# 1. The Calculus Environment

The FC Environment is a special software design for calculus processing, consisting of:

- A set of environment programs
- A special memory organization
- A special set of data structures

It involves the dynamic recasting of real variables into linked data structures, and the overloading of real variable arithmetic operations to facilitate differential arithmetic. Thus real variables can exist in two modes:

- Ordinary mode - Real*8 variables containing normalized floating point numbers, and
- Calculus mode - Real*8 variables containing pointers to calculus operand structures, including partial derivative arrays.

The contexts where real variables are in calculus mode are separate mathematical domains, i.e. coordinate systems. These contexts are separate (lower) levels in program hierarchies, created by the action of the calculus macro statements, i.e., they are below the macro statements in the hierarchy. In these calculus contexts, ordinary processing of real variables is outlawed. All real arithmetic and assignment operations are overloaded. Consequently all calls to lower level procedures are limited. They must be:

(a) Calls to MODELS or FMODELS,

(b) Performed via INCALL or OUTCALL macro statements, or

(c) Not involve real variables.

## 1.1. Environment Programs

The environment programs are a large set of library subroutines that make up the bulk of calculus programs (often more than 90 %). These programs are organized into a central subsystem called the FC Virtual Machine (FCVM) and a set of *solver* subsystems.

The FCVM contains the differential arithmetic primitives, the array operation subroutines, the common solver interface subroutines, all data and environment management subroutines, and common utilities used by more than one solver.

The solver subsystems are individual algorithms and polyalgorithms designed to solve generic mathematical problems. They are the solution elements of calculus processes

described in the next chapter. Solvers are invoked and controlled by the macro statements FIND, INITIATE, and INTEGRATE, and they operate on MODEL and FMODEL procedures.

## 1.2. Memory Organization

The environment memory is divided into two primary storage regions:

- The "Bucket": a large common block (FC3000) which acts as a bulk data memory to the FC virtual machine. All of its internal structures are dynamically managed by the FCVM;
- The Variables Dynamic Table (VDT): a computing buffer employed for temporary storage of calculus mode variable values and for the computation of calculus expressions.

The sizing of these two storage regions is defined by the first two parameters on the PROBLEM statement.

***Bucket Structure*** - The bucket is divided into three dynamic regions as shown in Fig. 1-1. The upper region is used for temporary storage of partial derivative values. The middle region is used for temporary storage of arbitrary blocks of values such as arrays, and the bottom region is a pair of stacks used for expression processing and process management by the FCVM. Each cell in the bucket is a 64-bit double word used to store either real*8 values, a pair of 32-bit integer pointers or an 8-character name.

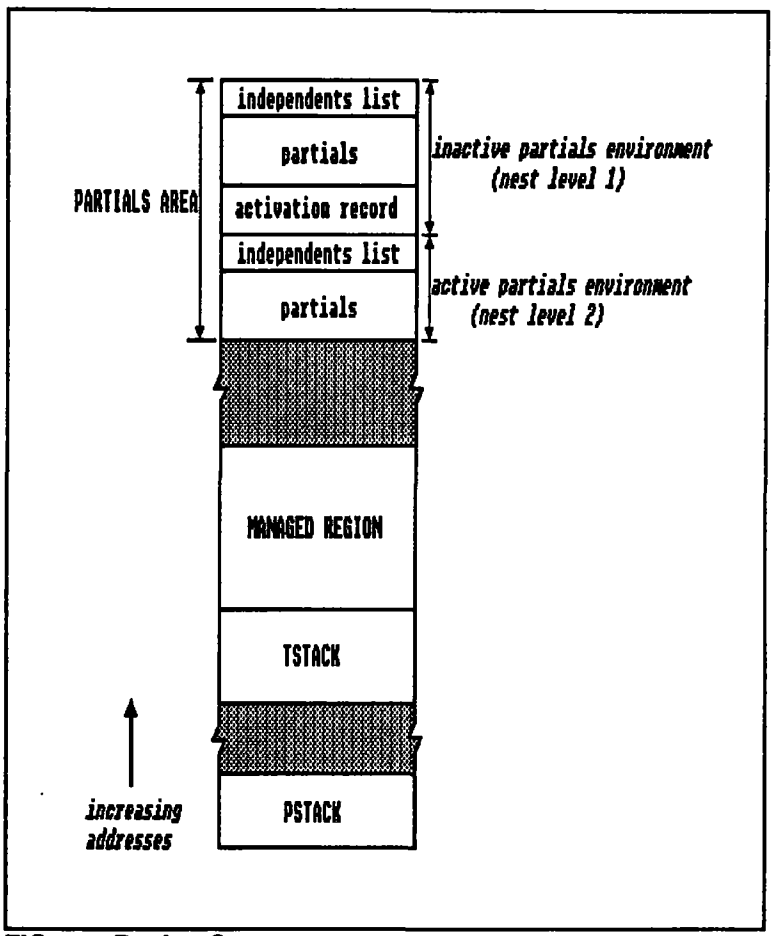


**FIG.1-1 Bucket Structure**

*Variable s Dynamic Table (VDT)* - The VDT is a smaller common block (FC3007) having the structure shown in Fig 1-2.

The upper region of the VDT is allocated to calculus-mode variables in the order that they are stored in a calculus process. The lower region is temporarily used as an execution stack during processing of an expression, thus it is abandoned when the value of the expression is stored.

A variable cell in the VDT is two 64-bit cells, the upper containing a real*8 value, and the lower containing up to two integer pointers.

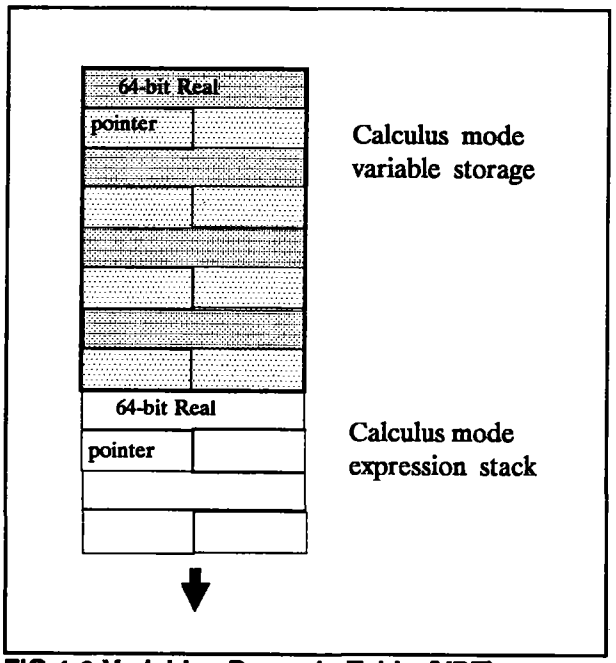


FIG.1-2 Variables Dynamic Table (VDT)

*Calculus-Mode Linkage* - When a real variable is converted from ordinary mode to calculus mode, its value is moved from its original address (variable-cell) to a new address in the VDT; and a cross-linkage is established between the variable cell and the VDT entry. This mode conversion occurs whenever an assigned expression is differentiated and assigned to a variable, or when a constant is assigned to a variable that was previously differentiated. These two cases are illustrated in Figure 1-3 and Figure 1-4 respectively.

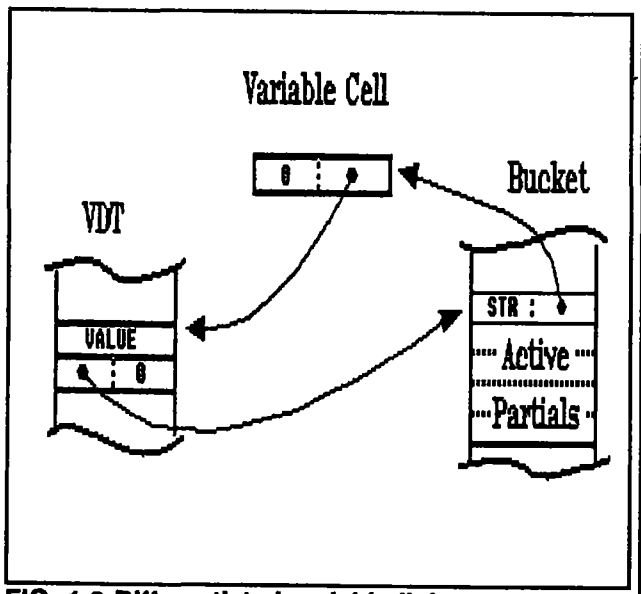


FIG. 1-3 Differentiated variable linkage

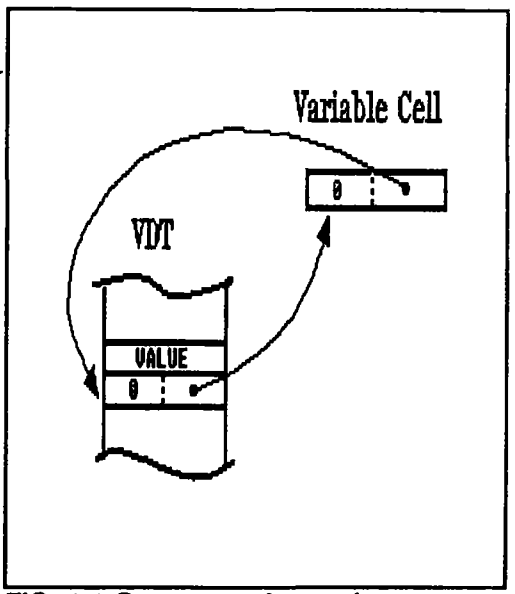


FIG. 1-4 Constant value replacement

In the first case, the partial derivatives resulting from differentiating the expression are stored in the upper region of the bucket and a pointer to this "partials node" is stored in the first integer cell of the VDT entry. Also a pointer to the original variable cell is stored in the partials node.

In the second case, where a constant is assigned to a variable that previously had partials, the partials node is destroyed (returned to free storage) and a pointer to the original variable cell is stored in the second integer cell of the VDT entry.

## 1.3. Macro Statements

The FC macro statements are the extensions to FORTRAN 77 syntax that are translated by the FC translator into FORTRAN 77 statements. The following is the identifying prefix of each macro statement and a short description of its purpose:

### Declarations :

| | | |
|---|---|---|
| **PROBLEM** | : | Program Heading: used for program sizing and initialization |
| **CONTROLLER** | : | Solver controller subroutine heading: used to pass control parameters to solvers |
| **MODEL** | : | Model subroutine heading: Real assignment statements are translated into overloaded arithmetic form in a MODEL. |
| **FMODEL** | : | Function Model Heading: Real assignment statements are translated into overloaded arithmetic form in an FMODEL. |
| **PROCDURE** | : | IMPLICIT REAL*8 subroutines |
| **OPERATOR** | : | IMPLICIT REAL*8 function subprograms |
| **DYNAMIC** | : | Dynamic variable declaration: Defines real arrays to be allocated and deallocated dynamically |
| **GLOBAL** | : | Generic scope declaration: Defines variable classes included in automatic generation of common statements |
| **GRAPHICS** | : | Graphics output mode selection |

### Executable Statements:

| | | |
|---|---|---|
| **FIND** | : | Constraints solving and optimization |
| **INITIATE** | : | Initial value problem definition |
| **INTEGRATE** | : | Initial value problem function generation |
| **INVOKE** | : | Differential arithmetic control |
| **ALLOT** | : | Dynamic array creation and sizing |

| | | |
|---|---|---|
| **PURGE** | : | Dynamic array deletion |
| **INCALL** | : | Model input subroutine call |
| **OUTCALL** | : | Model output subroutine call |
| **<** *array* **>** **=** | : | Array arithmetic assignment statements |
| [*matrix*] **=** | : | Matrix operations statements |
| *@verb(arguments)* | : | command call (general form) |
| **@STREAM** | : | output stream selection |
| **@PRINTER** | : | PRINTER stream unit specification |
| **@SCROLL** | : | SCROLL stream unit specification |
| **@SECTION** | : | SCROLL section partitioning |
| **ROWPRINT** | : | Array report generation by row |
| **COLPRINT** | : | Array report generation by column |
| **VECPRINT** | : | Vector report generation |
| **MATPRINT** | : | Matrix report generation |
| **TABULATE** | : | Tabular report generation |
| **GRADPRINT** | : | Gradient report generation |
| **HESSPRINT** | : | Hessian report generation |
| **PARDPRINT** | : | Partial Derivatives report generation |
| **USE** | : | Report format control |
| **RESET FORMAT** | : | Reset default format controls |

The following sections describe the syntax of these statements and further describe their usage. Braces are used to denote optional syntax.

## 1.3.1. Context Declaration Syntax

The following describes the syntax of the macro statements which initiate and qualify procedure contexts. Further descriptions of these statements are given elsewhere as noted in each case

### PROBLEM Declaration

**Syntax:**

**PROBLEM** *name* {(*bucketsize* {,*vdtsize*{,*stacksize*}} ) }

**Symbols:**

*name* : identifying name (1-8 characters)

*bucketsize* : integer literal = size of bucket storage (default =5000)

*vdtsize* : integer literal = size of VDT (default = 1000)

*stacksize* : integer literal = size of stack region of bucket (default = 1000)

This declaration generates a main program (FC000000) containing sized common blocks for the bucket and VDT storage, and initialization data statements. The program unit identified by name becomes a subroutine which is called by the main program.

### CONTROLLER Declaration

**Syntax:**

**CONTROLLER** *name* (*solver*)

**Symbols:**

*name* : identifying name (1-8 characters)

*solver* : solver name

This declaration generates a subroutine heading and a common block declaration for the common block /C*solver*/ used to communicate control parameters to the solver.

### MODEL Declaration

**Syntax:**

**MODEL** *name* {(*parameters*) }

### Symbols:

*name* : identifying name (1-8 characters)

*parameters* : list of dummy parameters

This declaration generates a subroutine heading and initiates translation of the real variable assignment statements into overloaded differential arithmetic assignments.

## FMODEL Declaration

### Syntax:

**FMODEL** *name* ({ *parameters*})

### Symbols:

*name* : identifying name (1-8 characters)

*parameters* : list of dummy parameters

This declaration generates a real function heading and initiates translation of the real variable assignment statements into overloaded differential arithmetic assignments.

## PROCEDURE Declaration

### Syntax:

**PROCEDURE** *name* { (*parameters*) }

This declaration is an alias for a FORTRAN subroutine containing

IMPLICIT REAL*8 (A-H,O-Z)

as the first statement. The use of this form is encouraged in order to insure type compatibility with other FORTRAN CALCULUS subprograms.

## OPERATOR Declaration

### Syntax:

**OPERATOR** *name* { (*parameters*) }

This declaration is an alias for a FORTRAN REAL*8 FUNCTION containing

IMPLICIT REAL*8 (A-H,O-Z)

as the first statement. The use of this form is encouraged in order to insure type compatibility with other FORTRAN CALCULUS subprograms.

## DYNAMIC Declaration

### Syntax:

DYNAMIC $v_1,...,v_n$

### Symbols:

$v_1,...,v_n$ : variable names

This declaration specifies variables which may be allocated as real dynamic arrays using array assignment statements or the ALLOT statements, and thereafter deleted using the PURGE statement. The FORTRAN equivalent of the declared variables are integer scalars used as pointers to arrays created as blocks of storage in the bucket. Subscripted references to these variables are translated into indexed positions in the bucket vector. Subscripted (i.e. dimensioned) reference to these variables may not appear in common statements and when appearing unsubscripted in common must also appear in DYNAMIC statements in all program units having the same common, or be declared integer (implicitly or explicitly). Dummy arguments of subroutines, MODELs or FMODELs may not be dynamic variables.

## GLOBAL Declaration

### Syntax:

MUST INCLUDE REAL*8 for all I-N variables in every subroutine.

GLOBAL *classes* {; IN *procedures* }

### Symbols:

*classes* : *a list of:* **REALS | CHARACTERS | INTEGERS | PRINCIPALS | ALL**

where REALS denotes all real variables, CHARACTERS denotes all character variables, INTEGERS denotes all integer variables, PRINCIPALS denotes all variables appearing in FIND, INITIATE, or INVOKE statements, and ALL denotes all program variables.

*procedures* : a list of procedure names

?← still true? (COMMENTS ?)

This statement applies to an entire program and if used must precede the PROBLEM declaration. It is used to define the scope of all of the variables in a given class as global within the named procedures. It causes the translator to collect all variables in the specified classes and to include them in a common block /GLOBAL/. Arrays declared in DIMENSION statements are included in dimensioned form. Thus the DIMENSION statements only need appear once in the program.

## GRAPHICS Declaration

### Syntax:

**GRAPHICS SCREEN|FILE|BOTH**

This statement applies to the entire program, and if used must precede the PROBLEM declaration. It is used to select the mode of graphics output. Thus if effects the selection of graphics utilities to be merged with the program by the linker. A different set of utilities are associated with each option. In the absense of this declaration, the default graphics mode is SCREEN.

## 1.3.2. Computing and Data Management Syntax

The following describes the syntax of the executable macro statements which perform computing and data management operations. Further descriptions of these statements are given elsewhere as noted in each case.

### FIND Statement

**Syntax (general form):**

**FIND** *independents*; **IN** *modelcall*; **BY** *solver* {(*controller*)};
{{**WITH**}**LOWER**{**S**} *bottoms;*} {{**WITH**|**AND**} **UPPER**{**S**} *tops*;}
{{**WITH**|**AND**} **BOUND**{**S**}*bounds*;} {{**WITH** |**AND**} **SCALE**{**S**}*scales* ;}
{{**WITH**|**AND**} **REPORTING** *names*;} {{**WITH**|**AND**} **FLAG** *flag*;}
{{**WITH**|**AND**} **HOLDING** *inequalities*;}
{{**WITH**|**AND**} **MATCHING** *equalities*;}
**TO** *criterion*

**Symbols:**

| | | |
|---|---|---|
| *independents* | : *varlist* | of independent variables |
| *modelcall* | : | name {(parameters)} |
| *solver* | : | solver name |
| *controller* | : | controller name |
| *bottoms* | : *varlist* | of lower limits on independents |
| *tops* | : *varlist* | of upper limits on independents |
| *bounds* | : *varlist* | of bounds on independents step changes |
| *scales* | : *varlist* | of scale factors on independents |
| *names* | : *namelist* | of auxiliary model variables |
| *flag* | : | flag variables (real) |
| *inequalities* | : *varlist* | of inequality constraint variables |
| *criterion* | : one of the following: | |

**MAXIMIZE** *objective*
**MINIMIZE** *objective*
**EXTREMIZE** *objective*

**MATCH** *constraints*

| | | |
|---|---|---|
| *objective* | : | scalar variable or array element |
| *constraints* | : *varlist* | of equality constraints |
| *varlist* | : | *element* {,*varlist*} |
| *element* | : | scalar \| array \| *array-element* \| *array-part* |
| *array-element* | : | array-name (subscript expression) |
| *array-part* | : | array-name [size] |
| *namelist* | : | name {,*namelist*} |

This statement is used for constraint matching (simultaneous equation solving) and all forms of mathematical optimization (unconstrained-nonlinear, constrained nonlinear, linear, linear and mixed integer, zero-one, etc.). Allowed phrases in FIND statements depend upon individual solvers, as specified in chapters 4, 6 and 7. The FIND statement performs two roles with respect to the calculus environment: (a) it defines the problem to be solved and its process of solution, and (b) it controls the solution process.

## INITIATE Statement

### Syntax (general form):

**INITIATE** *solver* {(*controller*)}; {{**WITH**} **FLAG** *flag*;}

{{**WITH**|**AND**} **UPPER**{**S**} *tops*;}

{{**WITH**|**AND**} **LOWER**{**S**} *bottoms*;}

**FOR** *model*; **EQUATIONS** *rates/states*;

**OF** *independent*; **STEP** *increment*; **TO** *limit*

### Symbols:

| | | |
|---|---|---|
| *solver* | : | solver name |
| *controller* | : | controller name |
| *flag* | : | flag variable (real) |
| *tops* | : *varlist* | of upper limits on states |
| *bottoms* | : *varlist* | of lower limits on states |
| *model* | : | model name |
| *rates* | : *varlist* | of ordinary derivative variables |
| *states* | : *varlist* | of dependent variables (output) |

| | | |
|---|---|---|
| *independent* | : | independent variable |
| *increment* | : | nominal integration step size |
| *limit* | : | integration limit |
| *varlist* | : | *element* {,*varlist*} |
| *element* | : | scalar \| array \| *array-element* \| *array-part* |
| *array-element* | : | array-name (subscript expression) |
| *array-part* | : | array-name [size] |

This statement is used to define initial value problems of ordinary differential equations. It specifies the integration solver; the model containing the differential equations; the ordinary derivative, dependent and independent variables of the equations; the nominal integration step size (usually varied by the solver); and the upper limit of integration. The control of the solution process is subsequently performed by the INTEGRATE statement.

## INTEGRATE statement

### Syntax:

**INTEGRATE** *model*; **BY** *solver*

### Symbols:

| | | |
|---|---|---|
| *model* | : | model name |
| *solver* | : | solver name. |

This statement is used to generate the integral functions of the differential equations of the model from the current value of the independent variable to the *limit* value.

## TERMINATE statement

### Syntax:

**TERMINATE** *model*

This statement is used to release the model from association with a solver, including any temporary storage allocated for integration.

## INVOKE statement

### Syntax:

**INVOKE GRADIENTS | HESSIANS | ORDER(***intvar***) ON** *independents*;

IN *modelcall*

### Symbols:

| | | |
|---|---|---|
| *intvar* | : | integer variable |
| *independents* | : *varlist* | of independent variables |
| *modelcall* | : | model-name {(parameters)} |
| *varlist* | : | *element* {,*varlist*} |

This statement is used to execute a model using differential arithmetic, producing partial derivatives of all dependent variables computed in the model with respect to the independents.

## ALLOT statement

### Syntax:

ALLOT *arraylist*

### Symbols:

| | | |
|---|---|---|
| *arraylist* | : | *array* {,*arraylist*} |
| *array* | : | *array-name* ($d_1${,$d_2$,...$d_7$}) |
| *array-name* | : | variable previously in a DYNAMIC declaration |
| $d_1$...$d_7$ | : | integer dimension expressions |

This statement allocates dynamic arrays in the managed region of the bucket array. If the array is already allocated, it changes the allocation, preserving unaffected values.

## PURGE statement

### Syntax:

PURGE $v_1$,...$v_n$

### Symbols:

$v_1$,...,$v_n$ dynamic variables

This statement deletes the dynamic arrays from the managed region of the bucket, returning the space to free storage, and sets the pointer values to zero.

## Array Assignments

### Syntax (general form):

*<darr> = rhs*

### Symbols:

*darr* : dynamic array

*rhs* : dynamic array expression : one of the following:

| | |
|---|---|
| {-}*array* | : static array assignment |
| {-}*<darr>* | : dynamic array assignment |
| {-} *element* | : scalar to array assignment |
| {-}(*expression*) | : scalar to array assignment |
| \| *darr* \| | : absolute value array assignment |
| \| *element* \| | : absolute value scalar to array assignment |
| \| (*expression*) \| | : absolute value scalar to array assignment |
| *<darr>* + \|- \|*\| / \| ** *<darr>* | : array - array arithmetic |
| *<darr>* + \|- \|*\| / \| ** *element* | : array - scalar arithmetic |
| *element* + \|- \|*\| / \| ** *<darr>* | : scalar - array arithmetic |
| <darr> + \|- \|*\| / \| ** (*expression*) | : array - scalar arithmetic |
| (*expression*) + \|-\|*\|/\|** *<darr>* | : scalar - array arithmetic |
| **EIGEN** [*darr*] {**VECTORS** [*darr*]} | : eigenvalues of matrix |
| **SORT** [*darr*] | : array sort |
| **REVERSE** *<darr>* | : array reverse |
| **RANK** *<darr>* | : array rank |
| **GRAD**(*element*) | |
| **GRAD**((*expression*)) | |
| **HESS**(*variable*) | |
| **HESS**((*expression*)) | |
| **DATA**($value_1$, ..., $value_n$) | |
| *<darr , row , col >* | : element selection |
| *<darr , $dvect_r$ , col >* | : column selection |
| *<darr , $dvect_r$ , $dvect_c$ >* | : partition selection |
| *<darr , row , $dvect_c$ >* | : ~~column~~ row selection |

| | |
|---|---|
| *array* | : static array |
| *element* | : scalar or array element (static or dynamic) |
| *expression* | : real scalar expression with at least one operator |
| *row, col* | : scalar or array elements |
| *dvect* | : dynamic vector |
| *value* | : scalar literal or element |

Further description of these operations is given in chapter 3

## Matrix Assignments

### Syntax (general form):

[ *dmat* ] = *rhs*

### Symbols:

*dmat* : dynamic matrix (2 dimensional array)

*rhs* : dynamic matrix expression : one of the following:

| | |
|---|---|
| { *dmat* } | : matrix transpose |
| [ $dmat_1$ ]*[ $dmat_2$ ] | : matrix product |
| [ $dmat_1$ ]*{ $dmat_2$ } | : matrix product transpose |
| { $dmat_1$ }*[ $dmat_2$ ] | : matrix transpose product |
| **INVERSE** [ dmat ] | : matrix inverse |
| [ $dmat_1$ / $dmat_2$ ] | : row meld |
| [ $dmat_1$ , $dmat_2$ ] | : column meld |

Further description of these operations is given in chapter 3.

## INCALL statement

### Syntax:

**INCALL** *name* ($arg_1${,$arg_2$...$arg_n$})

### Symbols:

*name* : subroutine name

$arg_1$, $arg_2$ : argument variable

This statement is used to call an ordinary FORTRAN subroutine from a MODEL procedure for the purpose of receiving input from the subroutine. If any of the

argument variables are real, including dynamic reals, they will nominally be in calculus mode in the model. Thus an ordinary call (i.e. from MODEL to MODEL) will transmit the real arguments in calculus mode. INCALL converts the incoming arguments from ordinary mode to calculus mode.

### OUTCALL statement

**Syntax:**

OUTCALL *name* (*exp1*{,*exp2*,...*expn*})

**Symbols:**

| | |
|---|---|
| *name* | : subroutine name |
| $exp_1$, $exp_2$ | : argument expressions |

(or FMODEL)

This statement is used to call an ordinary FORTRAN subroutine from a MODEL procedure for the purpose of transmitting output to the subroutine. Real variables appearing in the argument expressions will nominally be in calculus mode in the model. OUTCALL converts the outgoing variable values from calculus mode to ordinary mode before the call is made.

## 1.3.3. Command Calls

The command call is a class of macro statements which usually translates into a call to a special purpose library subroutine known to the FC translator. It takes the form

*@name* (*arguments*)

where ***name*** is the command identifier, which designates the subroutine to be called. Both the ***name*** and the *arguments* may be translated.

This form may also be used for user-defined subroutine calls. When employed in calculus context (in a MODEL or FMODEL) a command call is translated as an OUTCALL. This is particularly convenient for the modularization of graphical output subroutines for use as "commands".

# 1.3.4. Output Streams and Report Generation

***Output Streams*** - In addition to ordinary FORTRAN I/O, FC supports three output streams for preformatted output reports:

- CONSOLE - 80 column sequential standard output
- SCROLL - 80 column contiguous set of "section" files
- PRINTER - 132 column sequential file

## @STREAM - Routing output reports

At any time during program execution, one of these is designated as the "current" output stream. The current stream is selected by the @STREAM command call, as follows:

**@STREAM(*n*)**

where *n* is an integer expression whose value selects the output stream as follows:

| | |
|---|---|
| negative | : CONSOLE |
| zero | : PRINTER |
| positive | : SCROLL |

DEFAULT ? SCROLL

The user may generate output to the individual streams via FORTRAN WRITE statements by referencing the logical unit numbers assigned to the streams.

The CONSOLE stream is always associated with standard output, so its logical unit number is the traditional FORTRAN unit 6. ←

The PRINTER stream file name is "PRINTER.PRN", and its default unit number is 7. ←

## @OUTDIAGS - Routing diagnostic output

Ordinarily, diagnostic messages are printed to the CONSOLE (standard output) if no @STREAM command has been executed. When a @STREAM command is executed, diagnostic output is also routed along with reports. In order to route diagnostic output separately the @OUTDIAGS command is used:

**@OUTDIAGS(*n*)**

where *n* has the same usage as for @STREAM.

## @OUTMERGE - Merging report and diagnostic streams

The @OUTMERGE command causes the diagnostic stream and the report stream to be merged, so that routing of both streams is subsequently controlled by @STREAM. This command may also be used to switch streams:

**@OUTMERGE(*n*)**

where n has the same usage as for @STREAM.

## @PRINTER - Assigning the PRINTER stream unit number

The unit number for the PRINTER stream may be changed using the command call

**@PRINTER(*unit*)**

where *unit* is an integer.

## @SCROLL - Assigning the SCROLL stream unit number

The unit number for the SCROLL stream may be changed using the command call

**@SCROLL(*unit*)**

where *unit* is an integer.

## @SECTION - Separating SCROLL output into sections

The SCROLL stream is a succession of "section" files having the names SCROLL00.000 to SCROLL99.999, and its default unit number is 8. Also associated with the SCROLL stream is a table-of-contents file named SCROLL.TOC.

The size and significance of SCROLL sections are under user control. SCROLL output may be segmented into sections of arbitrary length by the command call

**@SECTION(*title*)**

where *title* is a character expression. This statement closes the previous section file, opens its successor, and enters the title string in the SCROLL table-of-contents file.

Whenever the SCROLL stream is used for output reports (iteration reports and summary reports), each report is written on a separate SCROLL section, and a report title line is written to the SCROLL table- of-contents.

The output of the SCROLL stream may be perused using a special SCROLLER program, or may be copied to a single file for printing.

***Preformatted Reports*** - Preformatted output reports may be generated by the statements ROWPRINT, COLPRINT, VECPRINT, MATPRINT, TABULATE, GRADPRINT, HESSPRINT, AND PARDPRINT. These statements produce reports on any of the preselected output streams according to their relation to the prior execution of a @STREAM statement.

Report formats for ROWPRINT, COLPRINT, and TABULATE have controllable specifications, as follows:

| Specification | Usage | Preset |
|---|---|---|
| LABELS | Controls printing of variable names | "on" |
| SKIPS | Number of lines to be skipped before each report | 1 |
| MARGIN | Number of blank characters in left print margin | 1 |
| FIELDS | Number of data fields | $5^3$ or $3^4$ |
| WIDTH | Width of data fields | 21 |
| DIGITS | Number of significant digits | 7 |

## USE Statement

### Syntax:

**USE {NO} LABELS**

FORMAT
±X.XXX···XE±XX ?

**USE** *number* **SKIPS**

**USE** *number* **MARGIN**

**USE** *number* **FIELDS** *width* **WIDE**

**USE** *number* **DIGITS**

### Symbols:

*number* : integer literal

*width* : integer literal

This statement is used to specify the various aspects of the report format for the statements ROWPRINT, COLPRINT, and TABULATE.

---

3 PRINTER stream

4 SCROLL or CONSOLE stream

## RESET Statement

### Syntax:

**RESET FORMAT**

This statement is used to restore the preset default values of the format specifications.

## ROWPRINT and COLPRINT

### Syntax:

**ROWPRINT|COLPRINT** {[*title*]} *list*

### Symbols:

*title* : character string

*list* : *element* {, *list*}

*element* : scalar|array|array-element|array-part|expression

These statements cause sequential printing of both the names and values of the designated elements in a report paragraph having the heading:

{title} VARIABLE VALUES .....

ROWPRINT causes names and values to be printed in equivalent positions of pairs of lines making name/value rows. COLPRINT causes names and values to be printed in two columns (name column followed by row column).

## VECPRINT and MATPRINT

### Syntax:

**VECPRINT|MATPRINT** *list*

### Symbols:

*list* : *element* {, *list*}

*element* : scalar|array|array-element|array-part

These statements generate individual reports for each *list* element, in which numeric values are printed to seven figure precision. There is a line eject preceding each report and the sequence of reports is specified by the order of the *list*.

These statements are primarily intended for printing dynamic arrays (which contain their own dimensions). Static arrays may be printed, but only as vectors, the lenghts of which are computed as the volume (product of all dimensions). Where dimensions are passed as arguments, using the asterisk as the last dimension, the asterisk is assumed to represent unity.

Although these statements are not specifically intended for printing scalar variables, if any members of the lists are scalars or array elements, their values will be printed as follows:

SCALAR VARIABLE *name* = *value* .

## TABULATE

### Syntax:

**TABULATE**{[*title*]} *list* {; **LENGTH** *exp*}

### Symbols:

*title* : character string

*list* : *element* {, *list*}

*element* : scalar | array | array-element | array-part

*exp* : integer expression

This statement will print a table of values having the arrays as its columns, from left to right as they appear in the *list*. Each static array or dynamic vector produces a single column, while each dynamic matrix produces a table column for each matrix column. The entire *list* may not generate more columns than will fit the page width of the output stream.

If the arrays have different numbers of rows, the shorter table columns are filled with blanks. The LENGTH option provides a means of limiting the length of the table to *exp* rows.

The format of the TABULATE statement may be controlled by preceding USE statements.

## GRADPRINT, HESSPRINT, and PARDPRINT

### Syntax:

**GRADPRINT|HESSPRINT|PARDPRINT** *list*

### Symbols:

*list* : *element* {,*list*}

*element* : scalar|array element

These statements generate reports consisting of the element value and the selected partial derivative values for each element in the *list*.

GRADPRINT selects the printing of gradient (first partial derivative) vectors,

HESSPRINT selects the printing of Hessian (second partial derivative) matrices, and

PARDPRINT selects printing of both gradient vectors and Hessian matrices.

If used when derivative evaluation is not active, these statements will result in messages stating that derivatives are not available. Also, when variables in the *list* are constants, or not functions of the independent variables, a similar message is displayed.

## 1.4. Special Syntax Rules

Since FC programs are translated into FORTRAN, certain enhancements to FORTRAN are provided, and certain peculiarities due to FORTRAN must be adhered to.

**Semi-colon: Phrase Separator**

Because FORTRAN does not distinguish blanks as token separators, a delimiter is required to separate alphanumeric tokens. Since each phrase of the calculus macro statements begins with a keyword, a semi-colon is required to prevent this keyword from being concatenated to the trailing identifier of the preceding phrase. Thus the somewhat awkward appearance of semi-colons in macro statements is necessary.

**Colon: Statement Separator**

The colon is used as a statement terminator, allowing multiple statements on a line. This will not conflict with FORTRAN usage of the colon in character expressions, since colons only appear parenthesized. The other use of colons (also parenthesized) in DIMENSION statements is disallowed as indicated below.

## 1.5. Special Restrictions

Since the FC translator must translate all of FORTRAN syntax, certain FORTRAN 77 embellishments have been deferred as a matter of lower priority. A case in point is lower and upper bounds in DIMENSION statements. The use of them will cause a syntax error. The inclusion of them in later versions is a matter of demand priority.

**Type Restrictions**

The most serious restriction is the lack of a COMPLEX type. This will be rectified in the next major release.

In the FC implementations on 32-bit machines, there is no REAL*4 type. This will cause a syntax error. Since calculus reals require 64-bits, the allowance of 32-bit reals can potentially corrupt common storage when proper type declarations are inadvertently left out of a subroutine somewhere. The errors that result from common corruptions are notoriously difficult to locate. It is best therefore to eliminate the use of REAL*4.