The TYPES Users Guide:
A Data Abstraction Package in FORTRAN

Version 1.0

Saul Youssef
Supercomputer Computations Research Institute
Florida State University
Tallahassee, FL 32306-4052

FSU-SCRI-90-03

ABSTRACT

Types is a collection of Fortran programs which allow the creation and manipulation of abstract "data objects" without the need for a preprocessor. Each data object is assigned a "type" as it is created which implies participation in a set of characteristic operations. Available types include scalars, logicals, ordered sets, stacks, queues, sequences, trees, arrays, character strings, block text, histograms, virtual and allocatable memories. A data object may contain integers, reals, or other data objects in any combination. In addition to the type specific operations, a set of universal utilities allows for copying, input/output to disk, naming, editing, displaying, user input, interactive creation, tests for equality of contents or structure, machine to machine translation or source code creation for any data object. This document is the users guide for the Types package.
# TABLE OF CONTENTS

1. INTRODUCTION ........................................... 1
2. BASIC CONCEPTS ............................................ 3
3. APPLICATIONS AND EXAMPLES ............................... 8
   3.1. The Allocatable Memory Type .......................... 8
   3.2. Manipulating a Simple Data Object .................... 9
   3.3. Error Checking and Debugging .......................... 10
   3.4. Example: Removing Duplicates from a Set ............... 11
   3.5. Using Types Utilities for User Interfaces .......... 13
   3.6. Moving Data Objects from Computer to Computer .... 14
   3.7. Machine Independent Help Systems .................... 15
   3.8. The Virtual Memory Types ............................. 15
   3.9. Using Types for Storing Benchmarks .................. 16
4. TUTORIAL ............................................... 18
5. AVAILABLE TYPES AND UTILITIES ....................... 27
   5.1. Primitive Types ...................................... 27
      5.1.1. pUndefnd ........................................ 27
      5.1.2. pScalar ......................................... 27
      5.1.3. pBScalar ........................................ 27
      5.1.4. pOrSet .......................................... 27
      5.1.5. pStack .......................................... 28
      5.1.6. pChaStr ......................................... 28
      5.1.7. pMemory .......................................... 28
      5.1.8. pText ............................................ 28
      5.1.9. pVeArray ....................................... 28
      5.1.10. pLogical ....................................... 28
      5.1.11. pSequenc ....................................... 28
      5.1.12. pHistGrm ....................................... 29
      5.1.13. pQueue ......................................... 29
      5.1.14. pVirMem ....................................... 29
   5.2. General Types ....................................... 29
5.2.1. dOrSet ........................................... 29
5.2.2. dStack ........................................... 29
5.2.3. dTree ............................................. 29
5.2.4. dStArray ......................................... 29

5.3. Display and Editing Utilities .................................... 30
  5.3.1. dCreate ......................................... 30
  5.3.2. dInfo ........................................... 30
  5.3.3. dSkeletn ......................................... 30
  5.3.4. dDisplay ........................................ 30
  5.3.5. dEdit ........................................... 30

5.4. Input/Output utilities ........................................ 30
  5.4.1. dWrite .......................................... 30
  5.4.2. dWriteQ ......................................... 30
  5.4.3. dRead ........................................... 30
  5.4.4. dReadA ........................................... 30
  5.4.5. dReadQ ........................................... 30
  5.4.6. dAlocIO .......................................... 31
  5.4.7. dOpenNew ......................................... 31
  5.4.8. dOpenOld ......................................... 31
  5.4.9. dClose ........................................... 31
  5.4.10. dUWrite .......................................... 31
  5.4.11. dURead .......................................... 31
  5.4.12. dUReadA .......................................... 31
  5.4.13. dReadTry ......................................... 31
  5.4.14. dSource .......................................... 31
  5.4.15. dToExch .......................................... 32
  5.4.16. dFrExch .......................................... 32

5.5. Utilities for Extracting Information about Data Objects .......... 32
  5.5.1. dSpace ........................................... 32
  5.5.2. dUSpace .......................................... 32
  5.5.3. dSpLeft ........................................... 32
  5.5.4. dHeadSz ........................................... 32
  5.5.5. dType ............................................ 32
5.5.6. dPrim ........................................... 32
5.5.7. dPrimTyp ...................................... 32
5.5.8. dTyNmGet ..................................... 33
5.5.9. dDsNmGet ..................................... 33
5.5.10. dTypeNam .................................... 33

5.6. Miscellaneous Utilities .......................... 33
5.6.1. dName ......................................... 33
5.6.2. dCopy ......................................... 33
5.6.3. dEqual ........................................ 33
5.6.4. dCongru ....................................... 33
5.6.5. dUnion ........................................ 33
5.6.6. dCheck ........................................ 33
5.6.7. dCheckQ ....................................... 33
5.6.8. dChecAll ...................................... 33
5.6.9. dSetType ...................................... 34

6. REFERENCES ....................................... 35
1. INTRODUCTION

In advanced programming languages, data structures can be created with an abstract data "type" which implies participation in a characteristic set of operations. For example, a "stack" can participate in "push" or "pop" operations, an "ordered set" can have things put into the set or taken out of the set. In the case of an ordered set, the meaning of the set operations is independent of the contents of the set. This separation of the abstract properties of an object from its contents ("data abstraction") is characteristic of all types and is absolutely essential for solving difficult programming problems. Data abstraction is also necessary even to specify difficult programming problems. The purpose of the Types package is to provide a general mechanism for creating and manipulating abstract data objects[1] which fits naturally into standard Fortran.

Although Types is not oriented towards any particular discipline or programming situation, I will occasionally use examples from experimental High Energy Physics to be specific. It is not difficult to construct analogous examples in many different fields. In High Energy Physics, concepts like "event," "drift chamber track," "calorimeter cluster," "Z-candidate" etc. allow programming problems to be concisely specified. For example, problems like "sort a set of drift chamber tracks in increasing energy," or "remove duplicate calorimeter clusters" would be much more difficult to specify in terms of integers, reals and logicals. In addition, these problems are easily solved at the level of abstract set operations, but are more difficult at a lower level of abstraction. Types allows the solution to these sorts of problems by providing a mechanism for creating data objects to represent events, drift chamber tracks, etc. together with a collection of subroutines which perform the appropriate abstract operations.

Although there are many languages which support data abstraction, programmers are often constrained to write in simple languages like Fortran or C which are inadequate for creating and manipulating data objects. Thus, programmers are often forced into the tedious and error prone task of translating abstract programs into low level executable code. The Types package can alleviate these problems by "simulating" advanced language features in Fortran. In addition, there are features of Types which are difficult to find in other languages, for example, the universal utilities allowing copying, naming, input/output to disk, editing, displaying, user interface, interactive creation, tests for equality of contents or structure, machine to machine translation or source code creation for any data object. Such utilities are extremely useful since they reduce much of the tedious work associated with programming.
Types is a set of Fortran programs[2] and subprograms which implement a general mechanism for data abstraction without the need for a preprocessor. Types is designed so that pieces of the package can be used in otherwise normal style Fortran programs, or in programs in other languages which can call Fortran programs. For example, it might be desirable to use just one or two of the types or to use the allocatable memory or to just use the virtual memory type to store large amounts of data. Types is not intended to be a closed system in that we expect new types to appear for special application areas. It is intended to be easy add new operations to existing types or to create new types.

Types is currently available on VAX/VMS, SUN/(BSD Unix), ETA-10/(Unix System V) and Silicon Graphics 4D/(Unix System V). We anticipate supporting Types on other machines as needed including Cray/Unicos, VAX/Ultrix and NeXT/Mach. If you want a copy of this users guide or the types source code and help system or if you want to be on the Types mailing list, send a message to S. Youssef, Supercomputer Computations Research Institute, Florida State University, Tallahassee, Florida 32306-4052, internet: youssef@scril.scri.fsu.edu, DECNET: 47291::youssef.
2. BASIC CONCEPTS

The data types available in Fortran are integers, reals, logicals and character strings which can all be elements of arrays. In addition, Types includes a set of "primitive" types which have ordinary integers or reals as components. An ordered set of reals, a character string, a stack of integers and a block text are all examples of primitive types. The integer or real types are called "scalars." Each primitive type has a name which begins with "p." For example, the names of the ordered set of scalars, the character string, the stack of scalars and the block text are "pScalar," "pChaStr," "pStack," and "pText" respectively. Information about each of the types and associated operations is available through the help system (see the tutorial). In addition to primitive types, Types also includes a set of "general" types which can have other data objects as components and which have names beginning with "d." Examples of general types are the ordered set of data objects (dOrSet), the stack of data objects (dStack), the tree of data objects (dTree) and the array of data objects (dStArray).

Any contiguous set of memory locations, either in local variables or in COMMON blocks, can be assigned a type with a subroutine call which has the same name as the name of the type. For example, the following creates a dStack in an integer Fortran array S.

```
INTEGER S(100)
CALL dStack (100,S)
```

S(1) ... S(100) is now a data object of the type 'dStack' which can be referred to simply by the starting address 'S.' By virtue of S's new type, S can participate in the stack operations push (dStPush) and pop (dStPop). For example, the following code fragment pushes data object A onto dStack S and then pops S, putting the result in data object B.

```
CALL dStPush (A, S)
CALL dStPop (S, B)
```

Notice that the operation of pushing A onto S is independent of the type or contents of A. The fact that S is an integer array has no effect on S. S could equally well be declared as "REAL S(100)." It is often convenient to create data objects which exist in allocatable memory rather than in local variables. The allocatable memory is described in section 3.1.

Since there are a large number of type operations, a systematic naming convention is used where the first character of the type operation indicates a primitive or general type, and the next two or three characters indicate the type. For example, the available operations on the ordered set of data objects are: dOrSet, dOrIni, dOrEmp, dOrAdd, dOrTran,
There are some primitive types which are closely analogous one of the general types. For example, in addition to the stack of data objects (dStack) above, a separate type called "pStack" is a stack of integers or reals. For example, the following code fragment creates a pStack and pushes a real number onto it.

```fortran
INTEGER T (100)
CALL pStack (100,T,'REAL')
CALL pStPush (3.14159, T)
```

Primitive types like pStack and pOrSet (the ordered set of scalars) must include a specification of their components at the time of creation. Here the 'REAL' argument causes T to be created as a pStack of Fortran reals.

In addition to operations defined by the type of a data object, a set of universal utilities operate on any data object. These utilities always start with "d." Universal utilities include interactive creation of data objects (dCreate), displaying a data object (dDisplay), copying one data object to another (dCopy), input/output to disk (dWrite/dRead), naming a data object (dName), editing a data object (dEdit), testing two data objects for equality of contents or structure (dEqual, dCongru), user interfaces (dInput, dEdit, dReadTry), machine to machine translation (dToExch, dFrExch) and creating the source code equivalent of a data object (dSource). For example, the following call to dName assigns the character string "My Name" as the name of data object Q:

```fortran
CALL dName (Q,'My Name')
```

The name of a data object is stored inside the data object and can be up to 32 characters long. It is often useful to assign names to data objects so that utilities like dInput, dDisplay and dEdit can function usefully. For example, a utility like dDisplay uses the names, types and contents of a data object to make a non-graphics display of structure and contents. A display of any data object Q is produced simply by

```fortran
CALL dDisplay(Q)
```

Similarly, any data object can be created in an interactive session by

```fortran
CALL dCreate(Q)
```

Utility subroutines like dDisplay and dCreate also exist as stand alone programs which read data objects from a file for input. The command 'ddisplay', for example, causes a
display to be produced for a data object which has been stored in a file by, for example, the utility dWrite. Figure 1 shows the result of dDisplay for a data object stored in a file. In the standard display from dDisplay, each data object has a header of the form '/data object name/type/type of components.' Components of the displayed data object "GEANT Volume" are shown below the main header and indented. Thus, "GEANT Volume" is an ordered set of data objects (dOrSet) with the four components "GEANT Volume Shape," "Shape Parameters," "Displacement x,y,z(cm.)" and "Rotation Parameters." Note that the "rotation matrix," pOrSet of reals has type "pUndefnd." pUndefnd is the "undefined" data type which is useful in various situations. Here it is being used to suppress user input for each component of a rotation matrix when using a user interface utility like dInput. Later on, a Fortran program can calculate the rotation matrix from the rotation angles and insert this in place of the pUndefnd data object.

The space available in a data object is defined at the time of creation. In Types version 1.0, there is no mechanism for increasing the size of an existing data object other than by copying it into a larger data object. This means that an upper bound on the size of a data object must be known when it is created. To assist in this, Section 6 contains a brief discription of each type along with information about how much space is needed. It is often easy to get an upper bound on the amount of space needed for a data object. For example, if data object S is to hold components of an ordered set SET, then it is often convenient just to create S with the same amount of space as SET. S can be created as an allocated data object (see section 3.1) and then deallocated when it is no longer needed.

The contiguous memory of a data object contains a complete description of the data object including its type, size, name, amount of free space and all of its components whether they are integers and reals or other data objects. All data objects begin with a “header” of at most 21 integer sized words which has the same format for all types. Following the header, data objects have different, implementation dependent formats. The format of the universal header is given in the appendix. It is never necessary to know the internal structure of a data object to write and debug programs using Types. It is highly recommended to only operate on data objects through the type operations or utility routines.

The use of the types package implies a certain amount of overhead in memory and processor time. Each type operation implies at least one subroutine call and a simple check to insure that the argument is a legal data object of the proper type. The significance of this extra computation depends on how much work is done inside the subroutine. In the case of fetching the value of a scalar, the overhead is certainly substantial, but in a more
$ display$
= Enter filename:
box.gv

==============================================

/GEANT Volume/dOrSet/COMPOSITE

/GEANT Volume SHAPE/pChaStr/CHARACTER

"BOI"

/SHAPE parameters/pOrSet/REAL

<table>
<thead>
<tr>
<th>Element #</th>
<th>Value</th>
<th>3 Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.100000000E+02</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.100000000E+02</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.100000000E+02</td>
<td></td>
</tr>
</tbody>
</table>

/Displacement x,y,z(cm.)/pOrSet/REAL

<table>
<thead>
<tr>
<th>Element #</th>
<th>Value</th>
<th>3 Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.000000000E+00</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.000000000E+00</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.000000000E+00</td>
<td></td>
</tr>
</tbody>
</table>

/Rotation parameters./dOrSet/COMPOSITE

/theta1,phi1,...,theta3,phi3(deg.)/pOrSet/REAL

<table>
<thead>
<tr>
<th>Element #</th>
<th>Value</th>
<th>6 Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.900000000E+02</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.000000000E+00</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.900000000E+02</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.900000000E+02</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.000000000E+00</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.000000000E+00</td>
<td></td>
</tr>
</tbody>
</table>

/rotation matrix/pUndefnd/UNDEFINED

- Undefined -
typical operation involving many words of data the fractional time spent in the subroutine call can become negligible. Compilers with in-line subroutine expansion and computers supporting fine grained parallel or pipelined operations can further decrease processor overhead. A similar situation exists with respect to memory overhead. The fact that a scalar may take up to twenty-one words of memory is not a serious limitation because programs do not use much of their memory storing integer or real scalars. In addition, the allocatable memory type (section 3.1) and the virtual memory types (section 3.7) are powerful ways of reducing potential memory problems.
3. APPLICATIONS AND EXAMPLES

There follows a collection of illustrative applications which would be difficult without the ability to create and manipulate abstract data objects.

3.1. The Allocatable Memory Type

Almost all Fortran compilers allocate permanent space for local variables. This makes it essential to have a system of allocatable memory when writing large applications. The usual solution to this problem is to introduce a large vector (e.g. ‘W’) in a common block and allocate space inside W. An allocation operation might return a starting point (start) and a length (len) meaning that W(start), ..., W(start + len - 1) are available for use. Although this is an annoying complication in syntax, a more serious problem is that a mistake, such as altering W(start + len), is an error which can be very difficult to track down, even assuming that you are lucky enough to find out about its existence. The Types package includes a type called ‘pMemory’ which is a more robust solution for allocatable data objects.

Just as with any other type, any contiguous memory can be made into a pMemory. As a convention, however, a global common block /TypeMem/ is provided which contains a large array ‘o’ initialized as a pMemory. The net affect of using allocated data objects is that rather than having data objects A, B and C existing in, for example local arrays A(1000), B(1000) and C(1000), A, B and C could be ordinary integer variables and represent allocated data objects referred to as ‘o(A),’ ‘o(B)’ and ‘o(C)’ respectively. In general, ‘o(something)’ is the most complicated reference to the allocatable memory that is necessary. It is never necessary to keep track of the length of o(A) or to make sure that other parts of the allocatable memory are not corrupted as this is all automatically taken care of by the Type operation routines.

To illustrate a typical allocation, the following Fortran code allocates a 1000 word data object called ‘Track’:

```fortran
INTEGER Track
CALL pMemAloT(o, 1000,Track,'dOrSet')
```

The call to pMemAloT allocates 1000 words from pMemory o for data object Track and initializes Track to be an empty ordered set. A similar call

```fortran
CALL pMemAloc(o, 1000,Track)
```

would initialize Track to be an ‘undefined’ data object (type = ‘pUndefnd’). In both of these cases, the allocated data object is simply referred to as ‘o(Track)’ rather than
‘Track’. o(Track) is deallocated by a call to pMemDalc(o, Track). Data objects must be deallocated in reverse order – last allocated, first deallocated. Often, data objects are allocated with an amount of space which depends on the size of other data objects. There are pMemory routines to assist in this, for example, pMemSSiz(o, A,Q) allocates a data object o(Q) with the same amount of space as data object A. For examples of the use of pMemory, see section 3.3, the tutorial and the “pMemory” entry in the help system.

3.2. Manipulating a Simple Data Object

To give a concrete example of the ideas so far, consider the case of a drift chamber track as might occur in a High Energy Physics application. A drift chamber track might be an ordered set

\[
\text{Track} = (\text{Vertex}, \text{Momentum}, \text{PName}, \text{TrackOk})
\]

where Vertex is the starting point of the track (a pOrSet of three reals), Momentum is the three component momentum vector for the particle (another pOrSet of three reals), PName is a character string giving the name of the particle type (a pChaStr) and TrackOk is a logical value (a pLogical) indicating some “goodness” quality of the track. Supposing that Track already exists, we also suppose that it is to be manipulated inside a subroutine ‘TrackSub.’ If TrackSub needs only the particle Momentum, then the following code fragment might occur:

```fortran
SUBROUTINE TrackSub(Track)
INTEGER Track(*)
INTEGER Momentum(50)
CALL pUndefn(50, Momentum)
CALL dOrGet(2, Track, Momentum)
```

where the call to “dOrGet” gets the second element of Track and puts it in Momentum. pUndefn is called before dOrGet in order to turn the array Momentum into a data object. As in many other situations, the output argument of dOrGet must be a data object but it’s type is irrelevant. The exact requirements of the arguments to dOrGet are available from the help system. This code fragment has the disadvantage that an upper bound on the size of Momentum needs to be known in order to know that 50 words is sufficient (of course, if it is not sufficient, a sensible error message results). This problem can be avoided if a variant on dOrGet (called dOrGetA) is used which creates an allocated data object to store a particular element of the input dOrSet. In the following, the second component of Track is put into a data object which could then be referred to as ‘o(Momentum).’

```fortran
SUBROUTINE TrackSub (Track)
INTEGER Track (*)
```
INTEGER Momentum
INCLUDE 'Types$:TypeMem.inc'
CALL dOrGetA (0, 2, Track, Momentum)

The include file in the code above contains the allocatable memory ‘o.’ In this example, o(Momentum) is automatically allocated with the right amount of space to hold the second component of Track.

There is yet another way to get the second component of Track and that is to refer to components of Track by name. As mentioned, any data object can be assigned a name with the dName utility. If we suppose that the names of the components of Track are "Vertex," "Momentum," "Particle Type Name" and "Track is OK" respectively, then, for example, the following can be used to fetch the "Track is OK" component of track:

CALL dOrGetNm (Track, 'Track is OK', TrackOK)

If it is necessary to fetch all of the components of Track, then dOrLet can be used as in

CALL dOrLet4(Track, Vertex, Momentum, PName, TrackOK)

where the "Let" in dOrLet is meant to suggest the mathematical statement: "Let Track = (Vertex, Momentum, PName, TrackOK)," and where the "4" in the subroutine name indicates that there are four output arguments. As with dOrGet, there is an analogous operation, called dOrLtA, which fetches all of the components of Track and stores the components in allocated data objects. If allocated data objects are used, it is important to remember to deallocate them before the end of the subroutine and in last allocated–first deallocated order (see the next section). For example, if

dOrLtA4(0, Vertex, Momentum, PName, TrackOk)

is used to create o(Vertex), o(Momentum), o(PName) and o(TrackOk), then

dOrLtD4(Vertex, Momentum, PName, TrackOk)

must be called before the end of subroutine TrackSub to deallocate the new allocated data objects.

3.3. Error Checking and Debugging

The Types package is implemented with extensive run time error checking. Each operation on data objects in Types includes a check that data object inputs are "legal" (i.e. a special word is checked) and a check that input data objects have the proper types for the attempted operation. Each operation which adds data to a data object is preceded by a check that there is enough space inside the data object to perform the operation. These checks, combined with the integrity of the allocatable memory allow detection of errors which would otherwise go undetected or occur as a more obscure system error.
In debugging programs which use data objects, one must avoid the situation where the author is required to know and examine the internal structure of the data objects. Fortunately, this is easy to avoid if some of the utility programs are included in the program image (e.g. dInfo, dSkeletn, dDisplay, dInput, dCreate, dCopy, dWriteQ, dReadQ). With the utility routines available inside a symbolic debugger, it is easy to use these routines rather than examining individual words inside of data objects. The utility routines also make it easy to perform certain debugging checks which would be difficult otherwise. For example, if a complicated data object Q is being incorrectly modified, then one can copy Q to another data object Q2 with dCopy(Q,Q2) and then ‘set a break point’ when dEqual(Q,Q2) is no longer true.

The most common error associated with the allocatable memory is neglecting to deallocate an allocated data object. Since data objects must be deallocated in reverse order (last allocated – first deallocated), an error will only occur the next time an already existing data object is deallocated. The problem with this is that the error may not occur near the actual cause of the problem (it often occurs in the calling routine). The pMemory utility ‘pMemInfo’ is one way that errors of this kind can be quickly tracked down, although the easiest thing to do is to be conscientious about deallocating all allocated data objects.

3.4. Example: Removing Duplicates from a Set

In order to give a concrete idea of the character of programs using Types, consider a program to remove duplicates from an ordered set (see figure 2). Subroutine dRemDupl has a single input data object (the dOrSet ‘InSet’) and a single data object as output (the dOrSet ‘OutSet’). Conventionally, output data objects like OutSet are allowed to be any data object when the routine is called. When the routine is finished, however, OutSet must be a dOrSet and, in this case, OutSet must be a dOrSet containing the elements of InSet with duplicates removed.

Note the conventional declaration of InSet and OutSet as integer arrays with unknown lengths. This is the usual way to declare data objects which are passed as arguments. ‘REAL InSet(*)’ would work just as well since the declaration implies nothing about the contents of InSet. The ‘INCLUDE’ statement contains the common block /TypeMem/ with the large array ‘o’ initialized as a pMemory.

dRemDupl demonstrates many of the attractive features of programs written with Types. Removing duplicates from a set is an easy operation since Types allows a direct mapping of abstract set operations into Fortran code. Since InSet may be any ordered set, InSet may contain integers, character strings, or other ordered sets in any mixture. In
SUBROUTINE dRemDupl (InSet, OutSet)

C global constant: not modified by the program
C INTEGER InSet(*)
C global virgin: initialized by the program
C INTEGER OutSet(*)
C INCLUDE 'Types$:TypeMem.inc'

INTEGER dOrNum,j,TempSet
LOGICAL dOrMem

CALL dSetType(OutSet,'dOrSet')

CALL pMemSSiz(o, InSet,TempSet)

CALL pMemDalc(o, TempSet)
the normal Fortran style, each of the above situations would require a different, and more
difficult solution. Since the algorithm is so simple at the level of set operations and since
the Types routines are well established, there is almost nothing that can go wrong with
this program. Debugging dRemDupl is almost unnecessary. dRemDupl is also a robust
program. For example, if InSet is not a dOrSet, an error message will be produced by
the first dOrSet operation (dOrNum). If OutSet is not initially a data object, an error
message will be produced by the call to dSetType. If OutSet is a data object, but is not
large enough to hold InSet with duplicates removed, an error message will occur in one
of the dOrAdd operations. These built in safeguards are essential for using dRemDupl
with confidence and, in general, for building large applications out of similarly robust
programs.

Since almost all of the computations in a call to dRemDupl occur in the Types
routines rather than in the body of dRemDupl, the Types routines can be written to take
advantage of special hardware without complicating the use of the routines at a higher
level of abstraction. For example, Type operations can be optimized once and for all for
hardware pipelines or other architectures without making any more requirements on the
author of dRemDupl or similar programs.

3.5. Using Types Utilities for User Interfaces

Rather than writing a new user interface for each new program, the Types package
provides automatic user interfaces for arbitrary data objects. Types utilities include
dCreate (interactive creation of a data object), dEdit (editing an existing data object),
dInput (user initialization of a data object), dDisplay (non-graphics display of a data
object) and others. For example, dInput(Q) uses, the structure, types and names of data
object Q and its contents to hold an interactive conversation with the user and initialize
Q. By using these utilities, it is no longer necessary to write new user interfaces or display
programs.

As a concrete example, consider the situation of user input for a minimization
package. We assume that this minimization package is similar to other programs in that
there are a large number of parameters governing the minimization which only rarely need
to be changed. Usually, the package will be called specifying a function to minimize, with
one or two other parameters. The usual solution to this sort of problem is to write the
main minimization subroutine accepting all parameters as subroutine arguments. Then
two envelope routines could be written; one routine with default parameters and one
routine with no defaults, but with a user interface to initialize all parameters. This
solution is less than convenient because of the tedious job of writing the user interface
and because the easiest way to write the user interface results in the parameters having to be all re-entered for each change. Extra work is required to be able to save new parameter sets for future runs of the program.

There are several easy alternatives to this procedure using the Types utilities. One of the simplest is to write a main minimization routine assuming a data object (say ‘Params’) as an input argument containing all of the parameters. A default Params data object can be created with the interactive program dCreate. Then, Params can be translated into the corresponding Fortran subroutine (say iParams) with the utility dSource. The envelope would then contain

```
CALL iParams (Params)
```
to get the default parameters, followed by

```
CALL dEdit (Params)
```
to optionally change the parameters. A more sophisticated variation of this scheme is to use the utility ‘dReadTry’ instead of creating the default parameters directly. A call to

```
dReadTry (Params,'Params.d', iParams)
```
causes Params to be initialized by the subroutine iParams if the file ‘Params.d’ cannot be opened. If ‘Params.d’ is accessible, then Params is initialized with the contents of ‘Params.d.’ Each time dReadTry is called, an option is offered to save Params in file Params.d which will be read on future executions of the program. To change parameters, simply use the dEdit command to change the Params.d file. This style of using the Types utilities allows consistent, machine independent user interfaces to be created with very little effort.

### 3.6. Moving Data Objects from Computer to Computer

Data objects in Types are stored in unformatted files. In order to transfer such a data object to another machine, dToExch should be used to translate the data object into a formatted “exchange” file. The resulting file can then be copied to another machine as a character file and translated back into the normal unformatted form with dFrExch. The exchange format file is normally less than twice the size of the original.

One simple use of types is to transfer binary data from machine to machine. For example, suppose that a program contains a real array X(100,100) which needs to be copied to a different computer. The following code fragment copies X into a real pOrSet XSave and writes XSave to a disk file ‘XSave.de’ in exchange format.
CALL pMemAloT (0, 10000+21, XSave,'pOrSet-REAL')
CALL pOrVToOr (10000,X, o(XSave))
CALL dToExch (o(XSave),'XSave.de')
CALL pMemDalc (0, XSave)

After transferring the file Xsave.de to another machine, Xsave can be restored with the command dFrExch.

3.7. Machine Independent Help Systems

A Help system can be viewed as a tree or an ordered set of pairs of the form (keyword, explanation). Here 'keyword' could be character string (pChaStr) and 'explanation' could be a block of text (pText). A help system would then consist of a data object of the above form and a program to match keywords and display the corresponding explanation. The advantage of using Types here is that it is easy to implement and is automatically machine independent. The program invoked by the 'typehelp' command is a simple example of such a help system.

3.8. The Virtual Memory Types

What is one to do if an application exceeds real memory on a computer without virtual memory (like a Cray), or if the maximum virtual memory is exceeded? In such situations, one is likely to receive advice like 'write your own paging system.' This advice is correct, but is not something that should be redone for each new situation. To solve this sort of problem, Types includes a virtual memory type "pVirMem." When a pVirMem is created, it is automatically associated with a temporary file which holds pages just as in other virtual memory systems. The memory associated with a pVirMem can be accessed as one large one dimensional array through the type operations pVirPut and pVirGet. Note that any number of independent virtual memories can be used.

Data objects in Types are referred to by a starting address with the body of the data object starting at that address and continuing in contiguous memory. Inside of the type operation routines, contiguous memory is mapped into a dummy one dimensional vector in the usual way. It is sometimes convenient to implement data object on other linear 'substrates'. An example of this is dTree (tree of data objects) which is not directly implemented as operations on a dummy vector, but is instead implemented on a pSequenc (sequence of scalars) substrate. Similarly, if a type is implemented in a pVirMem virtual memory rather than in an ordinary array, it can take up much less physical memory space than would otherwise be necessary. Types includes virtual memory implementations of some of the types which are most likely to need large amounts of memory. In particular
virtual memory implementations exist for the ordered set of data objects (dOrSetV) and for the stack of data objects (dStackV). In both of these cases a virtual memory dOrSet supports the same set of operations with exactly the same syntax as an ordinary dOrSet. Thus, for example, if an ordered set is using too much memory, changing the initial call to 'dOrSet' to a call to 'dOrSetV' is the only change needed to cause most of the dOrSet to be stored in virtual pages.

3.9. Using Types for Storing Benchmarks

Working programs often need to be accompanied by a set of benchmarks so that a program can be tested in standard situations. This is helpful in the development stage when benchmarks can be re-run to verify that some change has not caused a problem. As an example, consider benchmarks for the detector simulation program 'MC4' [3]. A benchmark for MC4 consists of:

- A Text file describing the benchmark
- A set of "Lattice files" containing input primitive volumes
- A "Materials file" containing information about the materials used in the detector
- A command file (in VMS) to create a "Geometry file" from the Lattice files
- A set of initial particles to start the simulation
- A command file to run the simulation
- The first ten events of the simulation run
- A set of reference histograms.
- A command file to run the whole benchmark

On Unix systems, the "command files" above must be replaced with the analogous "shell script." This solution, although reasonable on the surface, should be criticized on several points: a) the command files or shell scripts are machine dependent, b) There is no mechanism for guaranteeing that a benchmark is complete — there may be files missing

c) there is no mechanism for guaranteeing that there is no ambiguity — there may be two different version of some of the files, d) there is no mechanism for guaranteeing that different benchmarks have the same set of components and that the command files do the same thing. All of these criticisms can be met by using data objects instead of data files and command files. Let a 'benchmark' be an ordered set

Benchmark = (Descript, Lattices, Materials, GeoCreate, MC4input, FirstTen, RefHistos)

where the components of Benchmark are data objects. For example, 'Descript' is a text description (pText), 'Lattices' is an ordered set of lattice data objects, 'MC4input' is a
data object used as input to the simulation, 'ReffHistos' are a set of reference histograms (a dOrSet of pHistgrms) etc. Benchmark can be defined so that to initialize a benchmark, you must supply all of the components. Since a Benchmark is self contained, it is natural that it be stored as a single data object in a single file. Since Benchmark is a data object, all of the types utilities are available including, for example, dDisplay can be used to examine the contents of Benchmark. Using the type operations, it is easy to write a single program which takes a benchmark as an input, extracts the components, calls the program for creating the 'Geometry file' (now a data object), calls the simulation, creates a new set of histograms, compares the new histograms with the old histograms and reports whether the program has passed the benchmark. If Benchmark is larger than convenient for virtual memory, one can simply use the dOrSetV virtual memory implementation instead of dOrSet so that Benchmark exists in a virtual pages rather than in memory (see section 3.7).

One can go even further with this idea and include the program that runs the benchmark and include machine independent compile and load data objects as pText components of Benchmark (it is easy to make a machine independent compile and load system). Then a single command file could execute any benchmark for any program by running a program which extracts the benchmark source, compiles, loads and runs the benchmark. Going to the next level of generality is only prevented because of the difficulty in calling the Fortran compiler from a Fortran program. Of course, this has to stop somewhere, but the point is that many of the functions performed by command files or shell scripts are more usefully done with Type operations.
4. TUTORIAL

The following examples can be followed to give an introduction to the use of Types operations and utilities. User input is indicated by bold type and comments are given in double quotes.

In order to get started, you have to execute a command file or shell script which defines a standard set of symbols needed by the Types system. This is done in various ways on various systems (see the installation instructions). When the symbols are successfully defined, you will get a message:

Enter [typehelp] for information about TYPES.

The help system Types is invoked by

\$ typehelp

which will result in a message analogous to the following.

===== TYPES version 0.2 ALPHA RELEASE, Dec. 26, 1989.
===== Information about TYPES and TYPE utilities.
- Enter the name of a Type or a Utility:

From this point, you might want to experiment with the help system. The command “intro” gives an introduction, “commands” gives a list of the commands available at the level of the operating system and “types” gives a list of the available types and utilities. Information about specific utilities and specific types can be found by entering the name of the type etc. For example, entering “pScalar” results in a list of the available operations on real or integer pScalars. Since there are a large number of subroutine calls available in the Types, the help system needs to be used constantly.

While you are still in the help system, try entering ‘dcreate’ to see the following:

```
/CREATE/pText/COMPOSITE
```

```
SUBROUTINE dCreate(Q)
```

```
dCreate(Q) creates and arbitrary data object Q by having a conversation with Fortran unit 6.
```

Input:

Q       Any primitive or composite data object. dCreate keeps the name and type of Q as given on input.
dCreate is an example of a Fortran callable program which has an analogue at the level of the operating system. This display is the standard output of dDisplay which has a header of the form ‘/data object name/type/type of components.’ The display consists of a pText which is simply the first line and comments from the dCreate subroutine. If you are working at a terminal, you may want to print out the entire help system since so that you can refer to it as you proceed. Since the help system is a data object, you can make a text file version of it with the dPrint utility. To invoke the dcreate utility, enter ‘dcreate’ and try the following session.

```bash
$ dcreate
= Creation of a data object.
Enter the NAME of the new data object:
Test
Enter a DATA TYPE for new data object:
dOrSet “Input is case insensitive”
Enter a sequence of components for [Test/dOrSet]:
Add another data object to [Test]?
yes “You are first asked for the elements of the dOrSet which can have primitive types or general types including dOrSet”
Enter the NAME of the new data object:
Welcome String: a first notification
String must be at most 32 characters; try again.
Enter the NAME of the new data object:
The Welcome String
Enter a DATA TYPE for new data object:
pChaStr “A character string”
Enter character string to initialize [First element of Test]:
hello
```

```
/The Welcome String/pChaStr/CHARACTER
"hello"
```
Add another data object to [Test]?  yes
Enter the NAME of the new data object: x,y,z position in cm.
Enter a DATA TYPE for new data object: porset "an ordered set of REALS"
No such type. Try again.
Enter a DATA TYPE for new data object: porset-REAL "When necessary, the type of a primitive data object is specified by -REAL or -INTEGER"
Enter a sequence of REALS to initialize [second element]; Enter a sequence ending with "/": 1.,2.,3./

Add another data object to [Test]?  yes
Enter the NAME of the new data object: A scalar integer
Enter a DATA TYPE for new data object: pscalar-integer
Enter INTEGER scalar value to initialize [An integer]: 12345
Add another data object to [Test]?  yes
Enter the NAME of the new data object: A logical flag
Enter a DATA TYPE for new data object: pLogical
Enter LOGICAL value to initialize [A logical flag]: .true.
Enter file name to hold data object [Test]:
test.d
Data object [Test/dOrSet] written to file [test.d].

The data object in 'test.d' could have also been created in Fortran program with a call to
dCreate or it could have been done "by hand" by creating each of the components and
adding them to a dOrSet and using dWrite to write the data object to a disk file.

Now that the data object 'Test' is in a file, all of the types commands can be used.
For example,

$ dinfo
  = Enter filename:
test.d

================================ Data Object Information =================================
- Name: Test
- Type: dOrSet
- Components: COMPOSITE
- Size in INTEGER words: 116
- Total space in INTEGER words: 116

================================ End of Data Object Info ===============================

This gives basic information about the data object,

$ dskeleton
  = Enter filename:
  test.d

------------------------------------------
/Test/dOrSet/COMPOSITE
/The Welcome String/pChaStr/CHARACTER
/x,y,z position in cm./pOrSet/REAL
/An integer/pScalar/INTEGER
/A logical flag/pLogical/INTEGER

------------------------------------------

gives the internal structure of the data object and

$ ddisplay
  = Enter filename:
  test.d

------------------------------------------
/Test/dOrSet/COMPOSITE
/The Welcome String/pChaStr/CHARACTER
  "hello"

/x,y,z position in cm./pOrSet/REAL

<table>
<thead>
<tr>
<th>Element #</th>
<th>Value</th>
<th>3 Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.10000000E+01</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.20000000E+01</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.30000000E+01</td>
<td></td>
</tr>
</tbody>
</table>

/An integer/pScalar/INTEGER

  Value = 12345

22
A logical flag/pLogical/INTEGER ------------------------------------------

Value = .true.

-----------------------------------------------------------------------

gives a complete display of the data object and its contents.

You can also edit 'Test' with dEdit. The following session changes the third element
of 'x,y,z position in cm.' to 999.0.

$ dedit
  = Enter filename:
  test.d
  = Edit session for data object [Test].
/Test/dOrSet/composite
The Welcome String/pChaStr/character
  "hello"
Change these value(s)?
  no
/x,y,z position in cm./pOrSet/Real

<table>
<thead>
<tr>
<th>Element #</th>
<th>Value</th>
<th>3 Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.10000000E+01</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.20000000E+01</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.30000000E+01</td>
<td></td>
</tr>
</tbody>
</table>

Change these value(s)?
  yes
Enter 3 REALs to initialize [x,y,z position in cm.]:
  1.,2.,999.

An integer/pScalar/integer
  Value = 12345
Change these value(s)?
  no

A logical flag/pLogical/INTEGER
  Value = .true.
Change these value(s)?
  no

= Enter output file (〈cr〉 for same as input):
Data object [Test/dOrSet] written to file [test.d].

Although this example does not show it, components of a data object can be skipped in dEdit so that you don’t have to answer ‘no’ for all data objects.

A data object like ‘Test’ can be read from its file in a Fortran program into a data object Q by “CALL dRead(Q, 'test.d')”. The alternative call “dReadA(o, Q, 'test.d')” allocates a data object o(Q) and does not require knowing the size of ‘Test’. It is sometimes more convenient to “hard wire” a data object into the equivalent subroutine so that applications do not accumulate too many input files. To create the equivalent source code for ‘Test’, use the dSource command:

```fortran
$ dsourse
== Source file to create a particular data object.
Enter the name of a file containing a data object:
test.d
Enter the name of the new subroutine:
iTest
Source for data object [Test data object] written to file: [iTest.for].
```

The file ‘iTest.for’ now contains a Fortran subroutine ‘SUBROUTINE iTest(Q)’. Calling iTest with a data object Q causes Q to be initialized to the data object ‘Test’.

The following program is an example of how data objects and their components are manipulated inside Fortran programs. Program Tester (figure 3) reads the data object ‘Test’, displays its contents and extracts its contents into normal Fortran variables. Follow the comments in Tester and use typehelp to verify the function of each of the type operations. The first three characters of the type operations identify what type they operate on. For example, dOrLtA4 operates on dOrSets. The meaning of a call to dOrLtA4 can be found in typehelp under “dOrSet.” To run Tester, you must compile it with the Types library. For the (machine dependent) way of doing this, see the installation notes.
PROGRAM Tester

program to read 'Test' and extract the contents

INCLUDE 'Types$:TypeMem.inc'
LOGICAL pLogValu

data objects

INTEGER Q,string,vector,int,flag

ordinary Fortran variables

REAL xVector(3)
CHARACTER*80 xstring
INTEGER xint,nxstring,len
LOGICAL xflag

Read the data object 'Test' from file 'test.d'.
'Test' is read directly into the allocated data object Q.

CALL dReadA(o, Q,'test.d')

write a display of the contents of o(Q) to Fortran unit 6.

CALL dDisplay(o(Q))

Let o(Q) = < string, vector, int, flag >. Once again, string, vector, int and flag are created as allocated data objects.

CALL dOrLitA4 (o, o(Q), string,vector,int,flag)

fetch the character string in o(string)

CALL pChaGet(o(string), nxstring,xstring)

copy o(vector) into an ordinary Fortran array.
pOrOrToV is meant to suggest Ordered-Set-to-Vector.

CALL pOrOrToV(o(vector), 3,xvector,len)

'3' is the amount of space available in xvector and 'len'
is the number of elements actually written to. len is
supposed to be 3, so...

IF (len.ne.3) CALL Abort(' Error in Tester. ')

fetch the scalar

CALL pScaGet(o(int), xint)

say "hello" if o(flag) is true and then copy it into an
ordinary Fortran logical.

If (pLogValu(o(flag))) WRITE(6,1)
1 FORMAT(' hello')
xflag = pLogValu(o(flag))

Don't forget to deallocate!

CALL dOrLtD4(o, string,vector,int,flag)
CALL pMemDalc(o, Q)

END
5. AVAILABLE TYPES AND UTILITIES

A brief description of each of the types and utility routines available in the current release of the Types follows. Calling instructions for the following operations can be found in the help system (typehelp). For each type, we give information on how much space is used by the current implementations. In this context, a ‘word’ means a Fortran integer sized word. Note that usually one only needs an upper bound on the amount of space needed by a data object since, in practice, most data objects are allocated and thus any wasted space is regained when they are deallocated. Note also that the following numbers are implementation dependent and may change with future versions of types (they are not likely to increase).

5.1. Primitive Types

A primitive type is a data object with fixed components often Fortran real or integers.

5.1.1. pUndefnd

pUndefnd is the simplest data type. No operations are available for pUndefnd objects. pUndefnd is useful for situations where one needs a data object, but it’s type is unimportant. An example is supplying an argument to a Type operation which is going to be initialized by a subroutine. Another use of the pUndefnd type is in situations where part of a data object is initialized through a user interface and another part is initialized by a program. If a data object is prepared with pUndefnd components, no inputs will be requested in a user interface utility such as dInput. The pUndefnd components can then be replaced in a Fortran program. pUndefnd needs only enough space for a header, i.e. at most 20 words.

5.1.2. pScalar

pScalar is a type for storing a single Fortran real or integer scalar value. pScalars need at most 21 words.

5.1.3. pBScalar

pBScalar is a bounded real or integer scalar which must be created with upper and lower bounds. A similar set of operations are available as with pScalar. The chief advantage of pBScalar is that the bounds are automatically reflected in the user interface utilities. pBScalar needs at most 23 words.
5.1.4. pOrSet

pOrSet is an ordered set of reals or integers with an extensive set of natural operations including copying to and from normal Fortran arrays. A pOrSet needs 21 words plus the maximum number of words to be stored in the pOrSet.

5.1.5. pStack

pStack is a push/pop stack of real or integer scalar values. pStack needs 21 plus the maximum number of words stored in the pStack.

5.1.6. pChaStr

pChaStr is a character string with operations including copying to and from ordinary Fortran character strings. pChaStr needs 21 words plus one word per character in the largest character string to be stored.

5.1.7. pMemory

pMemory is an extremely commonly used type which allows allocation and deallocation of other data objects. Types comes with a common block 'TypeMem' containing array 'o(pMEMLEN)' with parameter pMEMLEN set to a large value. This is the global pMemory which the Type routines expect for type operations. The array 'o' is automatically initialized at the time of the first allocation request. Data objects must be deallocated from a pMemory in reverse order to their allocation (last allocated, first deallocated). A pMemory operation called pMemInfo is available to give a message each time memory is allocated and deallocated. pMemory needs at least 50 words and, to be useful, usually has millions. The space used by the standard pMemory 'o' is determined at the time of installation. You can adjust this size by changing the pMEMLEN parameter in all programs and recompiling the entire package.

5.1.8. pText

pText is a multi-lined block of text. pText is currently implemented as an ordered set of pChaStr. If \( m \) is the maximum number of characters per line and \( n \) is the maximum number of lines in the block text, then \( 21 + (21 + m) \times n \) words will suffice.

5.1.9. pVeArray

pVeArray is a vector of three dimensional arrays of integer or real numbers. If MaxVec is the maximum number of 3d arrays of scalars to store in pVeArray, then \( (21 + 2 + 5 \times \text{MaxVec} + s) \) suffices where \( s \) is the number of elements in the stored 3d arrays.

5.1.10. pLogical

pLogical is a single logical value. 21 words suffices for a pLogical.
5.1.11. pSequenc

pSequenc is a sequence of integer or real scalars supporting operations like insertion and deletion. For a pSequenc to store a sequence of \( n \) reals or integers, \( 23 + 2 \times n \) words suffices.

5.1.12. pHistGrm

pHistGrm is a histogram. If there are \( n \) bins in the histogram, \( 7 \times 23 + 2 \times n \) words suffices.

5.1.13. pQueue

pQueue is a last-in-first-out queue of integer or real scalars. If \( n \) scalars need to be stored in a pQueue, \( 25 + n \) words suffices.

5.1.14. pVirMem

pVirMem is a virtual memory which is accessed as if it were a single large one dimensional array. The optimum size of a pVirMem and the optimum page size are difficult to determine for all cases. However, giving more memory to a pVirMem allows it to store more virtual pages in real memory and improves efficiency.

5.2. General Types

A general type can either have primitive types or other general types as components.

5.2.1. dOrSet

dOrSet is the ordered set of data objects and is used to build complex data objects from other data objects as components. dOrSet is one of the most often used types and supports many operations. An alternative implementation of dOrSet called dOrSet V performs the analogous dOrSet operations partially in private virtual pages. Since dOrSets are usually the largest data objects in an application, dOrSet V is a convenient way to save virtual or real memory at the expense of a temporary disk file. For a dOrSet to hold \( n \) data objects with a total used space of \( N \) words, \( 21 + n + N \) words suffices.

5.2.2. dStack

dStack is a push/pop stack of data objects. If a dStack is to store \( n \) data objects with a total used space of \( N \) words, then \( 21 + n + N \) words suffices.

5.2.3. dTree

dTree is one of the several types of Tree data structures. A dTree is an ordered set of pairs where each pair is a data object and a dTree. If a dTree contains \( n \) nodes and contains data objects with a total of \( N \) words in used space, then \( 2 \times 21 + 2 \times (21 \times n + N) \) is sufficient space to store it.
5.2.4. dStArray

dStArray is a two dimensional array where the array elements are data objects. If a dStArray is a max1 × max2 element array of data objects, with a total of \( N \) used words in the data objects stored as elements, then \( 21 + 3 + \max1 \times \max2 + N \) words is sufficient.

5.3. Display and Editing Utilities

5.3.1. dCreate

dCreate is a program which creates an arbitrary data object in an interactive session. dCreate is available both as a subroutine and as a stand alone program which can be invoked by the ‘dCreate’ symbol defined by the Types package.

5.3.2. dInfo

dInfo gives basic information about a data object stored in a file. dInfo is available as a command at the level of the operating system.

5.3.3. dSkeletn

dSkeletn gives a skeleton display of the structure of a data object without showing the contents. dSkeletn is available as a command at the level of the operating system.

5.3.4. dDisplay

dDisplay is the often used general display of any data object showing structure and contents. dDisplay is available as a stand alone program.

5.3.5. dEdit

dEdit is a program used for interactive editing of data objects either in a program (as subroutine dEdit) or in a file (as the dEdit command).

5.4. Input/Output Utilities

5.4.1. dWrite

dWrite writes a data object to a given file.

5.4.2. dWriteQ

dWriteQ also writes a data object to a file. The difference is that dWriteQ queries the user for the name of the file rather than requiring the file name as an argument. This is convenient for calling in a symbolic debugger.

5.4.3. dRead

dRead read a data object from a file.

5.4.4. dReadA

dReadA reads a data object from a file into an allocated data object in case that the size of the data object in the file is not known.
5.4.5. **dReadQ**

*dReadQ* read a data object from a file with the file name taken from user input.

5.4.6. **dAlocIO**

*dAlocIO(Unit)* allocates a Fortran unit number for i/o operations. The corresponding deallocation call is *dDalocIO(Unit)*. Types allocates Fortran unit numbers in the range 50, ..., 99. If Fortran unit numbers *Unit(1), ..., Unit(nUnit)* are used by other programs, these numbers should be reserved by calling *dAlocUsd(nUnit,Unit)* at the start of the main program.

5.4.7. **dOpenNew**

*dOpenNew* opens a new unformatted file for writing data objects. The corresponding *dClose* should be used to close the file.

5.4.8. **dOpenOld**

*dOpenOld* opens an existing unformatted file containing a data object or a sequence of data objects in successive records. *dClose* should be used to close the file.

5.4.9. **dClose**

*dClose* closes a specified file open by *dOpenNew* or *dOpenOld*. *dClose* also deallocated the Fortran unit number attached to the file.

5.4.10. **dUWrite**

*dUWrite* writes a data object to a file specified by a unit number. This is useful if you are writing multiple data objects to a sequential file.

5.4.11. **dURead**

*dURead* reads data object from a file specified by a unit number. *dURead* is useful for reading files containing more than one data object in successive records.

5.4.12. **dUReadA**

*dUReadA* is similar to *dURead* except that the read data objects are read directly into an allocated data object.

5.4.13. **dReadTry**

*dReadTry* reads a data object from a specified file, if the file can be opened. Otherwise, *dReadTry* initializes the data object by calling a specified subroutine. *dReadTry* is useful for user interfaces. See section 4.4.
5.4.14. dSource

dSource(Q,'SubName') with data object Q as an argument produces the source code for a Fortran subroutine called 'SubName' which, when called, initializes its output argument to Q. dSource is available both as a subroutine and as a stand alone program available as a command in the operating system. dSource is useful for “hard wiring” a data object into a Fortran subroutine so that applications do not have to accumulate many input files. Note that the source code produced by dSource is not, in general, transportable from machine to machine. However, you can transport the original data object with dToExch and dFrExch and then re-run dSource on the new machine.

5.4.15. dToExch

dToExch takes a data object as an argument and writes a file containing that data object in “exchange” format. The exchange format file can be transferred from machine to machine as a formatted character file and then restored on the target machine with dFrExch. The exchange format typically uses less than twice the space of the original data object. dToExch is available both as a subroutine and as a stand alone program which can be invoked by the symbol ‘dtoexch’.

5.4.16. dFrExch

dFrExch takes a data object stored in exchange format and creates the corresponding data object in standard form.

5.5. Utilities for Extracting Information about Data Objects

5.5.1. dSpace

dSpace returns the number of words contained in a data object.

5.5.2. dUSpace

dUSpace returns the number of words currently used by a data object.

5.5.3. dSpLeft

dSpLeft returns the number of free words left in a data object as room for expansion.

5.5.4. dHeadSz

dHeadSz returns the size (in words) of the “header” of a data object.

5.5.5. dType

dType returns an integer which is unique for each of the data types.

5.5.6. dPrim

dPrim returns .true. if and only if its data object argument is a primitive data object.
5.5.7. dPrimTyp

dPrimTyp returns an integer indicating the type of component of a data object.

5.5.8. dTyNmGet

dTyNmGet fetches the name of the type of a data object.

5.5.9. dDsNmGet

dDsNmGet fetches the name of a given data object.

5.5.10. dTypeNam

dTypeNam returns .true. if and only if its argument is the name of an existing data type.

5.6. Miscellaneous Utilities

5.6.1. dName

dName assigns a name to a data object. The name may be any character string of up to 32 characters.

5.6.2. dCopy

dCopy copies one data object into another.

5.6.3. dEqual

dEqual returns .true. if and only if its two data object arguments are 'equal'. Two data objects are equal if they have the same type and equal components (this is not word by word equality because two data objects may have the same content but one may have more free space).

5.6.4. dCongru

dCongru returns .true. if and only if its two data object arguments are 'congruent.' Two data objects are congruent if they have the same 'structure.' Formally, two data objects are congruent if they have the same types and each corresponding component has the same type.

5.6.5. dUnion

dUnion is available as a stand alone program and is similar to the dOrUnion type operation on dOrSets. dUnion forms the dOrSet union of several data objects stored in files.

5.6.6. dCheck

dCheck aborts unless its argument is a legal data object.
5.6.7. dCheckQ

dCheckQ returns .true. if and only if its argument is a legal data object.

5.6.8. dChecAll

dChecAll is available as a subroutine or as a stand alone program. dChecAll makes extensive checks of the integrity of a data object supplied as an argument or a file.

5.6.9. dSetType

dSetType sets or changes the type of a data object. When changing the type of a data object, all information is lost except for its size.
6. REFERENCES

[1] We use the term "data object" only to emphasize that the data structures created in the Types package are self-describing. We claim no connection to "Object Oriented Programming."

[2] Types is written in FORTRAN 77 allowing the extension of 8 character variable names which can be in mixed upper and lower case. In referring to symbolic names in Fortran programs, we use mixed upper case and lower case to improve readability. Input to the Types package is, however, case insensitive. Some examples in this manual also use an ‘INCLUDE’ statement for including text from a disk file.

All primitive or compound data structures (abbreviated 'dO') are stored in standard FORTRAN one dimensional arrays. Each data structure of the form S(1),S(2),..., has two main sections: a header containing general information and having the same format for each data structure and the data section containing data structure dependent information. The following defines the detailed format of the header. The terminology 'word' means a word with the same size as the FORTRAN data type 'INTEGER'.

<table>
<thead>
<tr>
<th>Word</th>
<th>Name</th>
<th>Meaning or value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>S(1)</td>
<td>Robinacci</td>
<td>$= 1123581321$</td>
<td>identifies the array as a dO.</td>
</tr>
<tr>
<td>S(2)</td>
<td>DataStart</td>
<td>Data section starting address</td>
<td>S(DataStart) is the first data word.</td>
</tr>
<tr>
<td>S(3)</td>
<td>DataEnd</td>
<td>dO end including data section</td>
<td>S(DataEnd) is the last word including the data section.</td>
</tr>
<tr>
<td>S(4)</td>
<td>SpaceEnd</td>
<td>End of the available space</td>
<td>S(SpaceEnd) is the last word that can be used. For a primitive dO, PrimType = 1 for an INTEGER, =2 for REAL, =3 for COMPOSITE =4 for UNDEFINED, =5 for a single CHARACTER, =6 for VIRTUAL.</td>
</tr>
<tr>
<td>S(5)</td>
<td>PrimType</td>
<td>Primitive type</td>
<td>A unique integer identifying the data type. nTyName&lt;=mTyName</td>
</tr>
<tr>
<td></td>
<td>dsType</td>
<td>INTEGER dO type</td>
<td>nTyName&lt;=mTyName</td>
</tr>
<tr>
<td>S(7)</td>
<td>nTyName</td>
<td>Length of the data type name</td>
<td>nTyName&lt;=mTyName</td>
</tr>
<tr>
<td>S(7+1)</td>
<td>TyName(1)</td>
<td>First character of the type name</td>
<td>Packed character data.</td>
</tr>
<tr>
<td>S(7+2)</td>
<td>TyName(2)</td>
<td>Second character of the type name</td>
<td>Packed character data.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S(7+nTyName)</td>
<td>TyName(nTyName)</td>
<td>nTyName'th character of type name</td>
<td>Packed character data.</td>
</tr>
<tr>
<td>S(7+nTyName+1)</td>
<td>nDsName</td>
<td>Length of the optional dO name</td>
<td>nDsName&lt;=mDsName.</td>
</tr>
<tr>
<td>S(7+nTyName+1 + 1)</td>
<td>DaName(1)</td>
<td>First character of the dO name</td>
<td>Packed character data.</td>
</tr>
<tr>
<td>S(7+nTyName+1 + 2)</td>
<td>DaName(2)</td>
<td>Second character of the dO name</td>
<td>Packed character data.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S(7+nTyName+1+nDaName)</td>
<td>DaName(nDaName)</td>
<td>nDaName'th character of dO name</td>
<td>Packed character data.</td>
</tr>
<tr>
<td>S(DataStart)</td>
<td>data(1)</td>
<td>First Data word</td>
<td>DataStart $= 8+nTyName+1+nDaName+1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S(DataEnd)</td>
<td>Last data word</td>
<td>Last data word</td>
<td>DataLen&gt;=0</td>
</tr>
<tr>
<td>S(DataEnd+1)</td>
<td>expand(1)</td>
<td>Expansion space</td>
<td>Free space to expand if desired.</td>
</tr>
<tr>
<td>S(DataEnd+2)</td>
<td>expand(2)</td>
<td>Expansion space</td>
<td>Free space to expand if desired.</td>
</tr>
<tr>
<td>S(EndSpace)</td>
<td>Last expansion word</td>
<td>Expansion space</td>
<td>Free space: TotalLen &gt;= DataStart + DataLen - 1</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------</td>
<td>-----------------</td>
<td>-------------------------------------------------</td>
</tr>
</tbody>
</table>
DISCLAIMER

This report was prepared as an account of work sponsored in part by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.