Inter-Office Memorandum

To	Wendell Shultz, David Liddle	Date	July 5, 1977
From	Charles Irby	Location	Palo Alto
Subject	**Final Draft of Programming Conventions (hopefully)**	Organization	SDD/Sd

XEROX

Filed on: <Irby>PC-Cover.bravo

Attached, please find the final draft (hopefully) of the Programming Conventions section of the SDD Policies and Procedures. I apologize for the long delay -- I was forced to give higher priority to other activities.

In the reviews of this set of conventions, several things came up that are not properly part of the conventions section but do require management attention. They are:

- The Mesa Source Formatter should be developed as soon as possible. Much of the effect of these conventions will not be realized until the formatter exists. Many bad habits and much improperly formatted code may result if the formatter is delayed. (I regret having introduced an additional delay by failing to publish these conventions sooner.)

- An improved version of Bravo is being developed. However, maintenance of this central tool is still uncertain, I recommend that the Tools Group accept maintenance responsibility for Bravo 7.0 for SDD (and no one else!). Also, a program should be developed that will print Bravo files of the stylized form we will be using directly from Maxc, the Alto exec, or the Program Librarian. This program should be released along with the Source Formatter. Again, I recommend that the Tools Group undertake this development.

- Individual software projects are being allowed the priviledge of specifying their own naming conventions and other specialized conventions. In return, each such project should publish the naming and other special conventions it is using -- and keep this up to date. Both the Diamond and Tools Groups have published such conventions. SDD Management will have to see to it that this is done with all projects. It has been suggested that either the Source Formatter or another program check the naming used within programs against the associated naming dictionary for the project. I think this warrents investigation.

- To give the Source Formatter the ability to treat comments and blank lines reasonably, it may be necessary to add a unique right comment delimiter to Mesa. This need will become more apparent when detailed work on the formatter is underway. This, of necessity, is a pure addition to the Mesa language.

- A separate "Guidelines for Producing Efficient D0/Mesa Code" document should be published by the Mesa Group. As new performance parameters

are discovered or old ones changed, the document should be updated and re-distributed to our programmers. I do not mean to imply that our programmers should sidestep Mesa and super-optimize their code. However, if we know that, for example, passing more than N words of parameters to procedures causes significantly slower procedure calls, we may be able to structure interfaces to avoid this performance problem.

- Management should seek out "good" coding examples and distribute them to the programmers. In particular, examples showing good use of Mesa types and signals would be very helpful and would contribute to the long-term maintenance of our software.

- An option to the cross reference program that reports possible signals that can result from calling a procedure should be developed.

cc: SD Managers, S. Wallace, J. Wick, D. Sweet, R. Ayers, T. Shetler

2.0 SOFTWARE DEVELOPMENT PROCEDURES

2.2 Programming Conventions

2.2.1 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, a program (described below) will be provided to automatically format source files according to these conventions. Consequently, 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: formatting 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 below. *Special Conventions* are in addition to the General Conventions and apply to logical subsets of the software development effort -- for a specific project, such as Pilot. Each project must publish any special conventions it uses. Special Conventions should include templates and dictionaries. 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.

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

2.2.2 Formatting Conventions

The objective of formatting conventions is to facilitate readable code. Format conventions adherence is achieved through the use of a program called the Mesa Source Formatter (see below). This program creates a new source file in the correct format.

The Mesa Source Formatter

The Mesa Source Formatter will be used to establish uniformity of Mesa programs with minimal programmer inconvenience. The following goals should be met:

- Standardize fonts to ensure reasonable consistency when the program is viewed, printed, and edited.

- Establish the horizontal and vertical spacing of the program in a way which reflects its logical structure.

The Mesa Source Formatter converts its input source file into a source file that conforms to the SDD Standard. It may be automatically called by the Program Librarian when a source file in checked in. It will utilize the parser from the Mesa compiler to ensure completeness and correctness.

Fonts and Formatting

The formatter produces a Bravo-compatible file as its output. It utilizes only a subset of Bravo format which is described in detail below. However, some general comments should be made here.

Horizontal spacing is done by setting the left margin. The formatter adjusts the paragraphing so that a series of statements at the same level form a single paragraph. At most two fonts are used (regular and small); identifiers containing all capital letters (notably Mesa keywords) appear in the smaller font. The names of procedures are bold where they are defined.

Comments are set to italics, to help distinguish them from program text. Multiple blank lines before or after comments are replaced by a single blank line.

Standard User.Cm

Because disk space is at a premium (for one-disk users) and because of Bravo operational difficulties, the User.Cm on the Basic Mesa Disk will be set up with just one Alto font and two (optional) Ears fonts. To maximize the content of a page, fonts slightly smaller than standard are used. The standard Mesa fonts are as follows:

```
Font:0 Helvetica 8 MesaFont 10
Font:1 Helvetica 6 MesaFont 10
```

MesaFont is a copy of Helvetica10.Al but with eight point capital letters. This allows the program to be typed in and edited using a single font; it will still resemble its eventual two font form. An eight point version of this font will also be produced so that look hardcopy will work.

The use of a single font is optional, but recommended. {If Bravo performance with multiple fonts is improved, then this recommendation is withdrawn.} The formatter will always output the program using two fonts, so that it will hardcopy correctly. We have collapsed them into a single font in User.Cm to improve Bravo's performance and conserve disk space.

Spacing

The following spacing conventions are a refinement of the rules in the *Mesa Language Manual.* In general, there are no spaces before or after atoms containing only special characters. Exceptions to this rule are as follows:

A space or carriage return follows a comma, semicolon, or colon (but none before). A space precedes all the interval constructions.

A space precedes a left square bracket when the bracket follows any of the keywords RECORD, MACHINE CODE, PROCEDURE, RETURNS, SIGNAL, PORT, PROGRAM, and DATA.

Spaces surround the left-arrow operator, except when the assignment is embedded in an expression.

The exclamation point (enabling) and equal-greater (chooses) operators are always surrounded by spaces. This also applies to equal signs used in initialization, and to asterisks used in place of variant record tags.

The equivalent of about five spaces are used for each level of indenting. The User.Cm on the Basic Mesa Disk will be compatible with this nesting and other attributes of the resulting files so that editing will be as straight forward as possible. If a comment appears to the right of a statement, it is preceded by two spaces; the entire comment should fit on the line in this case for good legibility.

We have not yet investigated a number of heuristics which the formatter might use in spacing complex expressions (for a suitable definition of "complex"). For example, in complex conditionals, spaces on each side of the relational operator improve readability. The general rule is that spacing should follow the precedence of the operators; the details of this rule will need to be worked out as part of the implementation. This mechanism will determine line breaks for multi-line expressions.

Structure

The remaining job of the formatter is to determine the indenting structure of the program. While there is general agreement on the indenting rules for the bodies of compound statements, there are several different rules currently in use for the placement of the brackets surrounding the compounds. (The brackets in Mesa include (), [], BEGIN-END, DO-ENDLOOP, and FROM-ENDCASE.) Where there are conflicting conventions, we have chosen the one which maximizes the amount of information on a page. For example:

```
Record: TYPE = RECORD [
    field: Type,
    field: Type,
    field: Type,
    field: Type];
```

```
Record: TYPE = RECORD
    [
    field: Type,
    field: Type
    ];
```

In both cases, the structure is clear; it is indicated by the indenting, not the placement of the brackets. We prefer the form on the left, because it requires less vertical space.

The rule illustrated above is not applied absolutely to all bracketing pairs, however; the placement of a bracket depends not only on the bracket itself, but also on its prefix and the clauses which follow (its suffix). For example, a loop statement has the following possible prefixes, brackets, and suffixes:

Prefixes	Brackets	Suffixes
FOR, WHILE	DO	OPEN
UNTIL, (empty)	ENDLOOP	ENABLE

Obviously, there are several special cases which must be examined. The sections below contain a number of examples. They observe the following rules for the placement of opening and closing brackets:

- The opening brackets [, FROM, and DO appear on the same line as their prefixes; on the other hand, BEGIN starts on a new line.

  If the remainder of the statement fits on a single line (with its closing bracket), it is placed there, indented one level. Otherwise, all closing brackets except] appear on lines by themselves.

These rules apply to all statements which do not fit on a single line. In general, each of the rules for placing statements on a line should have the phrase "if it will fit" attached.

Basic Statements

The basic statements are indented according to the rules in the *Mesa Language Manual.* These are repeated below, following the examples.

```
IF bool THEN                        FOR i IN [0..n) DO
    BEGIN                               body
    body                                ENDLOOP
    END
ELSE                                SELECT tag FROM
    BEGIN                               case => statement;
    body                                case => statement;
    END                                 ENDCASE
```

The statement following a THEN or ELSE is indented one level, unless it fits on the same line. THEN is on the same line as its matching IF, and ELSE is indented the same amount as IF. In the case that the statement following ELSE is another IF, both are written on the same line.

The labels of a SELECT (and its terminating ENDCASE) are indented one level, and the statements a second level, unless they fit on the same line with the label.

Each compound BEGIN-END, DO-ENDLOOP, or bracket pair is indented one level. When the rules for IF and SELECT call for indenting a statement, a BEGIN is not indented an extra level.

A body can appear on the same line with its brackets if the whole construction will fit. This leads to the following shorter forms:

```
IF bool THEN statement              FOR i IN [0..n) DO
ELSE BEGIN body END                     body ENDLOOP
```

The outer most Begin-End of a Module is not indented, however. More complex forms of these statements are considered in the following sections.

Statements with Opens and Enables

Some of the basic statements described above can have various optional clauses attached. In particular, BEGIN and DO statements may have OPEN and ENABLE clauses; they are formatted as follows:

```
FOR i IN [0..n) DO
    ENABLE
        BEGIN
        label => action;
        label =>
            BEGIN
            body
            END;
        END;
    body
    ENDLOOP
```

```
FOR i IN [0..n) DO
    OPEN def1, def2;
    ENABLE label => action;
    body
    ENDLOOP

BEGIN OPEN def;
ENABLE
    label => action;
body
END
```

A compound ENABLE clause is indented the same as a SELECT statement, with the action on the same line as the label if it will fit. If there is but one clause, it may appear on the same line as the ENABLE.

Note the two statements are handled slightly differently. In BEGIN statements, the OPEN or ENABLE keyword goes on the line with the BEGIN, whereas a DO always goes with its prefix. If there is no prefix, DO is on a line by itself and indented as though there were a prefix. If a Begin-End block exists without a prefix, it is not indented. The formatter will bracket such a block with blank lines.

Statements with Repeats and Exits

The REPEAT and EXITS keywords always begin a new line at the current indenting level. The exit clauses which follow are indented one level, and each appears on a separate line.

```
FOR i IN [0..n) DO
    body
    REPEAT
        label => action;
        label => action;
    ENDLOOP
```

```
BEGIN
body
EXITS
    label => action;
    label => action;
END
```

If an action does not fit on the line with its label, it is indented as in a select statement. If there is a single exit clause, it may appear on the same line as REPEAT or EXITS.

```
FOR i IN [0..n) DO
    body
    REPEAT
        label =>
            BEGIN
            body
            END
        label => action;
    ENDLOOP
```

```
FOR i IN [0..n) DO
    body
    REPEAT label => action;
    ENDLOOP

BEGIN
body
EXITS label => action;
END
```

Brackets without Prefixes

The example on the right illustrates a problem discussed above with the form of BEGIN which starts a new statement: because this form has no prefix (and because in general we indent BEGIN-END blocks), its scope would appear merged with the previous statement if it is also indented. In the example, the Begin-End block is not indented.

Program Structure

At the outermost level, the BEGIN-END pairs which surround program definitions are not indented although those surrounding procedure definitions are idented just as they are in simpler constructions. This gives the program the following format:

```
DIRECTORY
    OneDefs: FROM "onedefs",
    TwoDefs: FROM "twodefs";

DEFINITIONS FROM OneDefs;

Prog: PROGRAM [parm: Type] =
BEGIN

Proc1: PROCEDURE [parm: Type] =
    BEGIN
    body
    RETURN
    END;

Proc2: PROCEDURE . . .
    BEGIN
    . . .
    END;

main body
END
```

The procedures are separated by one blank line (or equivalent leading); the main body of the module is preceded by two blank lines. If declarations inside the procedures are separated from the code by a blank line, then two blank lines separate procedures, and three blank lines precede the main body.

Comments which begin a line are indented to the same level as the statement following them. Line spacing before and after such comments is collapsed to at most one blank line. Comments which appear to the right of statements are preceded by two spaces.

Breakage

We have not yet addressed all of the issues involved in breaking long lines (or combining short ones). The most common cases occur in procedure headings, record constructors, and catch phrases. They are handled as follows:

```
Proc: PROCEDURE [
    fieldName: TypeName, fieldName: TypeName, fieldName: TypeName]
    RETURNS [fieldName: TypeName, fieldName: TypeName] =
    BEGIN
    -- body of the procedure
```

```
    END;

pointer: POINTER TO . . . ;
pointer↑ ← Constructor[
    expression, expression, expression, expression];
next statement;

pointer↑ ← Proc[arg, arg !
    signal 1 => statement;
    signal 2 =>
        BEGIN
        body
        END];
next statement;
```

Long procedure headings are handled by first placing the RETURNS clause on a new line. Next, the parameters are placed on a separate line. Finally, the parameter or result lists are split into several lines. Constructors are handled in a similar manner, with the field lists indented one level. Catch phrases are indented just as SELECT statements, as if the procedure call had been the SELECT exp FROM prefix.

2.2.3 Control Conventions

Readers who are not interested in the control codes (included in the stored file) can skip this section and go to 2.2.4.

Mesa Source Formatter Output

The purpose of this section is to describe in detail the output format of the Mesa Source Formatter. The above discussed formatting in terms of the source's visual appearance, since that is how it is most commonly considered. The output of the Formatter, however, is of course a file, not paper and toner. The output format of the formatter is designed to be appropriate for subsequent input to a) compilers, cross-referencers, and other tools which view the file as Mesa source, b) program editors, which view the file as machine-readable text, and c) hardcopy programs which can produce results that approximate the visual formatting discussed earlier.

Control Code Formatting

The basis for the output format of the formatter is a restricted subset of the control codes that are used by the current series of Bravo editors. We will first describe these control codes, both their syntax and their current semantics, and will then describe the Formatter output in terms of them.

The output of the Formatter is a *sequence of paragraphs.* A *paragraph* is a *sequence of characters* which satisfies the following properties:

1. It ends with a carriage return (ascii octal 15).

2. It contain exactly one controlZ (ascii octal 32).

3. The character sequence beginning with the controlZ and continuing up to the terminating carraige return are the *paragraph's looks.* The looks are defined as follows:

a. They consist of a sequence of margin looks, followed by a slash ("/"), followed by a sequence of run looks.

b. A margin look is either the letter "l" [that's an *ell*] followed by a decimal integer, or the letter "d" followed by a decimal integer.

c. A run look is:
-- a non-nil sequence of distinct letters from the set "B" "b" "I" "i" [those are *eyes*] followed by a decimal integer, or
-- an "f0" or an "f1" followed by the above, or
-- an "f0 " or "f1 " followed by a decimal integer.

4. The sequence of characters in the paragraph prior to the controlZ are the *body* of the paragraph. The body may contain any characters other than a controlZ, and it is the set of paragraph bodies within the source program that are compiled by the mesa compiler.

In terms of the current Bravo editors, the meanings of the paragraph looks are as follows:

The margin looks define the left margin ("l") and the first line in the paragraph's left margin ("d").

The run looks define the formatting to be applied to a sequence of characters within the body of the paragraph. As the run looks are scanned from the slash to the carriage return, an occurrence of a "B" sets a "bold" flip-flop off; a "b" sets it on; an "I" sets the "italic" flip-flop off, an "i" sets it on; a "f0" sets the "font" value to zero, an "f1" sets it to one. As the scan begins, the cells are set to off, off, and zero. When a decimal integer is encountered in the run looks, that many characters from the body are given the characteristics defined by the current "bold" "italic" and "font" values. When the run looks are exausted, any remaining characters in the body get the final characteristic values.

Example: the paragraph

IF foo<2 THEN ERROR;[controlZ]l3200d3000/f1 3f0 6f1 5b5f0B[return]

defines a line to be formatted as

IF foo<2 THEN ERROR;

All Bravo-like control codes which appear within the input source file are completely ignored by the Formatter. A set of paragraph looks, as described above, is skipped over whenever encountered. (In other words, whenever a controlZ is encountered within the input stream, all subsequent characters up to, but not including, the following return are ignored.) Any controlL characters within the input are ignored.

Output Formating

As it creates its "formatted" output file from the supplied source input, the Formatter uses the above control codes in the following ways:

1. Whenever it puts a new Mesa statement on a new "line", it either begins a new paragraph or combines the statement with the previous paragraph. The only other time it puts returns within the body of a paragraph is to fold a single Mesa statement that is too long to easily fit on a line, as discussed in the section on "breakage".

2. Whenever it puts a single statement on multiple lines for the purpose of indenting, as in a record definition, it creates separate paragraphs.

3. All indenting is done via the margin looks; the "l" and "d" margin looks are specified for every paragraph.

4. The basic (level zero) margins are l3000 d3000. This provides for no additional indentation on line folding and the numeric values are appropriate to current usage.

5. When a paragraph is indented, 635 is added to both the "l" and "d" margins. [635 is a reasonablle number of micas for indenting and is half the default Bravo 'nest' adjustment.]

6. All Mesa keywords are made font one; everything else is made font zero. Except that, in the case of consecutive Mesa keywords, the entire sequence, including imbedded spaces, is made font one.

7. All comments are made italic; everything else is made non-italic.

8. The names of procedure constants are made bold where they are defined; everything else is made non-bold.

Hardcopy

One result of the above output formatting rules is that the Formatter output will hardcopy correctly (in the sense of agreeing with the conventions discussed above) when the output is printed via the normal "hardcopy" command of Bravo. Bravo will obtain the fonts that correspond to "zero" and "one" from the invoker's "user.cm". It will also obtain the right margin and paragraph and line leading information from "user.cm".

2.2.4 Naming Conventions

The objective of naming conventions is to provide a standard means of generating and interpreting names in the code. The general conventions are:

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

o **Compound names:** Each component after the first is capitalized (e.g., maxFilePage) in compound names. Components may be nouns, verbs, prepositions, adverbs, and so forth.

o **Procedures that modify the state of an object:** Procedures that modify the state of data structures that are part of an abstract object such that subsequent calls on the object/interface give different results, should be named with primary verb and noun components that clearly express the change to the object. For example, consider the following:

```
word: RECORD [name: STRING, meaning: INTEGER]

dictionary.DeleteWord[word]   -- not, dictionary.IsEmpty[word]
dictionary.AddWord[word]   -- not, dictionary.Has[word]
dictionary.SetMeaning[word]   -- not, dictionary.Meaning[word]
```

Note that this convention is not meant to exclude names such as DictionarySetMeaning or SetMeaningTo.

o **Abbreviations:** If used as initial name components, 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 project's special conventions dictionary. No standard suffix abbreviations exist at this writing.

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.

2.2.5 Coding Conventions

Except for the restrictions contained in the language manuals which reflect language constraints and the material that follows, 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 Plauger). This and other acceptable reference(s) of programming wisdom will be made available to our programmers.

In general, conventions for the use of signals, procedure variables, records, ports, processes and other Mesa features will be established on a project basis and will be documented as special conventions by the project. However, we do suggest the following for your consideration:

o Precede an invocation of a signal or error by the word SIGNAL or ERROR. The Mesa compiler does not require this, but your fellow programmers may. Signals behave very differently from procedure calls and the reader of your code should have no doubts about which you are using.

o Use signals with some care. Unless there is good reason to resume a signal (to save redoing an extensive computation) use an Error instead. As part of an interface specification, clearly indicate what Signals/Errors can result. If you have data that is "global" to an object in an inconsistent state when you call procedures that can result in signals that your caller might terminate, you should enable the UNWIND signal so you can clean up.

o Use a SELECT TRUE rather than a string of ELSE IFs for readability.

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

o Use ennumerated types rather than individually declared INTEGERS to represent the possible states of an object or control parameter. This enhances readability and ensures that you are using the correct values for the desired state. This also minimizes the storage required for the state information.

o Use discretion in the application of the PUBLIC and PRIVATE qualifiers in declarations. Declare items PUBLIC only if clients need to get at them. Use PRIVATE to hide variables and functions that your caller need not and should not know about. Recall that PUBIC is the default for Definition modules and PRIVATE is the default for Program and Data modules.

o Brackets should always be used when calling procedures with no parameters.

o Don't use ↑ with the . or array indexing opporators. Let the compiler do the dereferencing for you.