the FORTRAN style declaration. In the FORTRAN style case, the data name is followed with a list of integer numbers enclosed in parentheses, i.e within '(' and ')'. within the parentheses, non-negative numbers representing the dimensions must be separated from each other by commas.

Consider the same two data sets as discussed above in the case, when the FORTRAN slyle declaration is used for name subscripts: declaration:
static double array1D(3) = 7,77,777;
static int array2D(2,3) : 1,2,3,4,5,6;

In this case, three data units consequently initialize all three elements of the one dimensional array '$array1D$', i.e $array1D(0) = 7$, $array1D(1) = 77$, $array1D(2) = 777$. The multi-dimensional array '$array2D$' contains six integer number elements. The elements are stored in memory in the colomn-major order, i.e the last dimension index varies slower than the previest dimension index. That is, the first element will receive the value of the

first data unit, $array2D(0,0) = 1$, the second element will get the value of the second data unit $array2D(1,0) = 2$, and $array2D(0,1) = 3$, and $array2D(1,1) = 4$, and etc.

It is not allowed to prescribe array dimensions to data names using both C and FORTRAN style declarations (for example, the following data set: '*Garr* [2](2)[1] = 1,2,3,4;' is illegal). The negative or floating-point numbers in the dimension list will cause a fatal parse error.

Note, that in the case[6] when the data set corresponds to the internal (i.e **static**) array target, subscripts denote the dimensionality of this array. The empty pair of square bracket or empty parentheses are allowed. In this case, the correspondent default dimension is set to unity. For example, the data set '*static complex abra[2][ ][5] = (2,3);*' is equivalent to '*static complex abra[2][1][5] = (2,3);*'.

The maximal number of array dimensions which can be assigned to the data set name in a file is equal to 3. This number is introduced by means of macro-definition[7] IDF_FRMT_DIM_MAX.

In the case when data set has the **extern** class and corresponds to an array declared in the user's code, the subscripts attached to the name in

[6] In earlier version of IDF, the assignment of array dimensions to the **static** class names is not allowed and will result in a fatal error. There are also some restrictions on the type of data units for static variables: the data can be represented only by data units of an 'arithmetic' type, i.e by the integer, floating-point, complex, and formula units, and not by the character, string, or text data units.

[7] Here and further in the text we will use the words: macro-definition or macros, to denote the standard **C** preprocessor directive: **#define**.

a file denote the element in this array starting with which the stored data must be imported. In other words, non-negative numbers given in data name subscripts determine the offset from the beginning of array. If no subscripts are specified, the data will be transferred to an array target assuming the zero offset.

Assume that upper case letters K, M, N denote three dimensions of an array target in the user's program and that lower case letters k, m, n denote three subscripts assigned to the data set name in a file. In the case when the target is declared as the C-style three dimensional array '$[K][M][N]$', the FORTRAN-style subscripts '(k, m, n)' to the data set name define the offset equal to $N * (M * n + m) + k$, whereas the C-style subscripts '$[k][m][n]$' give the offset $N * (M * k + m) + n$. In the case when the target is declared as the FORTRAN-style array '(K, M, N)', the C-style subscripts '$[k][m][n]$' define the offset equal to $K * (M * k + m) + n$, whereas the FORTRAN-style subscripts '(k, m, n)' result in the offset $K * (M * n + m) + k$.

If the integer number is omitted inside the pair of square brackets or parentheses, it is assumed to be zero. If the number of subscripts assigned to data name in a file is less than the number of dimensions prescribed to the target, the lacking name subscripts are assumed to be equal to zero.

VI. Comments

A piece of data source file boardered by the character pairs /* and */ is considered as the C-style comment. A comment can contain any combination of characters including: white space, tabulation, line-feed, carriage-return,

form-feed, horizontal and vertical tabulation, and new-line characters. As in C language, C-style comments in IDF can not be nested.

Comments can be placed in data file anywhere a white space is allowed. When parsing the data file, IDF replaces a comment by a single white space (in some cases this additional space symbol may affect the data import or may generate a fatal error, for example, you can not break the data name or the integer-number data unit with a comment).

IDF also recognizes the C++ style comment: a text beginning with two consequtive forward slashes, //, and ending with a new-line symbol. The C++ style comment can be a part of the C style comment.

VII. Representation of data in IDF data file

In the data source file, the numerous data are represented by means of elementary data units. Such data units express a decimal, octal, or hexadecimal integer number, a character, a character string, a floating-point number, or a complex number in the literal form, i.e as a sequence of characters Hereafter this representation of data unit will be referred to as **S-value**. The legal data unit has its binary value, the **R-value**. The R-value can be related to one or more variables, the type of which is specified in C, C++, or FORTRAN languages. Each data unit is evaluated, converted and imported (i.e assigned to targets specified in the user's program) by the IDF utility.

Data units

There are seven types of data units: *character, string, text, integer, floating-point, complex*, and *formula*. This section explains how to define these data units.

Character data unit

Usually, the character data unit is represented by any printable ASCII symbol (except the backslash and single quotation mark symbols) that is enclosed in apostrophes. Between the apostrophes, a printable ASCII symbol can be accompanied by white spaces (for example: two units, 'a' and ' a ', express the same character a). The 'empty' character unit (given by the two consecutive apostrophes only) is not allowed.

Character data unit can be also expressed by some non-printable and non-ASCII symbols or by an escape sequence, if they are apostrophes delimited.

Character unit based on the ANSI escape sequence is one of:
'\$a$', '\$b$', '\$f$', '\$n$', '\$r$', '\$t$', '\$v$', '\' ', '\" ', '\\', '\?', '\0',
introducing respectively the ASCII characters for alert, backspace, form-feed, new-line, carriage-return, horizontal and vertical tabulation, single and double quotation mark, backslash, literal question mark, and end-of-string (NULL) symbol. In other cases, when character unit body consists from the backslash followed by any single symbol other than a digit or a, b, f, n, r, t, v, x, ', ", \, ?, $, the backslash is ignored (for example, the sequence '\c' produces the ordinary character 'c').

IDF recognizes character units based on the octal and hexadecimal escape sequences. The octal escape sequence starts with a backslash symbol followed by one to three octal digits[8]. In the case when less than three digits form the octal escape sequence, the escape sequence is automatically expanded by adding zeroes from the left hand side (for example, the unit '\41' is equivalent to the octal escape sequence '\041' and introduces the exclamation mark symbol). The hexadecimal escape sequence starts with the backslash followed by x or X, and thereafter, by one to three hexadecimal symbols[9]. The R-value correspondent to the octal or hex escape sequences must not exceed the eight bits of memory.

String data unit

The string data unit must be delimited in data file by a double quotation mark. This unit can contain any combination of printable ASCII characters and those escape sequences which represent an ASCII characters. In order to place the double quotation mark inside a string unit, the correspondent ANSI escape sequence (i.e \") has to be used.

The string unit can occupy more than one line in a data file using the same string concatenation rule as that adopted in C language. The backslash is considered as a string continuation mark, if newline symbol immediately follows the backslash. In this case the backslash is ignored and the string continues with the first character of the next line in data file.

[8]The octal digit is one of: 0,1,2,3,4,5,6,7 .

[9]The hexadecimal symbol is one of: 0,1,2,3,4,5,6,7,8,9, a,b,c,d,e,f, A,B,C,D,E,F .

Text data unit

The text data unit is a series of characters enclosed in the pound sign symbols (#).

Within a text unit, the IDF recognizes the so-called trigraph sequences. As in C language, the trigraph sequence consists of three characters. The sequence starts with two consecutive question marks followed by an ASCII symbol which will be converted into the punctuation character. The following nine trigraphs:

??= , ?? (, ??/ , ??) , ?? ' , ?? < , ?? ! , ?? > , and ?? −

will be replaced by the correspondent single ASCII characters:

, [, \ ,] , ∧ , { , | , } , and ∼ .

So, in order to deposit the pound sign character in the text, the correspondent trigraph sequence '??=' must be used instead of the pound sign symbol '#'.

Integer data unit

Integer data units can be represented as a decimal, octal, or hexadecimal number. IDF follows a simple rule to distinguish between these three types of numbers. By default, the integer unit is assumed decimal unless it starts with 0. If the integer unit begins with 0x or 0X, it is assumed hexadecimal, and it is octal, otherwise.

No delimiters required to introduce this unit in a data source file, although white space, comma, and some other symbols should be used in order to separate data units in a file.

Floating-point data unit

In general form, the floating-point unit has a sign, integral and fractional parts separated by decimal point, and optionally, an exponent. The integral or fractional part can be omitted, but the rest part must contain at least one digit. The exponent is expressed with a letter symbol (one of: e,E,d,D,g,G) followed by a sign symbol (optional) and at least one digit.

For example, the following data units:
777, $777.$, $+777.0e+0$, $7.77d+2$, $77.7D+01$, $.777g+3$, $77700G-2$,
introduce the same floating-point value which is equal to 777.

No delimiters required for a single floating-point unit in the data source file, but blank space and comma should be used in order to separate data units in a file.

Complex number data unit

The complex number is an ordered pair of floating-point values. The complex data unit is represented by two floating-point units which express respectively the real and imaginary parts: re and im, separated with a comma symbol. The complex unit must be enclosed in parentheses, i.e (re, im), and re and im must contain at least one digit. White space, tabulation, and new-line symbols are allowed inside the parentheses.

Formular data unit

The formula data unit can be represented by: (i) any executable mathematical expression (a formula), (ii) the name of mathematical or physical

constant from the standard IDF list, and (iii) the name of any *static* data set declared in the same data source file.

The content of a formula data unit must be enclosed within the dollar signs (by analogy with the TEX mathematics mode). Formular units cannot be nested, but subjected to concatenation in the case when two or more units are not properly separated.

VIII. Concatenation of delimited data units

In general case, if you are placing several data units on a line in the data source file, you must separate these units with blank space or comma symbols. At the same time, if two or more consequtive data units have different type of delimiters, it is not necessarily to separate these units with space or comma.

Concatenation rule is applied to data units delimited with ", ', #, and $ symbols in the case when the identical type units are not separated with white space, comma, or new-line symbols, or with a comment. For example, the translation of string type union: "The"" Bea""tle""s", is equivalent to the single string unit "The Beatles". The series of character units '\ ' 'x' '26' is equivalent to the hexadecimal escape sequence '\x26' which in turn represents the single ampersand symbol. The concatenation of two formula units $ sqrt(2) $$ ** 2 $ will result in the floating-point constant equal to 2.

IX. Implicit repetition of data units

A union of data units enclosed within curly braces (i.e { and }) is called **data block**. Data units in a block can have different types and must be properly separated by spaces or commas from each other. A block must contain at least one data unit, the 'empty' block is not allowed. Data blocks can be nested, i.e any block may appear within other blocks.

Consider the following example:

```
abc = { /* this data set introduces an alphabet. */
      { 'a', 'b',"cdef"}, {"ghij","klmn", "oprq"} "rst",
      'u'{"vw" "xyz"}
      };
```

Here the name '*abc*' is assigned to a data superblock. Superblock includes a C-style comment and consists of three data blocks and a single string-type data unit "*rst*". Each block contains three data units.

Data blocks in a series of blocks can be separated by a space or comma separators, although it is not necessarily. Use comments inside and outside a block where a space is lexically allowed. The appearence of: (i) consecutive commas inside or outside the block, (ii) the semicolon inside the data block, (iii) the openning brace followed by the comma, and (iv) the comma followed by either the semicolon or by the closing brace, causes a fatal error (even in the case when there are multiple spaces, comments, and tabulators between these pairs of symbols).

Important feature of data block is that each block can be implicitly re-

peated. To repeat the block, you should use an expression in which data block is multiplied at the left hand side by the repetition number. So, the expression of the form:

$$XXX * data\_block \quad ,$$

will be translated by IDF as "repeat XXX times the *data_block*". The repetition number XXX must be represented here by any positive integer number (in decimal, octal, or hexadecimal notation). Multiple repetitions of a block (for example: $1 * 2 * 3 * 4 * data\_block$) are not allowed. Instead, you can use the nested superblocks. For example, in the following expression: $1 * \{2 * \{3 * \{4 * data\_block\}\}\}$, the *data_block* will be repeated 24 times. If no repetition number followed by asterisk is given, the block appears only once, unless this block is a part of the super-block subjected to be repeated. The zero repetition number makes the IDF utility to ignore the content of correspondent data block (i.e in treating the data field, the zero times repeated block will be simply skipped).

In the same way you can implicitly repeat any single data unit in a file.

In order to demonstrate the implicit repetition of data unit and block, consider respectively the following examples:

$complexseven\_times\_seven$: 7*(7,0); // complex unit repeated 7 times

$onetwo\_times\_three$ = 3*{1,2}; //repetition number is 3 for a block

Here, the data set named $complexseven\_times\_seven$ contains seven identical complex numbers. In the data set $onetwo\_times\_twelve$, the sequence $\{1, 2\}$ is repeated 3 times, i.e the resulting data input is as follows: 1,2,1,2,1,2.

X. Built-in mathematical and physical constants

IDF reserves special names for symbolic representation of widely-used mathematical and physical constants. These names must be written with the upper case letters. All built-in names start with a sequence 'IDF_' in order to avoid their coincidence with data names given by a user. The list of mathematical constants is given in APPENDIX I. The physical constants are listed in APPENDIX II.

Being enclosed within dollar sign symbols, any special name constitutes a data unit of the formula type. For example, the formula data unit $ IDF_PI $ based on built-in name IDF_PI is equivalent to the floating-point data unit expressing the π number with a computer accuracy.

The special name can appear without delimiters inside the formula unit as a member of mathematical expression. In this case, the special name replaces the correspondent mathematical or physical constant in an expression.

XI. Built-in calculator

In general case the formula data unit is given by the mathematical expression including:

1) constants that represent the integer and floating-point number by their S-values;

2) algebraic operators;

3) references by name to the standard mathematical and physical constants specified in IDF, i.e the constants given by their R-value;

4) references by name to the R-values of *static* data sets defined by a user elsewhere in the given data source file;

5) names of built-in mathematical functions;

6) parentheses that delimit the list of function arguments, change the precedence of enclosed operators, or delimit the S-value constants;

7) commas that separate the function arguments from each other.

The expression in formula unit does not contain an assignment operator. The expression must be *executable*, its calculation must yield a single numeric value.

In order to introduce the complex number constants by their S-value in a formula, you must use the built-in functions: *complex, cmplx*, and *polar* (see APPENDIX III). For example, the resultant R-value of the following formula unit: $\$complex\ (1,2)\$$, is the complex number $(1, 2)$, whereas the formula expression $\$(1,2)\$$ is illegal.

The following operators are used in algebraic expressions: '+' for addition, '−' for subtraction, '∗' for multiplication, '/' for division, and '∧' or '∗∗' for exponentiation. All employed operators are binary operators, i.e they operate on the pair of operands. Except the exponentiation, all other operators associate from left to right. The computation of a binary expression results in the R-value.

Two algebraic operators must nowhere appear together in a formula. Most frequently, operators tend to appear together in the case when the algebraic operator is followed by a negative valued constant expression. To avoid fatal error, use parentheses to separate the negative constant in this

case. Unlike to C or FORTRAN, each S-value constant or a reference by name can be enclosed in the single or multiple parentheses, for example: \$ (0)+((10))*(2) \$, \$ $(-2)\wedge((-1))$ \$, \$ (IDF_PI)**(3) \$, and \$ $(1+(sqrt((2))))$ \$, . Nonessential parentheses do not affect the result and will be simply ignored by the calculator.

In evaluation of mathematical expression, algebraic operators range in their precedence. Plus and minus operators have the lowest precedence. If the expression contains operators other than '+' or '−', the addition and subtraction will be performed the last. Multiplication and division operators have the higher precedence than '+' and '−' operators. The precedence of the exponentiation operator is the highest. If a series of operators have an equal precedence, they are evaluated according to their associativity. The parenthesis may change the order of precedence for any operator.

All S-value constants entering the expression are converted either to the *long* integer or to the double precision floating-point (type *double*) constant. All mathematical computations are performed in double precision.

The type conversion of operands may occur even if both operands in a binary expression have the same data type. For example, to compute the function argument in the following expression: \$sqrt(1 + 5000000000)\$, the addition of integers will be replaced by the addition of floating-point numbers by conversion of each integer to the double precision floating-point value. In this case, the direct addition of integer numbers will cause an overflow in the resulting integer value.

If operands entering the binary expression have different data types, the

type conversion occurs before this expression will be evaluated. In the case when one of operands is a complex number, the other operand will be converted to the complex number (the converted complex number consists of the real number and the zero imaginary part). The execution of complex binary expression results in the complex number value. If integer and floating-point numbers constitute the binary expression, the integer number will be converted to the floating-point number. This binary expression results in the double precision floating-point number.

The built-in functions enter the expression in the same way as in C or FORTRAN languages. The name of function is followed by the parenthesized list of expressions correspondent to actual arguments of a function. A comma is used to separate the adjacent argument expressions of a function. The actual argument can be represented by any executable expression including those expressions which use the called function again.

Functions can be nested, i.e any function may serve as an argument of another function (for example: \$ $real(sqrt(exp(complex(1.0 + pow(2,3),$ IDF_PI)) \$). The call of a function without arguments or with a variable list of arguments is not allowed.

The full list and the description of built-in mathematical functions is given in APPENDIX III.

On using the function in mathematical expression, the type conversions may occur when: (i) the built-in function is directly responsible for the type conversion, such as *int, long, double, complex*, and *cmplx* functions; (ii) the unlike type value is passed as an argument to a function; and (iii) the value

returning by any other built-in function is subjected to further conversion according to the expression syntax.

In computing complicated expressions, the type of the resultant R-value is given by the last executable binary expression (for example, the result of \$ 1+ $int(cmplx(1))$ \$ is an integer number, whereas the result of \$ 1+$cmplx(1)$ \$ is a complex number), or by the last function call (for example, the last function that will be called in the expression: \$ $real(sqrt(complex(1,\mathrm{IDF\_PI})))$ \$, is $real(\ )$ and hence the resulting value has a storage type *double*).

The IDF calculator analyses the expression given in data file in the order, from left to right, searching for tokens (operands, constants, opening and closing parentheses, commas, and functions). Tokens are coded and stored until the accumulated expression can be completely executed or simplified by an execution of any its part. The maximal number of thus accumulated tokens is set to 100 by the macro-definition IDF_FORM_ELM_MAX.

Performing algebraic calculations, the IDF takes precautions against the overflow of floating-point number, the improper values of function arguments, and etc. IDF generates the correspondent error message before these errors occur.

XII. Parse of IDF data file

As soon as a user calls the special IDF function to open the data source file, IDF runs through this file searching for the data sets.

Insodoing, the IDF utility:

1) scans characters in the name field up to the first appearance of the assignment mark ('=' or ':');

2) skips comments and insignificant characters which may appear in the name field (such as the newline or carriage-return symbols, the nonessential consequent blank spaces and tabulators which may surround a name, etc);

3) splits the name field into three parts: the keywords, the data name, and the subscripts;

4) determines the scope class (static or extern) for a current data set;

5) determines the type of data set and its dimensionality;

6) stores (in the catalog) the name and some other information about the current data set;

7) if data set has *extern* class, skips the data field corresponding to the current data name up to the first appearance of semicolon (;) symbol;

8) if data set is *static*, parses the data field, evaluates, and stores the values of each data unit in the field;

9) continues with the next data set until the end of file will be reached.

As a result, the IDF utility creates two catalogs for the given input file.

The first catalog contains names and data R-values for all data sets specified in file as *static*[10]. The second catalog contains information about all data sets specified as *extern*. This information includes the data set name, the offset prescribed by data name subscripts, and the starting position of

[10]The total number of static variables given in file must not exceed 50, as specified by the macro-definition IDF_NAME_LISTL_N.

corresponding data field in the file[11].

On the user request, IDF imports data from the named data set. A user calls the special IDF function using the data set name as an argument of this function. The IDF function, first of all, searches this name in the catalog. If the name is found in the catalog, this utility positions the input file at the beginning of data field associated with a given name. After that, the function parsing the data field iteratively, from data unit to unit.

When parsing the data field assigned to the data name, the IDF utility:

1) scans characters in the data file starting with assignment mark ('=' or ':') until the end-of-field mark (i.e semicolon, ';');

2) searches for data units returning at the beginning of the data unit block, if the end of the block (i.e. right brace, '}') is reached and data units should be repeated;

3) removes comments and insignificant characters which may appear in the data field;

4) performs concatenation of data units, if it is required by a syntax;

5) if data unit is found, determines the type of data unit;

6) analyses the content of data unit, removes delimiters and any insignificant characters (for example: string continuation marks, newline or carriage-return symbols, consequent blank spaces and tabulators which may surround a number, and etc). In treating integer, floating-point, complex, and formula

[11]The total number of external data sets given in file must not exceed 50, as specified by the macro-definition IDF_NAME_LIST_N. In the excess, IDF stops execution and generates the error message.

units, IDF removes any kind of comment and replaces the comment with a single blank space. At the same time, IDF does not recognize comments placed inside the character, string, and text data units, i.e in between the ', ", and # delimiters (for example, in treating a character type data unit: '\ t/*this is an improper place for comment*/', IDF will generate an error instead of importing the single tabulation symbol);

7) stores in memory significant characters of data unit (except for the formula unit) for a time of further analysis and conversion;

8) converts the sequence of significant characters into a character, another character sequence, or a binary value according to the type of data unit;

9) performs mathematical calculations, if the data unit has a formula type, hence, converting the formula into the single binary value;

10) stores the obtained temporary R-value in the appropriate type variables (that provide the nominal accuracy for data representation) until this R-value will be transferred to the user's target.

In parsing the character unit, IDF searches for the significant character (white spaces are ignored in the case when the character unit contains any other ASCII symbol) or for the escape sequence. Escape sequence is replaced by its character or numerical equivalent. The result is temporary stored as an *unsigned int* variable.

In treating the string type unit, all escape sequences are replaced by their ASCII character equivalents. If the input string explicitly contains horizontal tabulation, this symbol is replaced by the equivalent number of blank spaces. The string unit input terminates at the first appearance of NULL character

(i.e \0). If the input string does not contain the NULL character, it will be automatically added to the end of the resulting string. The maximal length of the resulting string is limited to 328 (a number given by the macro-definition IDF_DBUF_LENGTH), and IDF will generate an error message for more lengthy string unit input from a file.

Except the trigraph case, the IDF re-writes the content of a text unit "as is", i.e without any kind of conversion. The length of any text unit is also limited to IDF_DBUF_LENGTH bytes.

The search algorithm for integer and floating-point units in the data file is more complicated than the search algorithm for a delimited data units. In skipping the leading spaces and tabs, the integer (or floating-point) unit can start either with a digit, or with the plus '+', minus '−', and decimal point '.' symbols. These units end properly with the first appearance of: (i) the semicolon (i.e ';') symbol which signals the end of data field, (ii) any space symbol (including the new-line and carriage-return), (iii) any kind of comment, (iv) the comma, (v) the curved braces, and (iv) optionally, the delimiter symbol which starts another data unit. If data unit does not contain the decimal point, the exponent part, and its value by modulus is less than the maximal *long int* number, the data unit content will be converted into an integer constant and stored temporary as *long int* variable. In any other cases, the unit will be converted into double precision floating-point constant and stored in a *double* variable.

When processing the complex number unit, IDF consequently search for two floating point numbers that are separated by comma. In success, IDF

stores the result of conversion in an array of two elements of the type *double*.

In treating the formula unit, the IDF utility substitutes the constants referenced to by their symbolic names and evaluates the mathematical expression using the built-in calculator. The resultant value is case dependent, it can be represented by either integer, floating-point, or complex R-value.

On the parse phase, the data unit gets its raw S-value and then converted into the temporary R-value. The further conversion and assignment of R-value to the user's target depend on the type of a target as will be discussed later.

The IDF utility stops executing when a fatal error occurs. It may be any parse, mathematical, file reading, or type conversion error. IDF generates an error message which usually consists of more than one line. Normally, the error message provides enough information to determine the cause of an error. In some cases, it contain the character which was inconsistent with the data representation rule, or the S-value of an improper data unit. The message also includes the data set name and its location in a file, the starting position of the data field, and the position current data unit. The position in file is given by three numbers: (1) the offset in bytes from the beginning of data file; (2) the line number, and (3) the column number. If an error appears in formula, IDF displays the list of accumulated tokens. The user can then invoke a text editor to correct (in the data source file) the text which caused an error.

XIII. Targets

Target is the final destination for the data stored in file. The target can be a single variable, a pointer, or an array, anywhere declared in the users code. Declared targets have different types representing the character, integer, or floating-point objects. A more complicated targets can be given by a sequence of unlike type objects with a known rule for positioning of objects in the common memory segment. Such a targets may be the structures[12] in C and C++ languages, or the common blocks in FORTRAN.

In order to import the named data set from a file, a user must specify the correspondent target in symbolic form by means of the Target String. Each Target String in IDF must have the syntax described in this manual.

The Target String is represented by ASCII character string and consists of the following characters:

1) target type symbols: 'c', 's', 't', 'i', 'l', 'f', 'd', 'z', and 'w' (these symbols can be also written in capital letters);

2) special symbols: %, #, &, and @;

3) digits representing any positive integer number written in decimal form;

4) parentheses, commas, or square brackets ([and]) that are used in order to specify the array subscripts;

5) curved brackets ({ and }) and asterisks (*) that are used to express the

[12]some dialects of FORTRAN language also include structures, for example: the VAX FORTRAN in VMS Version 5.0 or higher and the FORTRAN-90, Microsoft Fortran Power Station. The structures declared in FORTRAN program can also serve as a target for the IDF package.

implicit repetition of target unit in a string;

6) blank spaces and tabulators;

7) commas which separate the target units;

8) semicolon (;) and NULL ('\0') characters, with which the target string ends.

All other characters must not appear in the string. The blank spaces and tabulators are non-essential and will be ignored by the Target String parser.

The letters (**c,s,t,i,l,f,d,z,w**) are used as a keywords in the Target String. The stand-alone letter from this list denotes the single object of specific type. The same letter may represent an array or a pointer of the similar type, if it is used with subscripts or in the combination with special symbols. For structured targets, the Target String may contain a sequence of keywords.

Single object target

At present, IDF recognizes nine types of objects represented in the Target String by the following letters: 'c', 's', 't', 'i', 'l', 'f', 'd', 'z', and 'w'. Each type from this list corresponds to the particular data type specified in C and FORTRAN languages. An important characteristic of each type is the amount of bytes required to represent the corresponding object in memory.

Type c:

This type characterized the single character object which is a one byte variable. It corresponds to an object *X* declared as '*char X;*' in C language, as '*character*1 X*' in FORTRAN-77, and as '*character(1) X*' in FORTRAN-90.

Type s:

This type corresponds to an object which is a string variable. It is declared as an array of characters. The total number N of characters in the string (including the end-of-string symbol) must be specified. This type object X occupies N bytes of contiguous memory and is declared as '*char X[N];*' in C language, as '*character*1 X(N)*' (or, '*character*N X*') in FORTRAN-77, and as '*character(N) X*' in FORTRAN-90.

Type t:

This type characterizes the object declared as an array of characters. The total number N of characters must be specified. This type object X occupies N bytes of memory and is declared as '*char X[N];*' in C language, as '*character*1 X(N)*' (or, '*character*N X*') in FORTRAN-77, and as '*character(N) X*' in FORTRAN-90.

Type i:

This type represents the integer number, default declaration of which requires 4 bytes of memory. The correspondent object X is declared as '*int X;*' in C language, as '*integer*4 X*' in FORTRAN-77, and as '*integer(4) X*' in FORTRAN-90.

Type l:

This type represents the long integer number which normally requires 4 bytes of memory[13]. The correspondent object X is declared as '*long int X;*' in C language, as '*integer*4 X*' in FORTRAN-77, and as '*integer(4) X*'

[13]Depending on the computer and C compiler, an integer variable declared in C as **long int** may require the 8 bytes of memory

in FORTRAN-90.

Type f:

This type represents the single precision, floating-point number which occupies 4 bytes of memory. The correspondent object X is declared as '*float X;*' in **C**, as '*real*4 X*' in FORTRAN-77, and as '*real(4) X*' in FORTRAN-90.

Type d:

This type represents the double precision, floating-point number which requires 8 bytes of memory. The correspondent object X is declared as '*double X;*' in **C**, as '*real*8 X*' in FORTRAN-77, and as '*real(8) X*' in FORTRAN-90.

Type z:

This type represents the complex number given by two single precision, floating-point numbers. The correspondent object X occupies 8 bytes of contiguous memory and is declared as '*float X[2];*' in C language, as '*complex*8 X*' in FORTRAN-77, and as '*complex(4) X*' in FORTRAN-90.

Type w:

This type represents a complex number given by the pair of double precision, floating-point numbers. The correspondent object X requires 16 bytes of contiguous memory. This object can be declared as '*double X[2];*' in **C**, as '*complex*16 X*' in FORTRAN-77, and as '*complex(8) X*' in FORTRAN-90.

The stand-alone keyword: **c,i,l,f,d,z**, or, **w**, in the Target String represents directly a single object of given type.

Array target

Array, as the IDF target, is defined as a group of like type variables. The elements of an array are stored in memory contiguously in an increasing order, from the first element to the last. Array can be composed from single targets of the following type: *s, t, i, l, f, d, z*, and *w*. Note, that 'single' targets of type **s** and **t** are the arrays by definition.

One-dimensional array, as a target, can be specified in the following form:

$$\%\ \ number\_of\_elements\ \ target\_type\_symbol$$

where *target_type_symbol* is the keyword (one of: *s, t, i, l, f, d, z, w*, written in upper and lower cases), *number_of_elements* is the positive number written in the decimal form, and the percent sign (%) is used in this expression as the left-hand separator (optionally, % can be omitted). For example, Target Strings: "%100t" and "%6f", characterize respectively the text string of 100 characters, and the one dimensional, single precision array consisted of six elements.

You can also use either C, or FORTRAN syntax to prescribe dimensions to the target array (in the same manner as it has been discussed in the section "Subscripts for data name").

In the C-style case, the name is followed with a list of non-negative integer numbers, and each number is enclosed within the pair of square brackets, i.e within [and]. For example, Target Strings: "w[2][3]" and "s[100]", define respectively the two dimensional array consisted of six elements which are the double precision, floating-point, complex numbers; and the literal string

of 100 characters.

In the FORTRAN style case, the name is followed with a list of integer numbers enclosed in parentheses, i.e within (and). Inside the parentheses, the non-negative numbers representing the dimensions must be separated from each other by commas. For example, Target Strings: "i(2000)" and "d(2,3,4)", describe respectively the one dimensional array of integer numbers and the three dimensional array of double precision floating-point numbers.

You can define array targets in the generalized form:

$$\% \ number\_of\_elements \ \ target\_type\_symbol \ \ subscripts$$

In this case, the *number_of_elements* will be interpreted as an additional array dimension. For example, the real dimensionality of arrays in Target Strings: "%10i[2]", "%30s[20][10]", and "%4z(2,3)", correspond to "i[10][2]", "s[30][20][10]", and "z(2,3,4)", respectively.

Pointer target

Pointer is a variable which stores the address of an object to which the variable points. Normally, the pointer itself requires 4 bytes of memory[14].

The special symbols: ampersand (&) and *at* sign (@), are used in order to declare a pointer target in the Target String.

Consider the following examples in order to illustrate the specification of pointer targets.

[14]The size of memory required to represent a pointer may depend on the compiler, and the pointers to different type objects are not necessarily have the same size.

case 1:

$< \% >$ & $target\_type\_symbol$

case 2:

$< \% >$ & $< number\_of\_elements >$ $target\_type\_symbol$ $< subscripts >$

case 3:

$< \% >$ @ $target\_type\_symbol$

case 4:

$< \% >$ @ $< number\_of\_elements >$ $target\_type\_symbol$ $< subscripts >$

In all four cases considered here, the angled brackets are used to show that the enclosed item can be optionally omitted.

In the first and second cases, the ampersand (&) symbol is considered as an *address-of* operator which takes the address of an object which followed this operator. (in analogy to the similar operator in C language). The target (case 1) is a pointer variable which stores the address of single object of the given type $target\_type\_symbol$ = **c,i,l,f,d,z,w**. The target (case 2) is a pointer variable which points to an array of the $target\_type\_symbol$ type objects. In these two cases, the object must exist, i.e the user must allocate the correspondent amount of memory and assign the address to the pointer variable.

In cases 3 and 4, the *at* sign (@) qualifies the target as a pointer. In these cases the object must *not* exist. The @ sign tells the IDF utility to allocate the correspondent amount of memory and to assign the address to the pointer variable.

IDF does not support the targets which are defined as the nested pointers,

i.e Target Strings: "&&i", "@&z[100]", "@@w", "%&&&&&10d", are illegal.

XIV. Specification of multi-object target

The multi-object target consists of one or more *target units*. Each target unit can be written in the Target String in the following generalized form:
<%> <#> <@ or &> <integral_number> object_type_symbol <subsripts>
which represents the variety of targets which have been discussed in the previous section.

Comma is a valid separator for target units in the Target String. At the same time, two or more consequent commas appeared in a String cause the fatal error. Any number of consequent blank spaces or tabulators will be simply ignored. The usage of commas is illustrated by the following example: "100s, 100d(10,2), 5i[3][2]".

Fictitious target unit

The pound sign (#) is an additional symbol which may appear in the multi-object Target Srting. This symbol qualifies the target as a 'fictitious' target. Fictitious target means that the target is declared in the user's program, but at the same time, the corresponding named data set contains *no* data units associated with this target. When parsing the Target String, IDF ignores the fictitious target unit only in the sence that IDF associates no data units with this target, but IDF takes into account the presence of a fictitious target unit for calculating the address of other target units in the common

memory. You can use the fictious target units in order to assign values to the selected members of multi-object target.

Repetition of target unit in the Target String

In the case when the target unit starts with the special qualifier symbol (%,@,#, or &), the positive decimal integer number preceding this unit is considered as the repetition number for the whole target unit.

The repetition rule can be illustated by the following examples. According to this rule, the Target String "3%@100s[20]is equivalent to the String: "%@100s[20]%@100s[20]%@100s[20]%d", where the first target unit '%@100s[20]' is repeated three times. The following Target Strings: "5&d", "10%i", "3%5z[10]", are written in the much shorter form than the correspondent Target Strings: "&d &d &d &d &d", "iiiiiiiiii", "%5z[10], %5z[10], %5z[10]".

It is important to note, that because of *alignment* rules (see the next section), the Target String "3%c%d" is not necessarily equivalent to "c[3]%d", as well as the String "100%i,#d,3%i#d" is not the same as "i[100],#d,i[3],#d" or "i[103]".

Block of target units

The block of target units is defined as one or more target units enclosed within the curly braces ({ and }).

The content of a block can be repeated in the Target String in the following way:

$$XXX\ \{\ target\_unit\ <,>\ target\_unit\ <,>\ ...\ \}$$

where XXX denotes any positive integer number written in decimal form.

The target blocks can be nested, i.e one block may be a member of another target block, for example: "s[100]{6%d[4], 10{&s[100]@i[3]{#&z,d}}&w}". You can enclose any target unit within the nested braces. The following Target String: "2{3{4{5d}}}", which contains the multiple repetitions of the **d**-type target unit, is equivalent to String "120d".

XV. Alignment rules for multi-object target

The multi-object target characterizes the aggregate, the group of variables, pointers, or arrays stored together in memory. The members of the group may be of different types (i.e **c,s,t,i,l,f,d,z,w**). Such a target can be a structure or a common block declared in the user's program. The members of an aggregate are represented in memory consequently in an order as their names are given in the declaration list. The first member of the group has the lowest address and the last member has the highest address in group memory. The starting position for each member in group memory depends on the specific alignment rule used by a compiler to handle this group. In general, the alignment rule may depend on the type of members consisting the group. In the case when the group consists of unlike type members, the group memory may contain an unnamed spaces, the holes. At the same time,

if the group contains the array member, the elements in an array immedeately follow each other in group memory.

The alignment rule can be chosen by a compiler, according:

1) to the default compiler settings taking into account the processor peculiarities;

2) to the command-line option of a compiler, for example: **/Zp** option for the Microsoft C/C++; **-member_alignment** and **/Zp** options for DEC C compilers; **-dalign** option for SUN FORTRAN compiler; or **-align** option for DEC UNIX FORTAN compilers;

3) to the 'packing' directives defined in C or FORTRAN languages, which control the aligmnment of members of an aggregate object.

The alignment rule introduces the alignment boundary (or, in general case, the boundaries for each particular type of group members). Most frequently, the members (of structure or common block) are aligned in memory on the boundary which is the smaller of their own size or the specified packing size.

Some programming languages include the compiler directives which control the position of structure members in the computer memory. In C and C++ langauges such a directive is that given by the macros: '**# pragma pack (Npack)**'. In FORTRAN-90 the correspondent directive is the '**$ PACK: [Npack]**' directive. These directives tell a compiler to use one of simple packing rules[15]. The '*pack*' type directives specify the packing size

[15]The directives given within the source code of a user program override the settings established by the command-line option. At the same time, the compiler's manuals give

in terms of integer value *Npack* that is a power of two. The integer value *Npack* is passed as the macros argument to the compiler and characterizes the number of bytes to pad in order to align the data. The smaller *Npack*, the more compact the members of structure are packed in computer memory. At the same time, the performance may get worser for small *Npack* values (in particular, the time for accessing the members increases). The smallest *Npack* is zero, this value tells compiler to align the data on the byte boundary, i.e without any holes in memory between the members of structure or common block. The biggest $Npack = 9$ offers compiler the alignment on the page boundary, i.e the alignment of $2^9 = 512$ bytes.

To specify the boundary for data alignment, you can also use the following terminology. The value $Npack = 0$ corresponds to the *byte* boundary. The *word, longword, quadrword, octaword, hexword*, and *page* boundaries correspond, respectively, to the $Npack = 1, 2, 3, 4, 5, 9$ values.

Since the alignment rule may depend on the particular multi-object target, the processor, and the compiler, this part of IDF is the most intricated. The user must use IDF in this case with a caution. The data import breaks down with a fatal error, if the alignment rule chosen by the user for a target is incorrect. The IDF function responsible for the alignment control is written as the well isolated function which can be easily modified taking into acccount the additional perculiarities of your processor and compiler.

The variety of alignment rules the IDF utility handles by means of input parameter **align_mode**. In order to import the data stored in a file, a user

no warranty that the compiler will follow the '*pack*' directive rule.

must specify the value of *align_mode* parameter along with the Target String for each particular multi-object target. The integral values of *align_mode* correspond to the alignment rules considered below.

align_mode=0:

In this case, the target members are aligned on the byte boundary, i.e without any holes in memory between the members of structure or common block.

align_mode=1,2,3,4,5,6:

In these cases, the members (of structure or common block) are aligned in memory on the boundary which is the smaller of their own size or the specified alignment boundary. The specified boundary is the word ($align\_mode = 1$), longword ($align\_mode = 2$), quadword ($align\_mode = 3$), octaword ($align\_mode = 4$), hexword ($align\_mode = 5$), or page ($align\_mode = 6$) alignment boundary.

align_mode=7:

The target members are aligned on their natural boundaries, that is, on the next free boundary appropriate to the type of member. The character type members (**c,s,t**) are alingned on the byte boundary. The integer type members (**i,l**) are normally aligned on the longword boundary as well as the floating-point type members (**f,z**). The quadword boundary corresponds to the double precision type members (**d,w**).

align_mode=8,9,10,11,12,13:

The target members are aligned according to prescribed packing size, $Npack$. The prescribed alignment boundary is the word ($align\_mode = 8$), longword

(*align_mode* = 9), quadword (*align_mode* = 10), octaword (*align_mode* = 11), hexword (*align_mode* = 12), or page (*align_mode* = 13) boundary.

XVI. Import of named data set

IDF utility reads an input text file and imports the named data sets, name by name, in an order specified by a user.

In the simplest case when target is a single object, IDF reads the first data unit located imedeately after the data set name in file. The correspondent R-value is converted according to the target type and then assigned to the target. The rules for R-value to target type conversion are as follows: (i) the character R-value is equivalent to the type **c** target; (ii) the R-values of string and text data units corresponds to both **s**- and **t**-type targets ; (iii) the usual conversions of types (integer to floating point, floating-point to integer, single to double precision) may occur when R-values of integer, floating-point, complex, and formula data units are assigned to 'arithmetic' type target (**i,l,f,d,z,w**). Any attempt to assign the character R-value to 'arithmetic' type target or the 'arithmetic' type R-value to character type target (**c,s,t**), will cause the fatal error.

When string or text R-values are delivered to the **s** and **t** targets, the import of characters finishes if either the data unit content is exhausted or the last element of this target unit is imported.

If the set of R-values is assigned to an array target, the first R-value in the set is assigned to the first element of array, the second R-value is assigned to the second element of array, and so on, until either the set of data units

is exhausted or the end of array is reached. If the set does not use all the values given in data file, the remainder data units are ignored.

When subscripts are added to the data name in a file, they specify the element in an array target starting with which R-values should be imported from a given data set. In this case, IDF calculates the offset from the beginning of array (for the multi-dimensional arrays, a user must specify the Target String in order to introduce the dimensions of an array) and the correspondent number of elements will be left in the target before parsing the data set.

The IDF utility transfers data to the multi-object targets consequently, on the one-to-one basis, that is the value of each data unit will be assigned to the correspondent target unit in an order as the data units followed the data name in a file and as the target units are specified in the Target String. The R-value of each data unit will be converted (according to the target type given in the Target String) and assigned to the current target (the position of target in memory is calculated according to the specified alignment rule, i.e *align_mode*). The import of data finishes if either the list of target units or the sequence of data units in file get exhausted, whichever happens first (but, at least one value must be imported).

XVII. IDF functions to be used in the C program

The names of all IDF functions to be called in a user's C program start with the sequence of four characters '*idf_*'. The prototypes of these functions are given in the header file "**idfusr.h**".

The IDF functions must be called in the proper order. The function *idf_init* is called the first, in order to activate the IDF package. The *idf_open* function should be called the next to open the IDF data source file for input[16]. In success, you can use a variety of IDF functions to transfer the data stored in the current file to different targets declared in your program. Any attempt to import data from the unopened file will result in a fatal error. The function *idf_close* closes the current data file. The last function to be called is the *idf_finish*[17].

At an error, the IDF functions *idf_init, idf_finish, idf_open*, and *idf_close* automatically generate the error message, if the macros IDF_MISTAKE is set non-zero. If IDF_MISTAKE is defined as zero, you should call the *idf_err_prn* function to obtain the correspondent error message. If any kind of error occurs, all other functions responsible for data transfer from file to targets display the error message automatically.

idf_init

This function initializes some global IDF variables and allocates memory used by other functions of IDF package. Synopsis:

int idf_init();

The *idf_init* function receives no arguments. It returns zero in success and the positive integer number in the case of an error.

[16]Only one IDF source file can be opened at once. If you want to import data from another file, you should close the previously onepened file and continue the IDF input with the *idf_open* function for the next input file.

[17]If it is necessary to continue the IDF input after the *idf_finish* was called, you must start IDF again by calling the *idf_init* function.

idf_finish

The *idf_finish* function sets to zero all global IDF variables and de-allocates the memory which has been previously reserved for IDF with the help of *idf_init* function. Synopsis:

void idf_finish();

This function receives no arguments and returns no value.

idf_open

This function openes the specified IDF data source file for input. It also parsing the file and creates two catalogs (for static and external scope data) of data names given in this file. Synopsis:

*int idf_open(char *Fname);*

This function has one argument, *Fname*, which is a string containing the name of the input data file to be opened. It returns zero in success and the positive integer number at an error.

idf_close

This function closes the IDF data source file (if any file has been opened with *idf_open* function). Synopsis:

void idf_close();

This function receives no arguments and returns no value.

idf_err_prn

This function prints the error message for any error associated with the memory allocation, with the opening, reading, and parsing the IDF data source file, and with the creation of catalogs for data names. Synopsis:

void idf_err_prn();

This function receives no arguments and returns no value.

idf_c, idf_uc, idf_u

These functions import the named data set from a file for the single character target declared in the user's program as **char**, **unsigned char**, and **unsigned int**, respectively. Synopsis:

*int idf_c(char *name, char *Pobj);*

*int idf_uc(char *name, unsigned char *Pobj);*

*int idf_c(char *name, unsigned int *Pobj);*

The first argument of the function, *name*, is the string containing the name of data set to be imported. The second argument, *Pobj*, is the address of an object. Each function returns zero in success. At an error, the function returns a positive integer number (the error code). It also displays an error message in the case of error.

idf_i , idf_l , idf_f , idf_d , idf_z , idf_w

These functions import the named data set from a file for the single object target corresponding to the type **i,l,f,d,z,** and **w**, respectively. Synopsis:

*int idf_i(char *name , int *Pobj);*

*int idf_l(char *name , long *Pobj);*

*int idf_f(char *name , float *Pobj);*

*int idf_d(char *name , double *Pobj);*

*int idf_z(char *name , float *Pobj);*

*int idf_w(char *name , double *Pobj);*

The first argument of a function, *name*, is the string containing the name of data set to be imported. The second argument, *Pobj*, is the address of an object of the given type for **i,l,f,d** targets and the address of the first element of an array for the **z**- and **w**-type targets. Each function returns zero in success. In the case of error, the fuction returns a positive integer number and displays the corresponding error message.

idf_s , idf_t

These functions import the named data set from a file for the character string (**s**) and text (**t**) type targets, respectively. Synopsis:

*int idf_s(char *name , char *Str , int nelem);*

*int idf_t(char *name , char *Str , int nelem);*

The arguments of these functions are: *name* is the string containing the name of data set to be imported; *Str* is the address of the first element in the character string; *nelem* is the maximal number of characters in the string. Each function returns zero in success and the positive integer number in the case of error. At an error the function displays the error message.

idf_iarr , idf_larr , idf_farr , idf_darr , idf_zarr , idf_warr

These functions import a named data set from the current data file for the one-dimensional array target corresponding to the type **i,l,f,d,z,** and **w**, respectively. Synopsis:

*int idf_iarr(char *name , int *Arr , int nelem);*

*int idf_larr(char *name , long *Arr , int nelem);*

*int idf_farr(char *name , float *Arr , int nelem);*

*int idf_darr(char *name , double *Arr , int nelem);*

*int idf_zarr(char *name , float *Arr , int nelem);*

*int idf_warr(char *name , double *Arr , int nelem);*

Each function from this list receives three arguments: *name* is the string containing the name of data set to be imported; *Arr* is the address of the first element of an array; *nelem* is the maximal number of elements in an array (for complex number targets, **z** and **w**, each element in an array consists of two numbers). The function returns the number of imported elements in success. It returns the negative number at an error and displays the correspondent error message.

idf_get_array

This function imports a named data set from the current file for the multi-dimensional array target. Synopsis:

*int idf_get_array(char *name, char *TS, void *Ptarget);*

The arguments are as follows: *name* is the string containing the name of data set to be imported; *TS* is the Target String; *Ptarget* is the address of to the array target. The dimensionality of an array must be specified using subscripts to the target type specifier in the Target String. The subscripts can be also added (within a file) to the name of corresponding data set. In this case, the data set name subscripts denote the offset from the beginning of an array target (i.e the particular element in an array target starting with which the stored data should be imported). The function returns the number of imported elements in success. At an error, the function returns the negative number and displays the correspondent error message.

idf_get

This function imports a named data set from the current file and transfers data to the arbitrary target. Synopsis:

int idf_get(name , TS , align_mode , Ptarget);

*char *name, char *TS, int align_mode, void *Ptarget;*

The function arguments have the following meanings: *name* is the string containing the name of data set to be imported; *TS* is the Taget String; *align_mode* is the parameter defining the alignment rule; *Ptarget* is the address of the target. In success, the function returns the number of imported elements. In the case of error, it returns the negative number and displays the correspondent error message.

Examples of IDF usage in the C program

The following sample program written in C language demonstrates the data import from external file with the IDF package. In this program, all targets are the single variables and arrays. For each target, IDF imports the corresponding data set according to its symbolic name.

```
#include <stdio.h>
#include "idfuser.h"

int main()
/* The first sample program in C */
{
/*initialization of input IDF data file String*/
static char Fname[ ] = "couette.dat";
/* declaration of input parameters */
```

```
double Lz,Pr[3];
float Knu,Uw,Tw;
int Nz,Nv[2];
int ier;
/* start the IDF package */
ier=idf_init(); if(ier) goto er;
/* open the data file */
ier=idf_open(Fname); if(ier) goto err;
/* reading the data from file */
ier=idf_d("Lz", &Lz); if(ier) goto err;
ier=idf_f("Knu", &Knu); if(ier) goto err;
ier=idf_f("Uw", &Uw); if(ier) goto err;
ier=idf_f("Tw", &Tw); if(ier) goto err;
ier=idf_darr("Pr", Pr , 3); if(ier!=3) goto err; ier=0;
ier=idf_i("Nz", &Nz); if(ier) goto err;
ier=idf_iarr("Nv", Nv , 2); if(ier!=2) goto err; ier=0;
/* print the imported data sets*/
printf("Lz=%12.3e Knu=%12.3e\n",Lz,Knu);
printf("Uw=%12.3e Tw=%12.3e\n",Uw,Tw);
printf("Pr=%12.3e, %12.3e, %12.3e\n",Pr[0],Pr[1],Pr[2]);
printf("Nz=%5d Nv=%5d, %5d\n",Nz,Nv[0],Nv[1]);
/* close the data file */
err: idf_close();
/* finish the IDF */
er: idf_finish();
return ier;
}
```

The above program imports data from file "couette.dat". Assume that

this file contains the scientific data[18] and these data are stored in file in the same form as they have been given in Section II. Then, the following four strings will be displayed as a result of execution of this program:

```
Lz= 1.000e+00 Knu= 2.500e-02
Uw= 1.000e-02  Tw= 1.000e-01
Pr= 3.333e-01, 6.667e-01, 1.000e+00
Nz=  1000 Nv=   31,  31
```

Consider the next sample program to illustrate the data input from a file for the aggregate target with the IDF package. In this case, the input parameters are incorporated within a *structure* and are initialized all together using the symbolic name of the structure.

The C source file for the second sample program consists of:

the header file references

```
#include <stdio.h>
#include "idfuser.h"
```

the *COUETTE* structure type definition

```
#typedef struct {
   double Lz;
   float Knu,Uw,Tw;
   double Pr[3];
   int Nz,Nv[2];
                } COUETTE;
```

[18]As an example, we consider a data related to the so-called Couette problem, the classical problem of rarefied gas dynamics.

the C function *Couette_input*

```
int Couette_input(Cinp)
COUETTE *Cinp;
/*this function returns zero in success*/
{
/*initialization of input IDF data file String*/
char Fname[ ] = "couetteS.dat";
/* initialization of structure data name */
char StructName[ ] = "COUETTE";
/*initialization of alignment mode */
int amode = 7; /*assume the 'natural boundary' rule*/
/*initialization of Target String */
char Target_String[ ] = "dfffd[3]ii[2]";
int ier;
/* start the IDF package */
ier=idf_init(); if(ier) goto er;
/* open the data file */
ier=idf_open(Fname); if(ier) goto err;
/* reading the data from file to the Cinp structure*/
ier = idf_get(StructName, Target_String, amode, Cinp);
if(ier>0) ier=0;
/* close the data file */
err: idf_close();
/* finish the IDF */
er: idf_finish();
return ier;
}
```

and the program *main*

```
int main()
/* The second sample program in C */
{
/* declaration of input structure */
COUETTE Cinput;
int ier=0;
/* import data from file */
ier=Couette_input(&Cinput);  if(ier) goto err;
/* print the imported data sets*/
printf("Lz=%12.3e Knu=%12.3e\n",Cinput.Lz,Cinput.Knu);
printf("Uw=%12.3e Tw=%12.3e\n",Cinput.Uw,Cinput.Tw);
printf("Pr=%12.3e, %12.3e, %12.3e\n",
Cinput.Pr[0],Cinput.Pr[1],Cinput.Pr[2]);
printf("Nz=%5d Nv=%5d, %5d\n",
Cinput.Nz,Cinput.Nv[0],Cinput.Nv[1]);
err: return ier;
}
```

In this program, the input parameters, namely: *Lz, Knu, Uw, Tw, Pr, Nz*, and *Nv*, are the members of a structure, the type of which is declared as *COUETTE*. All IDF functions required to import data from "couetteS.dat" file are well isolated within the user's function *Couette_input*. The function *Couette_input* initializes in whole the input structure, *Cinput*, declared as *COUETTE* type in the *main* program. The program then prints the values of imported parameters. For this program, the correspondent IDF data source file "couetteS.dat" can be written as follows:

```
// IDF data file "couetteS.dat"
/* COUETTE, this structure contains input parameters for Couette problem */
```

```
COUETTE = {
/* Physical input parameters */
1.0  //Lz, A fixed plate separation distance, in cm
0.025 // Knu, Knudsen number
0.1    // Tw, Constant plate temperature, eV
0.01  // Uw, Constant relative plate velocity, U/Vt
      // Vt is the thermal velocity, sqrt(2RTw)
$1/3$, $2/3$ ,  1  // Pr, The set of Prandtl numbers
/* Computational algorithm input parameters */
1000  //Nz, number of nodes in the uniform mesh in z direction
31,31  //Nv, number of nodes in the 2D velocity space
};
```

The execution of this program results in displaying the same four strings as the output strings from the first sample program.

Usage of IDF functions in C++

For object-oriented programming the C++ language introduces an aggregate data type *class* which is the extension of *structure* data type in the C language. In this section we will consider an example of C++ program[19] which handles the data input with IDF package using the *class* mechanism. The C++ source file for the sample program consists of:

the header file references

```
#include <stream.h>
```

[19] We re-write the second sample program considered in the preceeding section from C to C++ language using the class *COUET* instead of structure *COUETTE*.

```
#define IDF_CPP 1
#define IDF_CPP_STYLE 1
#include "idfuser.h"
```

the *COUET* class definition

```
class COUET {
   int ierror;
  public:
   double Lz;
   float Knu,Uw,Tw;
   double Pr[3];
   int Nz,Nv[2];
      COUET(char*,char*);
  void COUETprn();
  int  COUETerr();
                  };
```

the class constructor

```
COUET :: COUET(char *Fname, char *Sname)
{
int amode = 7; // assume the 'natural boundary' rule
char Target_String[ ] = "dfffd[3]ii[2]"; //initialization of Target String
ierror=idf_init(); if(ierror) goto er; // start the IDF package
ierror=idf_open(Fname); if(ierror) goto err; // open the data file
// reading the data from file to the Cinp structure
ierror = idf_get(Sname, Target_String, amode, &Lz);
if(ierror>0) ierror=0;
err: idf_close(); // close the data file
er: idf_finish(); // finish the IDF
```

```
}
```

two class member functions, *COUETerr* and *COUETprn*,

```
int COUET :: COUETerr()
{
 return ierror; // transfer the error flag
}
void COUET :: COUETprn()
{
/* function prints the imported data sets*/
{
cout << form("Lz=%12.3e Knu=%12.3e\n",Lz,Knu);
cout << form("Uw=%12.3e Tw=%12.3e\n",Uw,Tw);
("Pr=%12.3e, %12.3e, %12.3e\n", Pr[0],Pr[1],Pr[2]);
cout << form("Nz=%5d Nv=%5d, %5d\n", Nz,Nv[0],Nv[1]);
}
```

and the program *main*

```
int main()
{
char File_Name[ ] = "couetteS.dat";  //initialization of input IDF data file String
char Class_Name[ ] = "COUETTE";  // initialization of structure data name
int ier;
// create the class object with the IDF based constructor
COUET Cinp(File_Name , Class_Name);
ier = Cinp.COUETerr();
if(!ier) Cinp.COUETprn();  // if no errors print input data
return ier;
}
```

In this C++ program, the constructor creates the object *Cinp* of the *COUET* class. The scientific parameters are the data members of this class. The constructor receives the IDF input file name "*couetteS.dat*" and the data set name "*COUETTE*" as the arguments. It uses then the IDF functions to initialize the data members of *COUET* class.

XVIII. IDF functions for the FORTRAN program

idfinit

This function initializes some global IDF variables and allocates memory used by other functions of IDF package. Usage in FORTRAN:

ierr = idfinit(0)

The function receives an integer type argument which has no meaning. It returns zero in success and the positive integer number *ierr* in the case of error.

idfinish

The *idfinish* subroutine sets to zero all global IDF variables and de-allocates the memory which has been previously reserved for IDF with the help of *idfinit* function. Usage in FORTRAN:

call idfinish

This subroutine is called without arguments.

idfopen

This function openes the specified IDF data source file for input. It also parsing the file and creates two catalogs (for static and external scope data)

of data names given in this file. Example of usage in FORTRAN:

character(*) Fname*

*integer*4 nlen, ierr*

parameter (len = 10 , Fname = 'file_name')

ierr = idfopen(Fname, len)

This function receives two arguments. The first argument, *Fname*, is the character string containing the name of the file to be opened. The second argument, *len*, is the number of characters in the file name. In success, the function returns zero, and the positive integer number *ierr* in the case of an error.

idfclose

The *idfclose* subroutine closes the IDF data source file (if any file has been opened with *idfopen* function). Usage in FORTRAN:

call idfclose

This subroutine is called without arguments.

idferprn

This subroutine prints the error message for any error associated with the memory allocation, with the opening, reading, and parsing the IDF data source file, and with the creation of catalogs for data names.

Usage in FORTRAN:

call idferrprn

This subroutine is called without arguments.

idfc

This function imports the named data set from a file for the single character target declared in the user-written program as **character**. Example of usage in FORTRAN:

character(*) name; character*1 Char*

*integer*4 nlen, ierr*

parameter (nlen = 10 , name = 'data_name')

ierr = idfc(name , nlen , Char)

The first argument of the function, *name* is the string containing the name of data set to be imported. The second argument, *nlen*, is the number of characters in the data set name. The third argument, *Char*, is the name of character variable declared in the user's code. The function returns zero in success and the positive integer number (i.e the error code *ierr*) in the case of error. If any error occurs, the function displays the error message.

idfi , idff , idfd , idfz , idfw

These functions import the named data set from a file for the single object target corresponding to the 'arithmetic' type **i,f,d,z,** and **w**, respectively. Example of usage in FORTRAN:

character(*) name*

*integer*4 nlen, ierr*

parameter (nlen = 10 , name = 'data_name')

*integer*4 ObjI; real*4 ObjF; real*8 ObjD; complex*8 ObjZ; complex*16 ObjW*

ierr = idfi(name , nlen , ObjI)

ierr = idff(name , nlen , ObjF)

ierr = idfd(name , nlen , ObjD)

ierr = idfz(name , nlen , ObjZ)

ierr = idfw(name , nlen , ObjW)

The first argument of a function, *name*, is the string containing the name of data set to be imported. The second argument, *nlen*, is the number of characters in the data set name. The third argument, *Obj_*, is the symbolic name of a target given in the user's code. Each function returns zero in success and the positive integer number *ierr* in the case of error. At an error, the function displays the corresponding error message.

idfs , idft

These functions import a named data set from the current file for the character string (**s**) and text (**t**) type targets, respectively. Example of usage in FORTRAN:

character(*) name*

*integer*4 nlen, ierr*

parameter (nlen = 10 , name = 'data_name')

*character*100 Str; integer*4 nelem=100;*

ierr = idfs(name , nlen , Str , nelem)

ierr = idft(name , nlen , Str , nelem)

The arguments of these functions are as follows: *name* is the string containing the name of data set to be imported; *nlen* is the number of characters in the data set name; *Str* is the symbolic name of a target declared as a character string in the user-written program; *nelem* is the maximal number of characters in the character string target *Str*. Each function returns zero in

success. In the case of error, the function returns a positive integer number *ierr* and displays the error message.

idfiarr , idffarr , idfdarr , idfzarr , idfwarr

These functions import the named data set from a file for the one dimensional array target corresponding to the type **i,f,d,z,** and **w**, respectively. Example of usage in FORTRAN:

character(*) name*

*integer*4 nlen, ierr*

parameter (nlen = 12 , name = 'array1D_name')

*integer*4 ArrI(3); real*4 ArrF(4); real*8 ArrD(2,3)*

*complex*8 ArrZ(5); complex*16 ArrW(7)*

ierr = idfiarr(name , nlen , ArrI , 3)

ierr = idffarr(name , nlen , ArrF , 4)

ierr = idfdarr(name , nlen , ArrD , 6)

ierr = idfzarr(name , nlen , ArrZ , 5)

ierr = idfwarr(name , nlen , ArrW , 7)

Each function receives four arguments, namely: *name* is the string containing the name of data set to be imported; *nlen* is the number of characters in the data set name; *Arr_* is the symbolic name of an array in the user-written program. The last argument is the smaller of the maximal number of elements in array or the number of array elements to be imported. The function returns the number *ierr* of imported elements in success. At an error, the function returns a negative number and displays the correspondent error message.

idfarray

This function imports a named data set from the current file for the multi-dimensional array target. Example of usage in FORTRAN:

character(*) name, TS*

*integer*4 nlen, nts, ierr*

parameter (nlen = 10 , name = 'array_name')

parameter (nts = 8 , TS = 'd(3,4,5)')

*real*8 Arr(3,4,5)*

ierr = idfarray(name, nlen , TS, nts , Arr)

The arguments are as follows: *name* is the string containing the name of data set to be imported; *nlen* is the number of characters in the data set name; *TS* is the Target String; *nts* is the number of characters in the Target String; *Arr* is the symbolic name of an array declared in the user-written program. The dimensionality of array must be specified using the target subscripts in the Target String. The subscripts can be also added (within a file) to the name of corresponding data set. In this case, the data set name subscripts denote the offset from the beginning of an array target (i.e the particular element in an array target starting with which the stored data should be imported). The function returns the number of imported elements in success. At an error, the function returns the negative number and displays the correspondent error message.

idfget

This function imports a named data set from the current file and transfer data to the arbitrary target. Example of usage in FORTRAN:

character(*) name, TS*

*integer*4 nlen, nts, align_mode*

parameter (nlen=12 , name = 'dataset_name')

parameter (nts=18 , TS = 'd(3,4,5)i(2)z(6,7)' , align_mode=0)

common/input/ Arr,Iarr,Carr

*integer*4 ierr, Iarr(2)*

*real*8 Arr(3,4,5); complex*8 Carr(6,7)*

ierr = idfget(name, nlen , TS, nts , align_mode, Atarget)

The function arguments have the following meanings: *name* is the string containing the name of data set to be imported; *nlen* is the number of characters in the data set name; *TS* is the Taget String; *nts* is the number of characters in the Target String; *align_mode* is the integer parameter (or variable) specifying the alignment rule; *Atarget* is either the symbolic name of the target (if target is the single variable or array), or the name of the first member in an aggregate target declared in the user-written program. In success the function returns the number of imported elements. In the case of error, it returns the negative number and displays the correspondent error message.

Example of IDF usage in FORTRAN program

This section illustrates the usage of IDF package for programming data input in FORTRAN language. Two simple sample programs written in

FORTRAN-77 are considered below.

The first program *couette* imports data from the file "couette.dat". This program is the FORTRAN-77 analog to the first **C** sample program considered in the previous section.

```
      program couette
c     The first sample program in FORTRAN
c     initialization of input IDF data file String
      character*(*) Fname
      integer*4 Flen, ier
      parameter (Flen = 11 , Fname = 'couette.dat')
c     declaration of input parameters
      real*8 Lz,Pr(3)
      real*4 Knu,Uw,Tw
      integer*4 Nz,Nv(2)
c     start the IDF package
      ier=idfinit(0)
      if(ier.ne.0) go to 1
c     open the data file
      ier=idfopen(Fname,Flen)
      if(ier.ne.0) go to 2
c     reading the data from file
      ier=idfd('Lz', 2, Lz)
      if(ier.ne.0) go to 2
      ier=idff('Knu', 3, Knu)
      if(ier.ne.0) go to 2
      ier=idff('Uw', 2, Uw)
      if(ier.ne.0) go to 2
      ier=idff('Tw', 2, Tw)
```

```
      if(ier.ne.0) go to 2
      ier=idfdarr('Pr', 2, Pr, 3)
      if(ier.ne.3) go to 2
      ier=idfi('Nz', 2, Nz)
      if(ier.ne.0) go to 2
      ier=idfiarr('Nv', 2, Nv, 2)
      if(ier.ne.2) go to 2
c     print the imported data sets
      print 10,Lz,Knu
10    format('Lz='E12.3 'Knu='E12.3)
      print 20,Uw,Tw
20    format('Uw='E12.3 ' Tw='E12.3)
      print 30,Pr(1),Pr(2),Pr(3)
30    format ('Pr='E12.3 ',' E12.3 ',' E12.3)
      print 40,Nz,Nv(1),Nv(2)
40    format('Nz='I5 'Nv='I5',' I5)
c     close the data file
2     call idfclose
c     finish the IDF
1     call idfinish
      stop
      end
```

The second FORTRAN program *couettes* transferes data from the file "couetteS.dat" to common block *COUETTE*, and all input parameters are the members of this block. The IDF input is performed within the user's function *couetinp*.

```
      program couettes
c     The second sample program in FORTRAN
```

```
c   initialization of input IDF data file String
    character*(*) Fname
    integer*4 Flen, ier, couetinp
    parameter (Flen = 12 , Fname='couetteS.dat')
c   declaration of input parameters within a common block
    common/COUETTE/Lz,Knu,Uw,Tw,Pr,Nz,Nv
    real*8 Lz,Pr(3)
    real*4 Knu,Uw,Tw
    integer*4 Nz,Nv(2)
c   import data with IDF package
    ier = couetinp(Fname,Flen)
    if(ier.eq.0) then
c   print the imported data sets
    print 10,Lz,Knu
10  format('Lz='E12.3 'Knu='E12.3)
    print 20,Uw,Tw
20   format('Uw='E12.3 ' Tw='E12.3)
    print 30,Pr(1),Pr(2),Pr(3)
30   format ('Pr='E12.3 ',' E12.3 ',' E12.3)
    print 40,Nz,Nv(1),Nv(2)
40   format('Nz='I5 'Nv='I5',' I5)
    end if
    stop
    end

    integer function couetinp(namef,lenf)
    character*(*) namef
    integer*4 lenf
c   function returns zero in success
    common/COUETTE/Lz,Knu,Uw,Tw,Pr,Nz,Nv
```

```
      real*8 Lz,Pr(3)
      real*4 Knu,Uw,Tw
      integer*4 Nz,Nv(2)
      integer*4 idfinit,idfopen,idfget
      integer*4 amode, lSN, lTS
c     initialization of Align Mode for F77 option '-align dcommon'
      parameter (amode=7)
c     initialization of Target String and Data Set Name
      parameter (lSN=7 , lTS=13)
      character*13 TS /'dfffd[3]ii[2]'/
      character*7 SN /'COUETTE'/
c     start the IDF package
      couetinp=idfinit(0)
      if(couetinp.ne.0) go to 1
c     open the data file
      couetinp=idfopen(namef,lenf)
      if(couetinp.ne.0) go to 2
c     transfer the data from file to common block
      couetinp = idfget(SN, lSN, TS, lTS, amode, Lz)
      if(couetinp) 2,3,3
3     couetinp=0
c     close the data file
2     call idfclose
c     finish the IDF
1     call idfinish
      return
      end
```

The *couettes* program imports the single named data set *'COUETTE'* from the "couetteS.dat" file (see the previous section) and prints the values

of input parameters.

XIX. Installing the IDF package

To install the IDF package on your computer you need, first of all, to create the object module file for each C source file of IDF package by invoking a C compiler. These modules should then be stored in an object module library. This library can then be linked to the user's code.

For UNIX platform, you may use the make file **idf.make**. This makefile contains additional comments which will help you to adjust commands to your operational system.

IDF package includes four header files, namely: "**idflib.h**", "**idf.i**", "**idf.h**", and "**idfusr.h**" [20]. In some cases, it is necessary to change macro-definition settings in "*idflib.h*", "*idf.i*" header files in order to adjust the IDF package to your computer, compiler, and program.

The IDF header file "**idflib.h**" contains a list of standard C header files[21] used in IDF. The *idflib.h* file also defines the macro IDF_CPP_STYLE which controls the declaration of C function prototypes[22]. The macro IDF_IEEE must be defined as unity, if your software implements the IEEE standard for binary floating-point arithmeic. You should also specify the proper type of

[20] The header file "**idfuser.h**" is created in running *idf.make*. This header file is a copy of "*idfusr.h*" file.

[21] In the case when the standard header files from this list does not exists (for example, "*float.h*" header file), follow instructions given in "*idflib.h*" in order to define standard macros required for IDF.

[22] Set IDF_CPP_STYLE to zero if you want to use Pre-Standard C declarations.

naming convention for C functions to be called from FORTRAN in terms of macro IDF_FORTRAN.

The IDF header file "**idf.i**" contains several macros important for IDF package performance. If you want to change the settings of these macros, follow comments given in this file.

APPENDIX I:

The List of IDF mathematical constants

Here is the list of names used by IDF for symbolic representation of several mathematical constants.

Name	**Value**
IDF_RAD	1 radian in degrees
IDF_DEG	1 degrees in radians
IDF_EU	Eu, Euler number
IDF_PI	π number
IDF_2PI	$2 * \pi$
IDF_PIPI	π^2
IDF_PIOVER2	$\pi/2$
IDF_PIOVER3	$\pi/3$
IDF_PIOVER4	$\pi/4$
IDF_1_PI	$1/\pi$
IDF_2_PI	$2/\pi$
IDF_EXP	exp(1)
IDF_EXP2	exp(2)
IDF_EXPI	exp(π)
IDF_EXPU	exp(Eu)
IDF_EXPE	exp(exp(1))

IDF_SEXP	$\mathrm{sqrt}(\exp(1))$
IDF_SQRT2	$\mathrm{sqrt}(2)$
IDF_SQRT3	$\mathrm{sqrt}(3)$
IDF_SQRT5	$\mathrm{sqrt}(5)$
IDF_SPI	$\mathrm{sqrt}(\pi)$
IDF_S2PI	$\mathrm{sqrt}(2 * \pi)$
IDF_LN2	$\ln(2)$
IDF_LN3	$\ln(3)$
IDF_LN10	$\ln(10)$
IDF_LNPI	$\ln(\pi)$
IDF_LNU	$\ln(\mathrm{Eu})$

APPENDIX II

The List of IDF physical constants

The following Table contains the list of names used by IDF for symbolic representation of several physical constants. The values of constants are given in Centimeter-Gram-Second (CGS) units system.

Name	**Symbol**	**Meaning**	**Value**
IDF_C	c	Speed of light in vacuum	2.99792458e10 cm/sec
IDF_H	h	Planck constant	6.6260196e-27 $erg\ sec$
IDF_HB	$\hbar = h/2\pi$	Planck constant	1.054592e-27 $erg\ sec$
IDF_K	k	Boltzmann constant	1.380658e-16 erg/K
IDF_K_EV	k	Boltzmann constant in eV	8.617385e-5 $k/e\ eV/K$
IDF_E	e	Elementary charge	4.806532e-10 $statcoul$
IDF_ME	m_e	Electron mass	9.1093897e-28 g
IDF_MP	m_p	Proton mass	1.6726231e-24 g
IDF_MPME	m_p/m_e		1836.152755
IDF_MEMP	m_e/m_p		5.446169971e-4
IDF_G	g	Gravitational constant	6.6732e-8 dyne cm^2/g^2
IDF_RY	R_y	Rydberg constant	109737.31534 cm^{-1}
IDF_RY_EV	R_y	Rydberg constant in eV	13.6056981 eV
IDF_RB	r_B	Bohr radius	0.529177249e-8 cm
IDF_CS	πr_B^2	Atomic cross section	0.8797356696e-16 cm^2

IDF_RE	r_E	Classical electron radius	2.8179e-13 cm
IDF_ALP	α	Fine-structure constant	7.297351e-3
IDF_IALP	$1/\alpha$	Inverse Fine-structure constant	137.0360
IDF_MUB	μ_B	Bohr magneton	5.78838263e-5 eV/$Tesla$
IDF_FA	$F = eN_A$	Faraday constant	2.892599e14 $statcoul/mol$
IDF_CW	$\hbar/m_e c$	Compton electron wavelength	3.861592e-11 cm
IDF_CR1	$8\pi hc$	First radiation constant	4.992579e-15 $erg\ cm$
IDF_CR2	hc/k	Second radiation constant	1.438833 $cm\ K$
IDF_S	σ	Stefan-Boltzmann constant	5.66961e-5 $erg/cm^2/sec/K^4$
IDF_NA	N_A	Avogadro number	6.022169e23 mol^{-1}
IDF_R	$R = kN_A$	Gas constant	8.31434e7 $erg/deg/mol$
IDF_TO	T_o	Standard temperature	273.15 K
IDF_PO	$P_o = LokT_o$	Atmospheric pressure	1.0133e6 $dyne/cm^2$
IDF_LO	Lo	Loschmidt's number	2.6868e19 cm^{-3}
IDF_VO	$V_o = RT_o/P_o$	Normal volume perfect gas	2.24136e4 cm^3/mol

APPENDIX III

Mathematical functions of IDF calculator

The following mathematical functions are used in the IDF built-in calculator[23]. The names of these functions are the *generic* names, that is a user refers by common name to the number of functions. For each particular function call, the IDF calculator dynamically selects the function, the type of dummy arguments of which matches the type of actual arguments. In this case, not only the returning value but also the data type returning by the function depends on the type of an actual arguments passed to this function.

cmplx(re)

This function creates a complex number a, whose real part is equal to re and whose imaginary part is set to zero, i.e $a = (re, 0)$. The function argument is a real number, whereas the returning value is aways a complex number.

complex(re,im)

The function creates a complex number a specified by its real, re, and imaginary, im, parts. The numbers, re and im can be associated with coordinates of a point at complex plane. That is why, the following form of complex

[23]The names of some functions listed in this APPENDIX match the names of standard functions specified in C and FORTRAN languages. At this point, the IDF mathematical functions are not the same functions as in C or FORTRAN, although their execution require standard mathematical functions from the C run-time library, namely: *sqrt, exp, log, pow, cos, sin, tan, acos, asin*, and *atan*.

number representation: $a = re + i\,im = (re, im)$, where i is the imaginary unit ($i^2 = -1$), is frequently referred to as the Cartesian form. The function returns the complex number composed from real numbers, re and im.

real(a)

If the argument a is the real number, the function returns the value of argument (converting an integer to the floating-point number, if necessary). If the argument a is the complex number, $a = x + iy$, the function returns the real part, i.e $real(a) = x$.

double(a)

If the argument a is the real number, the function returns the value of argument converting an integer to the floating-point number (type double), if necessary. If the argument a is the complex number, $a = x + iy$, the function returns the real part, i.e $double(a) = x$.

imag(a)

If the argument a is the real number, the function returns zero. If the argument $a = x + iy$ is the complex number, the function returns the imaginary part, i.e $imag(a) = y$.

conj(a)

If the argument a is the real number, the function returns the value of its argument, but with opposite sign, i.e $(-a)$. If the argument a is the complex number, $a = x + iy$, the function returns the complex conjugate, i.e $complex(x, -y)$.

abs(a)

If the argument a is the real number, the function calculates the absolute value of a. In the case when the argument is the complex number $a = x + iy$, the function computes the modulus or radius of a complex number, i.e $\rho = sqrt(x^2 + y^2)$. The function returns then the value of ρ which is always the real number.

arg(a)

The argument of this function must be the complex number represented in Cartesian form: $a = x + iy$. Any complex number can be re-written in the polar form: $a = \rho * [cos(\phi) + i\ sin(\phi)]$, where ρ is the radius and ϕ is the angle or argument of complex number at the complex plane. Since the trigonometric functions are periodic functions, only the principle value arg of an argument ϕ, $0 \leq arg < 2\pi$, is usually used. In the special case when $a = (0, 0)$, $arg(a) = 0$, otherwise the principle value is calculated as: $arg(a) = acos(x/\rho)$ if $y \geq 0$, and $arg(a) = -acos(x/\rho)$ if $y < 0$. The arg function returns the principle argument correspondent to complex number a. The returning value is always the real number.

polar(ρ,Arg)

The function creates the complex number a specified by its modulus ρ and principle argument Arg, i.e $a = \rho * cos(Arg) + i\ \rho * sin(Arg)]$, The arguments of this function are real numbers, whereas the returning value is always a complex number.

int(re)

This function convert the argument to the integer number by truncation the floating-point number towards zero. The argument must have integer or floating-point type, whereas the returning value is always the integer number. The complex argument is not allowed in this function.

sqrt(a)

For any real argument a, the $sqrt$ function returns the square root of a. The complex square root of complex argument $a = x + iy$ is defined as follows:

$$\sqrt{a} \;=\; sqrt(abs(a)) \;*\; [cos(arg(a)/2) \;+\; i\; sin(arg(a)/2)].$$

For the branching point, i.e when $a = (0, 0)$, the function returns complex zero. In all other complex cases, the function returns the complex number:

$$\sqrt{a} = complex(g, \frac{y}{2g}),$$

where $\rho = abs(a) = sqrt(x^2 + y^2)$ is the modulus of a and the parameter is calculated in the following way: $g = sqrt[(\rho + x)/2]$ if only $x \geq 0$, otherwise $g = sqrt[(\rho - x)/2]$ for $y \geq 0$ and $g = -sqrt[(\rho - x)/2]$ for $y < 0$.

sqrt1pz2(a)

This function evaluates the expression $sqrt(1 + a^2)$ for any real argument a. When applied to complex numbers $a = x + iy$, this expression results in: $sqrt(\; complex(\; 1 + x^2 - y^2 \;,\; 2xy \;)\;)$.

sqrt1mz2(a)

This function evaluates the expression $sqrt(1 - a^2)$ for the real argument $a \leq 1$. When applied to complex numbers $a = x + iy$, this expression results in: $sqrt(\ complex(\ 1 - x^2 + y^2\ ,\ -2xy\ )\ )$.

max(a,b)

This function returns the maximum of two real numbers a and b. If only one of the arguments is the complex number, the other argument will be converted to the complex number with the zero imaginary part. In comparison of complex numbers a and b, the function chooses the number amongst them, whose modulus is the biggest. If $abs(a) = abs(b)$, the function returns the number whose principle argument is the biggest.

min(x,y)

This function returns the minimum of two real numbers a and b. If only one of the arguments is the complex number, the other argument will be converted to the complex number with the zero imaginary part. In comparison of complex numbers a and b, the function chooses the number with the smallest modulus. If $abs(a) = abs(b)$, the function returns the number whose principle argument has the smallest value.

pow(a,b)

This is the generalized power function. For real arguments, the function raises a to the b power. According to C language standards, the *pow* function returns: the zero value, if $a = 0$ and b is a positive number; the unity value, if $a = b = 0$; the unity value, if a is positive and $b = 0$; the a^b value for arbitrary b, only if a is positive; the a^b value, if a is the negative number and

b is the integer number. It is not allowed to exponentiate the negative base a to the non-integer power, or the zero base to the negative power. When applied to the complex numbers, the function returns the complex number which is defined and computed as:

$$pow(a, b) \;=\; exp(\; b * log(a)\;).$$

exp(a)

This is the natural exponential function which for real numbers raises e to the a power. In the case when function argument is the complex number $a = x + iy$, the function returns the complex number defined as:

$$exp(a) \;=\; exp(x) \;*\; [cos(y) \;+\; i\; sin(y)].$$

pow10(a)

This function raises the base 10 to the a power for any real or complex argument. The function calculates 10^a as follows: $10^a \;=\; exp(\; a * log(10)\;)$, using the natural exponential function $exp(a)$ and the constant value $log(10)$.

log(a)

This is the natural logarithm function. For a real argument, it returns the natural logarithm of a. The complex natural logarithm of the complex argument $a = x + iy$ is evaluated as follows:

$$log(a) \;=\; log[abs(a)] \;+\; i\; arg(a).$$

log10(a)

This is the common logarithm function. The function calculates logarithm to the base 10 of its argument a as: $log10(a) = log(a)/log(10)$, using the natural logarithm function $log(a)$ and the constant value $log(10)$.

cos(a)

For real numbers, this is the standard trigonometric function. The function calculates the cosine of its argument. The real number argument is expressed in radians. When applied to complex numbers, the cosine function is defined as follows:

$$cos(a) = [exp(ia) + exp(-ia)]/2.$$

The complex cosine is calculated for $a = x + iy$ as:

$$complex(\ cos(x) * cosh(y)\ ,\ -sin(x) * sinh(y)\ ).$$

sin(a)

For real numbers, this is the standard trigonometric function. The function calculates the sine of angle a specified in radians. The complex sine function is introduced as:

$$sin(a) = i\ [exp(-ia) - exp(ia)]/2.$$

For $a = x + iy$, the complex sine is calculated as follows:

$$complex(\ sin(x) * cosh(y)\ ,\ cos(x) * sinh(y)\ ).$$

tan(a)

For real numbers, this is the standard trigonometric function. It calculates the tangent of angle a specified in radians. The complex tangent function is defined as follows:

$$tan(a) \;=\; i\;[exp(-i2a) - 1]\;/\;[exp(-i2a) + 1].$$

For the argument given in Cartesian form, $a = x + iy$, the complex tangent is calculated as follows:

$$complex(\; sin(2x)/g,\; sinh(2y)/g\;),$$

where $g = cos(2x) + cosh(2y)$.

acos(a)

When the argument is real, the *acos* function returns the angle, expressed in radians, whose cosine is equal to a, $-1 \leq a \leq 1$. When applied to complex numbers, the arc cosine function is introduced as:

$$acos(a) \;=\; -i\;log[a \;+\; i\;sqrt(1 - a^2)].$$

asin(a)

The *asin* function calculates the inverse sine of argument a. For real argument $-1 \leq a \leq 1$, the arc sine function returns the correspondent angle in radians. The complex arc sine function is defined as:

$$asin(a) \;=\; -i\;log[ia \;+\; sqrt(1 - a^2)].$$

atan(a)

The $atan$ function calculates the inverse tangent of argument a. For any real argument, the arc tangent function returns the correspondent angle in radians. The complex inverse tangent function is defined as:

$$atan(a) \;=\; -0.5\; i\; log(\frac{1+ia}{1-ia}).$$

For $a = x+iy$, the expression under logarithm results in the complex number:

$$complex(\; [1-x^2-y^2]/g \;,\; 2x/g \;) \quad ,$$

where $g = x^2 + (1+y)^2$.

cosh(a)

This is the hyperbolic cosine function:

$$cosh(a) \;=\; [exp(a) \;+\; exp(-a)]/2,$$

and for real argument a, the function returns the value given by this expression. The complex hyperbolic cosine of argument $a = x + iy$ is calculated as follows:

$$complex(\; cosh(x) * cos(y) \;,\; sinh(x) * sin(y) \;).$$

sinh(a)

The hyperbolic sine function $sinh$ is defined as:

$$sinh(a) \;=\; [exp(a) \;-\; exp(-a)]/2.$$

This expression is used to evaluate $sinh$ for real argument a. When applied to complex numbers $a = x + iy$, the complex hyperbolic sine is calculated as:

$$complex(\; sinh(x) * cos(y) \;,\; cosh(x) * sin(y) \;).$$

tanh(a)

This is the hyperbolic tangent function. For any real argument a:

$$tanh(a) \;=\; \pm\frac{1 - exp(-2|a|)}{1 + exp(-2|a|)},$$

where $|a|$ is the absolute value of argument, and $\pm$ is the sign of a. For complex numbers, the hyperbolic tangent is defined as:

$$tanh(a) \;=\; -i\ tan(ia).$$

For any $a = x + iy$, the $tanh$ function can be evaluated as follows:

$$complex(\ sinh(2x)/g,\ sin(2y)/g\ ),$$

where $g = cos(2y) + cosh(2x)$.

acosh(a)

This function is the inverse hyperbolic cosine of real argument $a > 1$. The inverse hyperbolic cosine of complex number $a = x + iy$ is given by the following expression:

$$acosh(a) \;=\; log[a \;+\; sqrt(a^2 - 1)].$$

asinh(a)

This function is the inverse hyperbolic sine of any real argument a. The complex inverse hyperbolic sine of $a = x + iy$ is defined by the expression:

$$asinh(a) \;=\; log[a \;+\; sqrt(a^2 + 1)].$$

atanh(a)

The function calculates the inverse hyperbolic tangent of real argument $-1 < a < 1$. The complex inverse hyperbolic tangent is defined as:

$$atanh(a) \;=\; 0.5\; log(\frac{1+a}{1-a}).$$

For $a = x+iy$, the expression under logarithm results in the complex number:

$$complex(\; [1-x^2-y^2]/g \;,\; 2y/g \;) \quad ,$$

where $g = (x-1)^2 + y^2$.

hypot(x,y)

The *hypot* function is defined only for real numbers. It calculates and returns the following value: $sqrt(x^2 + y^2)$, which is, historically, the length of hypotenuse in rectangular triangle. The arguments, x and y, as well as returning value are always real numbers.