Comments/corrections from Irby

Inter-Office Memorandum - DRAFT

To: **Distribution** Date: **February 11, 1977**

From: **C. Irby, T. Shetler, and C. Simonyi**

Subject: **Programming Conventions** Organization: SDD/SD

Keywords: **Programming Conventions, Policies, Procedures**

Filed on: tonimemo.bravo on Toni Pack One

Attached is the draft of the Programming Conventions subsection (2.2) of the Software Development Procedures section of the SDD Policies and Procedures document.

The concept of a *Programmer's Notebook* is introduced in the attached subsection and refers to the collection of reference and training materials that every programmer in SD should have. While this notebook is not intended to replace any existing or planned documentation, it is considered a necessary repository for related memorandum, notes, etc. The outline and content of this notebook is not discussed here because it is more appropriately addressed in the context of training or reference materials for the staff than in a discussion of programming conventions, though materials that are described as in the Programmer's Notebook will simply be attachments that follow the subsection. The exception is the reference to good coding examples which are still being sought. A "Programmer's Notebook" already exists whether it is in a formal format or in a form each individual defines for himself. (Perhaps someone already has created such an object for himself and with a few modifications, we could adopt it for SD?)

Please read, use, and prepare comments on these Programming Conventions. We would like to have the first final form of this document available for general distribution by the end of February. If you have any questions, please contact one of us. There will be a meeting during the week of February 28th to review comments based on your trial use of these conventions.

Distribution:

B. Ayres
G. Benedict
L. Bergsteinsson
I. Clark
D. DeSantis
J. Frandeen
E. Harslem
P. Heinrich

Charles Simonyi, there are several inconsistencies in your sample relative to the general conventions. I noted the ones I found. Toni, I will edit in these changes if you tell me the Maxc file name.

R. Johnsson
D. Liddle
M. Lorous
B. Malasky
W. Maybury
R. Metcalfe
B. Parsley
R. Purvy
W. Shultz
R. Sonderegger
D. Stottlemyre
R. Sweet
J. Szelong
C. Thacker
T. Townsend
D. Wallace
J. Wick

2.0 SOFTWARE DEVELOPMENT PROCEDURES

2.1 Documentation Conventions

This section will be completed at another time with another set of text.

2.2 Programming Conventions

Introduction:

The purpose of these conventions is to support the software generation effort so the code produced will:

- o Facilitate the creation of the software components of the OIS products.
- o Be portable among the technical staff.
- o Be extensible and maintainable over the life of the product.

In addition, we expect these conventions to be easy to train new staff members to use.

The conventions presented are, for the most part, not a radical deviation from many practices of our experienced staff, nor do they diverge dramatically from the literature on accepted software engineering conventions and practices. Adherence to this set of conventions is expected to have long-term gains in the development and on-going maintenance of software that should offset any short-run setbacks that result from modifying existing code, modifying work habits, or adjusting to different coding rules.

The three areas of programming conventions described are: format conventions, naming conventions, and coding conventions. The concept of general and special conventions underlies the description of these Programming Conventions. *General Conventions* apply to all the code generated and are explicitly defined in the description of each area. *Special Conventions* apply to logical subsets of the software development effort -- for a specific system, such as Pilot -- and the description of currently available special conventions will be included in ***The Programmer's Notebook***. Templates are predefined form files which contain the skeleton for program or data modules. The skeleton includes program delimiters such as PROGRAM .... END, and it may also contain placeholder text which can be replaced by actual names. Dictionaries contain both rules for generating names and definitions for special technical terms or words with specialized meaning. Templates and dictionaries can be enhanced by descriptions of specific coding rules to which a project will adhere. Existing templates, dictionariees, and supplimental coding rules are contained in ***The Programmer's Notebook*** (see attachment A, ~~B~~ and B for Diamond, Mesa Runtime, ~~and~~ Tools conventions, respectively).

Special conventions are not a way to subvert general conventions. They are well defined rules to guide the programming activity and are implemented through the definition and description of specific coding conventions, programming templates, and dictionaries, which identify structures, naming rules, and definitions of technical or special terms that apply to the software development effort. The staff on a particular project may create a set of special conventions for the system they are working on if their needs cannot be met by using a set (or subset) of existing special conventions. We expect new sets of special conventions to be introduced infrequently. Special conventions must be approved and published for inclusion in *The Programmer's Notebook.*

Since the reader is assumed to be familiar with the programming language documentation, references to language characteristics are not included in this section.

(this does not include italics or bold face)

Format Conventions

The objective of formatting conventions is to facilitate readable code. The general ~~[illegible]~~ conventions that apply to formatting are:

LB:85
PB:90
R:5~~[illegible]~~27
X:6
Y:12.
(don't use <control-Q>) or <control-O>
nesting: 36

- o The editor used to create code is BRAVO release 6.0 or a follow-on editor.

(A) use of more than one font is discouraged for operational reasons. ~~We~~ An attempt ~~will be made~~ to create a font with standard user.cm settings.

- o Each statement is a separate BRAVO paragraph.
- o All indentation is done using the Looks nesting command and preset margins. with 2nd-last line left margin 5 pts. in (90)
- o **Comments:** Comments appearing at the right of some construct refer only to that construct, except for comments appearing to the right of a BEGIN which apply to the block. Longer comments are in paragraph form and refer to the program text which follows. (Do Not begin each line with "--" as this makes updating difficult.)
- o Spacing:

  1. A space should be placed after a comma, semicolon, or colon, but not before.

  2. No spaces should be placed around brackets or parentheses.
  Exceptions to this are: a) there should be spaces between declaring words such as RECORD, PROCEDURE, and RETURN and the adjacent brackets; and b) there should be equal amounts of space next to matching brackets and parentheses that are hard to spot.

  3. Equal amounts of space should be placed on each side of ← or any binary operator that connects two lengthy expressions.

- o Indentation and Line Breaking

  1. Use ~~indentation~~ Bravo nesting commands, not tabs.

  2. Write no more than one statement on a line, except where several

short statements are logically one.

```
temp ← x;  x ← y;  y ← temp;
WriteString[s]; WriteChar[CR];
```

3. Indent the labels of a SELECT (including the ENDCASE) one level, and the statements a second level (unless a statement will fit on the same line with the label).

```
SELECT e FROM
        case1 => s;
        case2 =>
                lengthy
                 statement;
        case3 =>
                BEGIN
                ...
                END
        ENDCASE
```

4. Indent one level for the statement following a THEN or ELSE (unless it fits on the same line). Put THEN on the same line as IF, and indent ELSE the same amount as IF. If the ELSE is followed by another IF, write both on the same line.

```
IF condition THEN consequence ELSE alternative

IF condition THEN consequence
ELSE alternative

        or

IF condition1 THEN consequence

ELSE IF condition2 THEN alternative

ELSE
        BEGIN
        ...
        END
```

space
✓
5. Consider using a SELECT TRUErather than a string of ELSE IFs for readability.

```
SELECT TRUE FROM
        condition1 =>
                lengthy
                        consequence
        condition2 =>
                lengthy
                        alternative
        ENDCASE =>
                BEGIN
                ...
                END
```

6. A compound should be either all on one line or one item per line. A

compound should be indented from the surrounding material.

```
    BEGIN s; s; END

    BEGIN
    s;
    s;
    END


    DO s; s; ENDLOOP

  <- DO
        s;
        s;
        ENDLOOP
```

7. A record declaration should be either all on one line or one identifier list per line with the brackets on separate lines.

```
    Bla: TYPE = RECORD[x: Dictionary, y, z: INTEGER];

    Bla: TYPE = RECORD
            [
            x: Dictionary,
            y, z: INTEGER
            ];
```

8. A procedure declaration should have the BEGIN and END on separate lines.

```
    SomeProc: PROCEDURE[x: Ta, y, z: Tb] RETURNS [Tc] =
            -- ~~This may do something~~ Function: . . .
            BEGIN
            ...
            END          (which will be automatically ...)
```

9. If a procedure declaration will not fit on about one line move the RETURNS clause to a second line (indented one space). If either the PROCEDURE [arglist] clause or the RETURNS [retlist] clause will not fit on a single line, break it into one line for each member of the arglist or retlist, respectively, with the enclosing brackes, [], on separate lines.
first Parameter . . .

```
    SomeProc: PROCEDURE [x: Ta, y, z: Tb]
     RETURNS [Tc] =
            BEGIN          expand identifyers to fill line
            ...
            END
```

The following illustration assumes the arglist will not fit on one line or that the author wishes to comment the arguments.

```
            SomeProc: PROCEDURE
                    [
                    SomeProc: PROCEDURE [x: Ta, y, z: Tb]  ]
                     RETURNS [Tc] =                        ] -e
                            BEGIN
                    ...
                    END  7e
                x: Ta,
                y, z: Tb
                ]
            RETURNS [Tc] =
                BEGIN
                ...
                END
```

10. A long statement should be broken into many lines by: a) rewriting it as several statements, b) indenting the arguments of the main procedure call or record constructor as one would indent the components of a long record declaration, or c) breaking the statement into lines at reasonable places and indenting the continuation lines one space in from the initial line. [illegible]

Example of a rewrite of several statements:

```
        a ← TheFirstProc[TheSecondProc[...], TheThirdProc[...]];

        temp2 ← TheSecondProc[...];
        temp3 ← TheThirdProc[...];
        a ← TheFirstProc[temp2, temp3];
```

Example of indenting the arguments of the main procedure call or record constructor as one would indent the components of a long record declaration:

```
        a ← TheFirstProc
                [
                TheSecondProc[...],
                TheThirdProc[...]
                ];
```

~~e~~Example of breaking the statement into lines at reasonable places and indent the continuation lines one space from the initial line:

```
        a ← TheFirstProc[ TheSecondProc[...],
         TheThirdProc[...] ];
```

The special format conventions are: —e

A

o There are no **Standard Fonts.** A project creating a system, such as PILOT, may decide upon a standard font and should include this in the programming documentation for the system.

9 o Use of specialized text features, other than the indentation command of BRAVO, should be minimized (for example, forcing page boundaries).

Naming Conventions

The objective of naming conventions is to provide a meaning to the names in the code.

The general conventions are:

o **Capital Letters:** Type, procedure, label, module, and signal names start with a capital letter, all else starts with a lower case letter.

o **Compound names:** Each component is captialized (e.g., maxFilePage) in compound names.

o **Procedure with side effects** (such as changes to the abstract state of its object): The procedure name begins with a verb (except "is" and "has"); otherwise, verb/noun naming is preferred. but not required.

word: [illegible] Record [name: STRING, ~~valu~~ meaning: Integer;

```
    dictionary.Insert[word]
    stream.Get[]
    pool.ObtainBufferWhichHas[virtualAddr]
As opposed to:          Add   Delete
    dictionary.IsEmpty[word]--use dictionary.~~Clear~~Word[word]
    dictionary.Has[word]--use dictionary.~~Set~~Word[word]
    dictionary.Meaning[word]--use dictionary.SetMeaning[word]
    stream.CurrentPos[i]--use dictionary.SetPos[i]
```

invert

o **Abbreviations:** If used, the standard prefixes are:

p*	pointer to a *
i*	index of something in a *
l*	length of a *
n*	number of *'s

Any other special abbreviations appear in a section in a dictionary.

o **Signals and Errors:** Precede an invocation of a signal or error by the word SIGNAL or ERROR.

Special naming conventions may appear in a project dictionary and include:

o Additional rules for contructing names.

o Technical or special terms that are used for naming.

Coding Conventions

Except for the restrictions contained in the language manuals which reflect language constraints, the coding rules to which we expect the projects to adhere can be found in books or articles such as *The Elements of Programming Style* (Kernighan and Plaguer), and the redundancy of repeating those in this section is not necessary. ok ~~Instead, one (two or three)~~ acceptable reference(s) of programming wisdom will ~~should~~ be selected and made available through ~~in~~ our Secretaries ~~Development Environment~~. (~~After this initial experiment, a copy of the agreed upon~~ source ~~should be given to each staff~~ member when he ~~joins the project~~.) Copies of good illustrations of practices we would like to see (*coding literature*?) will be contained in ***The Programmer's Notebook***.

APPENDIX A: MESA Formatting Rules for DIAMOND

"Rules of programming style, like those of English, are sometimes broken, even by the best writers. When a rule is broken, however, you will usually find in the program some compensating merit, attained at the cost of the violation. Unless you are certain of doing as well, you will probably do best to follow the rules." (Kernighan/Plaguer/Strunk/White)

1. No special formatting (bold, fonts, etc.) except in comments if there is a real need.

2. Tabs are used for indentation (standard tab width is 55pt).

3. Punctuation rules are similar to those in the Mesa manual:

   a space (or carriage return) after a comma, semicolon, or colon and none before. However in declarations, a colon may be followed by a tab (see below).

   no spaces (immediately) inside brackets or parenthesis. Space should be left between reserved words and [: RETURN [];

   spaces should be written around binary operations at the outermost level, e.g. ← in an assignment statement, < in a conditional statement, + in a ← a + 1; When embedded in expressions, parameter lists and so on, it is recommended (but not required) that spaces be omitted.
   (as in n ← IF (n←n+1)<nMac THEN n1 ELSE n2;)

   spaces are not written around . and ..

4. In general, indentation should be used sparingly, only when dictated by the logical structure. If a long identifier or expression pushes a delimiter beyond the correct level of indentation, that delimiter should be written on a new line. Indentation rules for statements are (almost) according to the Mesa manual (p137). These rules are best summarized in the following examples:

```
IF a < aMac THEN a ← aNil;    -- short statement

IF a < aMac THEN
        BEGIN
        a ← aNil;
        END
ELSE IF a < aMax THEN
        BEGIN OPEN x;
        a ← aNil;
        EXITS
        End =>     a ← a1;    -- Label naming is explained below
        NoMoreEntries =>
                BEGIN
                a ← a2;
                END;
        END; -- OF OPEN x

FOR a IN [0, aMac)
        DO
        PrintA(a);
        ENDLOOP;

SELECT a + da FROM
```

indent

```
aNil =>   BEGIN
          a ← 0;
          END;
aFirst => ~~NULL~~;
IN [aFirstfree..aLastfree] =>
          a ← aNil;
ENDCASE;
```

5. Comments may appear to the right of some construct, preceded by a tab, and then they may refer only to that construct. Longer comments appear on separate lines, without indentation, referring to the program text which follows.

6. In other declarations, the type is preceded by a tab:

```
a:              INTEGER = a1;
a, ta:          INTEGER;
bmpicergcaLast:         INTEGER;
```

Records (and procedures, see below) are declared as follows:

```
B:          TYPE = MACHINE DEPENDENT RECORD
                      [a:         A,
                      pa:         Pa,
                      aLastfreemac:         A
                      ];
C:          TYPE = MACHINE DEPENDENT RECORD
                      [a:         A ← aNil,
                      VARIANT:
                      SELECT vr: VrC FROM
                      vrC1 =>    [b:         B,
                                 d:          D
                                 ],
                      vrC2 =>    NULL,
                      ENDCASE
                      ];
```

Note that the variant part has the standard name VARIANT by convention. The naming of the variants is explained below.

If the type includes indenting (procedure or record type), type should start strictly one level indented from the (possibly long) name:

```
LongProcedureName:
            PROCEDURE
                      [a:         A,
                      b:          B,
                      c:          C,
                      d:          D
                      ];
```

7. Procedure declaration format is defined in the Module template. Note that in the common, non-local, procedures, some of the outermost levels of indentation are omitted by convention, so that the procedure bodies may start without any indentation.

Procedures with simple parameter lists may be declared in a single line:

```
PROCEDURE [a: A] RETURNS [A] =
```

while more complex procedures should be declared in the same format as two records: one containing the parameter list, and the other the returned values.

```
PROCEDURE
        [a:     A,
        b:      B,
        c:      C,
        d:      D
        ]
RETURNS
        [A,
        B,
        C
        ] =
```

8. Pack and Module formats are described in the module templates. Emphasis to procedure names in heading comments is given by expanding and capitalizing I N T H I S W A Y. Characters originally capitalized should be preceded by en extra blank: X Y I N I T.

9. Continuation lines are indented by two blanks. For example:

```
a ← a1 + a2 + a3 +
  a4 + a5;
```

use single paragraph with CR

10. Variable and constant names start with a lower case letter; type, procedure, label, module, and signal names start with a capital letter. Reserved words (even if reserved by convention) are fully capitalized.

11. Variable and constant names consist of a *type tag* and 0 or more *modifiers.* The type tag may not contain capitals. The first letter of each of the modifiers (if any) is capitalized. Some type tags are standards, others may be defined in the meta-programs. New type tags may be constructed using standard constructors which are described below. Type tags are usually defined to be very short (two or three letters) so that the lengths of constructed tags remain manageable.

The type tag denotes the type of the variable or constant, of course. The name of the type is just the type tag. Note that the first letters of type names are capitalized. For example, in:

cpMac: Cp

the "cp" is the tag, "Mac" is a modifier. "Cp" is the name of the type.

Modifiers are used to distinguish variables in some scope (record, local, or global frame) which have the same type. Single digits or numbers may be used as modifiers in some cases. Some modifiers imply standard semantics, as described below. If there is just one value of a given type in some scope, the modifier should be left empty; e.g.

write GetObject(nh: Nh) instead of GetObject(nhObject: Nh)

If constructed tags get too long and cumbersome, a new tag may be defined to stand for the complex type.

Painting (particularizing) of types is done by the modifier. For example the nh of a pl is written as: nhPl. This is not a type construction, therefore Pl is capitalized. The type of nhPl is Nh, of course.

To express repeated painting and modifying of quantities, the modifiers should be concatenated in the order of construction (the earliest modifier first). Eg: hrFreeMin: an hr, painted Free; then the hrFree modified by Min.

12. Standard types (and constants) are defined in StandardDefs:

```
F:       TYPE = BOOLEAN;  -- Flag
W:       TYPE = CARDINAL; -- Unsigned word
INT:     TYPE = INTEGER;  -- Handy abbreviation, not a tag
VrF:     TYPE = {vrTrue, vrFalse};     -- Used for certain variant fields
InNil:   TYPE = [0..0);        -- Domain for variable size arrays

-- Powers of 2, to be used in field definitions
w2to0:    W = 1;
w2to1:    W = 2;
w2to2:    W = 4;
...
w2to15:   W = 100000B;
```

13. The standard type constructions are the following: (X and Y denote arbitrary tags, througout).

pX   type is declared as: ORDERED POINTER TO X
~~Pointer to X~~. Let ↑ be the indirection operation. pX↑ is then an X.

aX   type is declared the same as X, with initial POINTER removed
A structure pointed to by X. paX would be an X. Used for declaring larger records. a in conjunction with mp or rg (see below) is used to declare arrays.

bX   type is declared as: INT ??
Based X. There exists some pointer Y such that Y+bX is an X. Used, for example, to name relative pointers to fields in records which have fields of varying sizes.

bXY   type is declared as: INTEGER
Based X. Same as above, with the type of the base given.

cX   type is declared as: W or INTEGER   cCo ?
Counts instances of X (not necessarily all instances). For example, cco could be a counter counting colors which appear in a graph (assuming the type definition co = {coRed, coGreen, coBlue}).

dX   type is declared as: INTEGER
First difference of X. X + dX is an X. ~~DESCRIPTOR FOR ?~~

mpXY   type is declared as ~~POINTER TO~~ ARRAY InX OF Y
Array (map) with domain X and range Y. The domain InNil is specified if the array is of variable size. mpXY↑[X] is a Y. Note that ampXY is an ARRAY InX OF Y, ampXY[X] is a Y.

InX   type is declared as [0..XMax)
This is a type construction only, for use in array declarations and loops. There need not be any values of type InX. "In" stands for "index", "interval", and "IN" (as in FOR ALL x IN InX DO). See below for XMax.

rgX   short for mpiXX, array with domain iX and range X.

iX   type is declared as: INTEGER
Domain of rgX. Not defined to be an interval, so that XNil, XMac, and XMax, all of which are, strictly speaking, outside of the domain of rgX, can all be declared IX. Same is true for other index types, that is for any tag X which appears in the domain of a map mpXY.

lX   type is declared as: INTEGER
Length of an instance of X in words.

tX   temporary X, the same type as X. A somewhat unelegant but efficacious device to distinguish between parameters and local (temporary) variables in procedures, without

APPENDIX A, continued: DIAMOND Template

Pack: set of Mesa modules implementing an abstraction.
Reasons for more than one module/pack:

1. single module may be too small to hold all operations
2. the frequency of usage (and core residency) may be different for groups of operations. In particular, initialization should, in most cases, be separated from the frequently used operations.

A Pack implementing the Abstraction Xy would consist of the following modules:

Xy	containing the declaration of the Xy structure and the most frequently used operations
Xy1	no declarations, more operations of the data structure. Reasons for split from Xy, as above.
Xy2	2nd split, and so on.
XyInit	initialization and binding code
XyTest	test output and check procedure code (same format as xy1)
XyDefs	contains the type declarations and external interface definition (externally used operations in Xy, Xy1 etc.)
XyPriv	private type declarations and internally used operations in Xy, Xy1 etc.

Module named XX would be stored in file xx.mesa. The command file to be executed when xx.mesa is changed is xx.cm. Under current operational procedures, the command files are created by the programmer at the same time as the corresponding source file, and contain the following:

xy.cm	?
xy1.cm	?
xyinit.cm	?
xytest.cm	similar to xy1
xydefs.cm	?
xypriv.cm	?

no!

Module templates follow. The standard formatting for Mesa files is (single paragraph/module):

Margins: L: 85pt R: 580pt P: 90
Lead: X: 1pt Y: 0pt
Tabs: Plain-tabs: 55pt
Font: 0

Other formatting rules are described in mesa-rules.memo.

The suggested use of the template is as follows: start creating the pack by editing the template, scrolling back and forth between definitions, declarations, initialization, and code. For smaller packs, it may be worthwhile to save the pack as a whole on a .pack file. When ready to compile, select a module by paragraph selection, copy it into a new window and write it on the correct .mesa file. Further edits may be made on the saved .pack or .mesa file (but not interchangeably) as it is convenient.

no

```
-- X Y

DIRECTORY
-- Pack Specific
          XyDefs:    FROM "xydefs",
          XyPriv:    FROM "xypriv",
-- Module Specific
          StandardDefs:          FROM "standarddefs";

DEFINITIONS FROM
-- Pack Specific
          XyDefs,
          XyPriv,
-- Module Specific
          StandardDefs;

-- END OF DIRECTORY

Xy:
PROGRAM =
BEGIN

-- Pack Data Structure

nm1:      Nm;

-----------------------------------------------------------
-- Pack Operations
-----------------------------------------------------------
-- O P   O N E

OpOne:
PUBLIC PROCEDURE [nm1: Nm, nm2: Nm] RETURNS [Nm] =
BEGIN
nm3:      Nm;
Proc:     PROCEDURE =
          BEGIN
          END; -- OF Proc

IF nm1 THEN
          BEGIN
          END;

END; -- OF OpOne

-----------------------------------------------------------
-- O P   T W O

OpTwo:
...
END; -- OF OpTwo

-----------------------------------------------------------
-- Private Operations
-----------------------------------------------------------
-- O P   T H R E E
```

indent

```
OpThree:
PROCEDURE [nm1: Nm, nm2: Nm] RETURNS [Nm] =
BEGIN

END; -- OF OpThree

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

END. -- OF Xy
```

```
-- X Y 1

DIRECTORY
-- Pack Specific
            XyDefs:     FROM "xydefs",
            XyPriv:     FROM "xypriv",
            Xy:         FROM "xy",
-- Module Specific
            StandardDefs:           FROM "standarddefs";

DEFINITIONS FROM
-- Pack Specific
            XyDefs,
            XyPriv,
-- Module Specific
            StandardDefs;

-- END OF DIRECTORY

Xyl:
PROGRAM [xy: POINTER TO FRAME [Xy]] SHARING Xy =
BEGIN OPEN xy;

-----------------------------------------------------------
-- Pack Operations
-----------------------------------------------------------
-- O P   F O U R

OpFour:
...
END; -- OF OpFour

-----------------------------------------------------------
-- Private Operations
-----------------------------------------------------------
...

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

END. -- OF Xyl
```

```
--X Y  I N I T

DIRECTORY
-- Pack Specific
            XyDefs:     FROM "xydefs",
            XyPriv:     FROM "xypriv",
            Xy:         FROM "xy",
            Xy1:        FROM "xy1",
            Xy2:        FROM "xy2",
            XyTest:     FROM "xytest",
-- Module Specific
            ControlDefs:            FROM "controldefs",
            StandardDefs:           FROM "standarddefs";

DEFINITIONS FROM
-- Pack Specific
            XyDefs,
            XyPriv,
-- Module Specific
            ControlDefs,
            StandardDefs;

-- END OF DIRECTORY

XyInit:
PROGRAM [nm: Nm] SHARING Xy, Xy1, Xy2, XyTest =
BEGIN

SetBindingEntry:        EXTERNAL PROCEDURE [frame, entry: GlobalFrameHandle];

------------------------------------------------------------
-- Private Operations
------------------------------------------------------------
--I N I T

Init:
PROCEDURE =
BEGIN

-- Local Declarations

nm:         Nm;
yzInit:     POINTER TO FRAME [YzInit]; -- "Owned" Module
xy:         POINTER TO FRAME [Xy];
xy1:        POINTER TO FRAME [Xy1];
xy2:        POINTER TO FRAME [Xy2];
xytest:     POINTER TO FRAME [XyTest];
xyInit:     POINTER TO global FrameBase;

-- Remove Self From Binding Path

xyInit ← REGISTER[Greg];
SetBindingEntry[xyInit, xyInit.bindlink];

-- Initialize "Owned" Modules
```

```
yzInit ← NEW YzInit[...];
BIND yzInit; START yzInit;

-- Instantiate Pack

xy ← NEW Xy[];
xy1 ← NEW Xy1[xy];
xy2 ← NEW Xy2[xy];
xytest ← NEW XyTest[xy];
SetBindingEntry[xyInit.ownerlink, xyInit.bindentry];
BIND xy;
BIND xy1;
BIND xy2;
BIND xytest;

-- Initialize Pack Data Structure

BEGIN OPEN xy, xy1, xy2, xytest;

(initialization code)

END; -- OF OPEN xy

END; -- OF Init

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

Init[];

END. -- OF XyInit
```

```
-- X Y   D E F S

DIRECTORY
          StandardDefs:            FROM "standarddefs";

DEFINITIONS FROM
          StandardDefs;

-- END OF DIRECTORY

XyDefs:
DEFINITIONS =
BEGIN

-- Abstractions

Nm:       TYPE = INT;

nmNil:    Nm = -1;

-- Operations

OpOne:    PROCEDURE [nm1: Nm, nm2: Nm] RETURNS [Nm];

END. -- OF XyDefs
```

```
-- X Y   P R I V

DIRECTORY
            XyDefs:                 FROM "xydefs",
            StandardDefs:           FROM "standarddefs";

DEFINITIONS FROM
            XyDefs,
            StandardDefs;

-- END OF DIRECTORY

XyPriv:
DEFINITIONS =
BEGIN

-- Abstractions

Nm:         TYPE = INT;

-- Operations

OpThree:    PROCEDURE [nm1: Nm, nm2: Nm] RETURNS [Nm];

END. -- OF XyPriv
```

```
--S T A N D A R D   D E F S

StandardDefs:
DEFINITIONS =
BEGIN

-- Abstractions

F:        TYPE = BOOLEAN;
W:        TYPE = UNSPECIFIED;
Ch:       TYPE = CHARACTER;
Sm:       TYPE = STRING;
INT:      TYPE = INTEGER;
VrF:      TYPE = {vrFalse, vrTrue};
InNil:    TYPE = [0..0);

w2to0:    W = 1B;
w2to1:    W = 2B;
w2to2:    W = 4B;
w2to3:    W = 10B;
w2to4:    W = 20B;
w2to5:    W = 40B;
w2to6:    W = 100B;
w2to7:    W = 200B;
w2to8:    W = 400B;
w2to9:    W = 1000B;
w2to10:   W = 2000B;
w2to11:   W = 4000B;
w2to12:   W = 10000B;
w2to13:   W = 20000B;
w2to14:   W = 40000B;
w2to15:   W = 100000B;

END. -- OF StandardDefs
```

Appendix A, continued: DIAMOND Dictionary

To Be Supplied Next Week By C. Simonyi

[illegible]

RACK

see hz.spec, hzdefs.mesa

hz	heap global frame
ahf	block descriptor
hf	pointer to ahf
hff	pointer to free ahf
hfn	pointer to normal ahf
hr	pointer to block
h	pointer to hr = pointer to pointer to block
hx	convertible finger
rghx	finger table
ihx	index into rghx
nh	pointer to hx
hzace	hz-type ace
hzce	pointer to hzace

CACHES

see ca.spec, cadefs.mesa

ca	cache global frame
nw	name of whatever is being cached
lu	last used time
ace	cache entry descriptor
ce	cache entry, pointer to ace
rgce	cache entry array
ice	index into rgce
rgace	descriptor array

OBJECT SWAPPING

see nsg.spec, nsgdefs.mesa

ns	name of swapable object
sns	object segment
rgsns	segment array
isns	index into rgsns
dw	segment chunk size

PAGE SWAPPING

see pbg.spec, pbgdefs.mesa

np	name of page
snp	page segment
rgsnp	segment array
isnp	index into rgsnp
fl	file descriptor
rgfl	array of fl
ifl	index into rgfl
fp	file pointer
fpn	file page number
rw	relative word in page
frw	relative word in file
pb	page buffer = [array[256],np]
hrpb	pointer to pb
hpb	pointer to pointer to pb
pbgace	pbg-type ace
pbgce	pointer to pbgace
fh	file handle
sh	stream handle

Appendix B: TOOLS Special Convention Setion

To Be Supplied Next Week By C. Irby

Appendix C: Mesa Runtime Special Convention Setion

To Be Supplied Next Week By C. Irby