DESIGN NOTEBOOK - SECTION V

SUBJ: RUNCOM - A Macro-Procedure Processor for the 636 System

FROM: Louis Pouzin

DATE: April 7, 1965

* * * * * *

CONTENTS

		Page
1. -	Principles	M-1
2. -	Requirements	M-2
3. -	Composition of a Macro-Procedure	M-2
4. -	The Meta-Language	M-3
5. -	Making of Macro-Procedures	M-13
6. -	Calling a Macro-Procedure	M-13
7. -	Expansion Mechanism	M-16
8. -	Optional Meta-Language	M-18

1. - Principles

1.1 - It is suggested to refer to the CC-Memo 238 in order to get acquainted with the present macro-command machinery built for the 7094 CTSS system. Indeed, most of the basic ideas are carried through this paper as preliminary assumptions.

1.2 - Another paper describing the SHELL (636 System Design Notebook Section IV) explains how the most general procedure may be executed simply by specifying its name, and the list of its arguments. Inputting such a request is conveniently done from a console as a message directed to the supervisor, or the requests may be read from an external device, or from a file kept onto a secondary storage, like disks. Reader's knowledge of the SHELL description is assumed in this paper.

1.3 - Chaining requests is a normal function of the SHELL, but if there is no limit as to the number of successive procedures executed in a row, requirements are that all the actual arguments be specified somehow at the time of execution. In most cases, some arguments are rarely modified, whereas some others are usually changed for every execution. Furthermore, many jobs may be described in terms of smaller tasks, the sequence of which is almost invariably determined, barring a few options or error conditions. For these reasons it is very tempting to describe some procedures as macro-procedures, with a skeleton of fixed parameters, and a set of substitutable arguments to be specified at execution time. The macro-procedure is then given a name, stored permanently, and invoked as if it were a single normal procedure.

A similar technique is now a mandatory feature of any serious assembly language, and its utility is no longer a point of discussion, but rather of implementation.

1.4 - A macro-procedure machinery would then consist of the following:

- a Meta-language intended to describe how a sequence of procedures is to be constructed for execution.
- a Macro-processor using the meta-language syntax in order to build up the actual list of procedures to be executed, with their actual arguments.

- a procedure initiating and controlling the execution of the sequence so made up. This is the SHELL, outlined elsewhere.

2. - Requirements

2.1 - By assumption, the description of a macro-procedure (say the prototype) is fairly simple, and does not carry any sophisticated logic. If it did, the user had better use some of the powerful programming languages, which the ACM journal is crowded with. We mean that the kind of logic that we may expect to describe in a macro-processor language is voluntarily restricted to elementary options.

2.2 - For the same reason, and as a compensation, the macro-procedure should not require any debugging. Writing should be as straightforward as the making up of a deck for, say, an FMS job.

2.3 - In executing the sequence of procedures, there should not be any restriction arising from the use of the macro-processor.

2.4 - Let aside some possible naming conventions, a macro-procedure must provide the same facilities as a regular procedure. For example, there should not be any restrictions in calling a macro-procedure from inside another macro-procedure.

3. - Composition of a Macro-Procedure

3.1 - A macro-procedure is a sequence of procedures described in a prototype. The prototype is stored in a file with class name BCD.

3.2 - The name of the macro-procedure is the primary name of the BCD file, or is mentioned in the prototype itself.

3.3 - A prototype contains several classes of information.

3.3.1 - Identification of formal (or substitutable) arguments

3.3.2 - Names of (macro) procedures to be gathered into a list for execution, along with their arguments.

3.3.3 - Names of (macro) procedures to be executed immediately, when encountered by the macro-processor. Also are specified their formal or actual arguments.

3.3.4 - Editing control words

3.3.5 - Expansion control words

3.3.6 - Comments

3.3.7 - End markers

4. - The Meta-Language

4.1 - In order to recognize the formal arguments of a macro-prototype, they are all mentioned, as spelled when used, in a header, which can take two forms:

E.g.

	CHAIN	ONION	POTATO	GARLIC	etc...	
or	MACRO	RECIPE	ONION	POTATO	GARLIC	etc...

Either word CHAIN or MACRO must be used as spelled. When CHAIN is used, the name of the procedure is the name of the file containing the prototype. This allows changing the name of the macro simply by renaming the file.

When MACRO is used, the name of the macro is the word following immediately MACRO, (here RECIPE). This allows searching files for a specific macro.

All other words, following CHAIN or the macro name, are formal arguments. This means that wherever POTATO e.g. is mentioned in the prototype, (exception will be clearly stated) it will be replaced by whatever is specified at the same rank in the macro-call. E.g. if one calls RECIPE ONION IDAHO etc... ONION will not change, since the actual value is identical to the formal spelling, and POTATO will be replaced by IDAHO.

There may be any number of formal arguments.

In the macro prototype, CHAIN or MACRO must appear before any executable procedure, and before any control word which controls the expansion of an executable procedure. This statement will become clearer in the following

Only one CHAIN or MACRO can be used in a macro prototype, but neither one is necessary if there are no formal arguments, and if the macro-name is the same as the file name. There is no argument substitution in a CHAIN or MACRO heading.

4.2 - The prototype contains any number of procedure requests, consisting of the procedure name followed by its arguments. A request is contained in a logical BCD record, whatever it is. But it will be likely a string of BCD characters ending with a carriage return, (or end of record mark). We assume for simplicity that the usual SCANNER will be used, (see SHELL description) so that each procedure request will be interpreted as a list of BCD strings, (words) separated by one or several blank characters, or tabs. Nevertheless, this point is already treated in the SHELL description. Suffice it to say that every procedure request is interpreted according to the delimiters conventions of the system, possibly modified by private user's settings.

A request may contain any number of arguments, such as:

MAIL GEORGE MOLLY BOB ARTHUR etc...

Usually the first word is the name of the procedure to be executed, and all other words are arguments to the procedure. But, generally speaking, any of the arguments may be a meta-argument, or a special type (# convention) as specified in the SHELL paper (paragraph 5.10 and 8). In other words the meaning of the whole request is not taken care of by RUNCOM, but later on, by the SHELL at execution time. In particular, when a request starts or ends with the meta-argument (MORE), RUNCOM does not attempt to stick together pieces of requests with their continuation counterparts. As far as RUNCOM is concerned, every logical BCD record is processed as a separate request. Any process based on syntax or semantics is left for the execution phase, to the SHELL and other procedures called.

Any word of a request, regardless of whether it is a special, meta, normal argument, or the name of a procedure, is always substitutable, if it matches one of the formal arguments specified with CHAIN or MACRO. (See inhibition of this rule in 4.4.1).

4.3 - Inside the prototype, execution of requests may be controlled by the single control words (NOW) and (LATER)

E.g.

```
...
(NOW)
EDIT    ALFA    GAP
PRINT   ALFA    GAP
GAP     ALFA    (NOLIST)
(LATER)
...
```

When RUNCOM reads the prototype, the occurence of the control word (NOW) calls for a different expansion mode. Successive requests are still scanned for substitution of arguments, but, instead of being saved into a list, the SHELL is called for immediate execution for every request. Furthermore, on return from the SHELL, the actual values of the formal arguments are updated with the values given by the executed procedure. Thus it is very easy, while expanding a macro, to perform any argument modification, simply by calling an appropriate procedure.

E.g.

```
...
(NOW)
IFEQUALL    A    B    PRØC
(LATER)
PRØC
```

A, B, PRØC are formal arguments, and IFEQUAL is a procedure which gives to PROC the value (NIL) if A is different from B; otherwise PROC is not altered. Runcom will substitute for PROC whatever value has been given in the macro-call, hut this value may have been turned into (NIL) by IFEQUAL. Thus when RUNCOM will transmit to the SHELL the list of actual procedures, PROC will be either executed for its actual value, or skipped by the SHELL if it has been (NIL)ed.

It is important to notice that all logic pertaining to the checking of arguments, whether actual or formal values, is rejected from RUNCOM upon external procedures, which may in turn be as sophisticated as desirable. One does away through that technique with the always unsatisfactory design of all conditional IF pseudo-operations found in assembly languages.

The mode of immediate execution stops as soon as the control word (LATER) is encountered.

Successive (NOW)'s are redundant, and have no more action; so are successive (LATER)'s. In other words, (NOW) sets a switch, and (LATER) resets it; nesting has no meaning.

(NOW) and (LATER) are recognized after argument substitution, i.e. they may be substituted as actual values.

4.4 - Some editing options are controlled by the current permanent options of the user, and will be mentioned below in paragraph 7.8. In addition, the following is performed by RUNCOM according to the text of the prototype.

4.4.1 - No argument substitution will be performed in a request when the first character is - (minus sign). Execution is carried through with the literal values as specified in the prototype.

4.4.2 - A meta-argument ' (apostrophe) permits concatenation of the previous and the following arguments, after substitution.

E.g. M ' BER 196 ' D

will come out ØCTOBER 1965 if M and D are formal arguments set to OCTO and 5.

There may be several successive concatenation yielding a single argument.

Substitution suppressor and concatentation are allowed in the mode of execution (NOW).

Apostrophe may be substituted as actual value for a formal argument, and yet means concatenation after substitution wherever there is such a value as argument.

4.5 - Several features control the expansion of the macro-prototype so as to allow a large flexibility in constructing the final list of procedures to be executed.

4.5.1 - (SKIP) and (STOP) are two control words which allow by-passing a part of the prototype. Either one must be a single word request.

E.g.
```
...
(SKIP)
(GAP      ALFA
(STØP)
...
```

When processing the prototype, the occurence of (SKIP) inhibits both the immediate execution of procedures, and the expansion of the list for later execution. However, argument substitution and other editing functions keep activated. The occurence of (STØP) resumes the normal mode. Thus, one can ignore, as far as procedure execution is concerned, an arbitrary part of the prorotype. Since argument substitution is performed, both (SKIP) and (STOP) may be substituted as actual values for arbitrary formal arguments. (SKIP) and (STOP) correspond to a single switch. Consequently nesting has no meaning, and redundant occurences are ignored.

E.g.
```
...
TEST1
sequence1
TEST2
sequence2
(STOP)
...
```

One may execute sequence1 + sequence2 if TEST1 and TEST2 are (NIL) or (STOP) or (LATER).
One may execute sequence1 only, if TEST2 is changed to (SKIP)
One may execute sequence2 only, if TEST1 is (SKIP) and TEST2 is (STOP)
One may execute nothing, if TEST1 is (SKIP) and TEST2 is (NIL)
Setting of TEST1 and TEST2 may be done by substitution, or by external procedures for which an immediate execution has been requested before in the prorotype.

4.5.2 - Labels may be assigned in order to refer symbolically to various steps in the prototype. This is done through the control word (LABEL).

E.g.
```
...
- (LABEL)    HERE
...
```

The BCD string HERE is then associated with the corresponding position in the prototype of the request (LABEL) HERE.

Neither (LABEL) nor the specified value may be substituted as actual arguments, and there must be a minus sign heading the request. Otherwise there would not be any label assignment, and the whole request would be processed as an ordinary one. (LABEL) may not appear before CHAIN or MACRO.

4.5.3 - By using (GOTO) one may force an unconditional transfer in the prototype.

E.g.
```
MACRO   DOSEGMENT   ALFA   WHERE
(NOW)
- (LABEL)  AGAIN
EDIT    ALFA        GAP
GAP     ALFA
IFIL    ALFA    PROCEDURE   WHERE   OK   AGAIN
(GOTO)  WHERE
- (LABEL)  OK
(LATER)
PRINT   ALFA    GAP
```

In the previous example IFIL is a procedure which sets the value of WHERE to be OK if the file ALFA PROCEDURE exists, and to AGAIN if the file does not. All these values are arguments to IFIL. A part of the prototype is specified as executable immediately, and depending on the value assigned to WHERE, RUNCOM keeps processing the following of the prototype or goes back to the EDIT request.

Since both (GOTO) and the specified label may be substituted as actual values of formal arguments, one may use this inconditional transfer as an optional transfer to an arbitrary request. (GOTO) with no label, or with label (NIL) has no action.

(GOTO) may not appear before CHAIN or MACRO.

4.5.4 - Although it would be possible to control iterations with (GOTO)'s it seems desirable to have a more specific mechanism for the repetition of the same sequence with different sets of arguments.

E.g.

```
MACRO    ASSEMBLE    FIL     CLASS
(LOOP)   FIL         CLASS
LISTF    FIL         CLASS
CLASS    FIL
(LOOP)
```

The sequence of requests bracketed by the (LOOP)'s will be repeated several times if the formal arguments FIL and CLASS are substituted with lists of values, instead of single values.

For example, the macro may be called by:

```
RUNCOM   ASSEMBLY   ( ONION TOMATO GARLIC )   ((GAP ALGOL )
```

The expansion of the macro would come out:

```
LISTF    ONION     GAP
GAP      ONION
LISTF    TOMATO    ALGOL
ALGOL    TOMATO
LISTF    GARLIC    ALGOL
ALGOL    GARLIC
```

One can understand the simple logic by the previous example. Each of the formal arguments specified with the first occurence of (LOOP) is replaced by an actual value taken from the list associated with each argument. The number of iterations is equal to the number of values specified in the longest list. When a list is exhausted, the corresponding formal argument holds its last current value.

There may not be dynamically nested (LOOP)'s in the same prototype, but there may be as many disjointed (LOOP)'s as desired. On the other hand, any of the procedures included in the scope of a (LOOP) may be a RUNCOM containing iterations, etc..., at any depth.

(LOOP) may be substituted as an actual value for a formal argument. It may not appear before MACRO or CHAIN.

4.5.5 - Another feature turns out to be very useful when the successive sets of arguments for a loop are too numerous. It also permits an arrangement of arguments which may be closer to the way users have them in mind.

E.g. Assuming that a user wants to repeat part of a process with a series of files, say moving them to another file directory, he might write a macro as:

```
...
(LOOP)    A    B
UPDATE    3    A    B
(LOOP)
...
```

and call the macro with two lists as:

```
RUNCOM ... ( ONION PEAS SALT LEMON )  ( GAP GAP GAP ALGOL )
```

Each item, ONION GAP, PEAS GAP, etc..., is scattered through each list, and the visual association of which goes with what may be cumbersome, when there are many arguments and several lists requiring several lines to fit.

Therefore, another form of feeding the loop is available: Successive items are stored into a file, with class name BCD.

E.g.
```
ONION    GAP
PEAS
SALT
LEMON    ALGOL
...
```

Each item goes in a single logical BCD record, (line or whatever it is). Instead of (LOOP) the prototype contains:

```
(FILE)    NAME    A    B    C    ...
```

where NAME BCD is the name of the file containing the successive items for the loop. A, B, C, etc... are the formal arguments to be substituted with actual values into the loop.

The following rules apply for the substitution.

- A first loop is driven with the actual values specified in the call to RUNCOM. If all arguments are (NIL) or missing the next step is directly started.
- If the file NAME BCD exists, and is not void, a logical record is read, and each formal argument, as specified with the (FILE) control word, is replaced by the corresponding actual value read from the file NAME BCD. Explicitly (NIL) values are substituted as such, but missing values are not substituted, and the corresponding formal argument holds its current value. Hence, one may specify them in a hierarchical manner so that semi-constant values are rightmost in the file, and need be mentioned only when they change.
- When the file is exhausted, or if it does not exist, looping is terminated.

The next occurence of(FILE) without argument causes RUNCOM to jump back to the previous (FILE) in the prototype, and to proceed with a new set of values.

Nested (FILE)'s are not allowed.

Both (FILE) and the name of the file may be formal arguments and substituted with actual values.

N.B. It seems to the author that (FILE) and (LOOP) loops could be nested at one level, dynamically speaking, but not all implications have been examined. Therefore, this point is to be settled at a later step of implementation.

4.5.6 - Any request with the first character * or $ is a comment. Comments headed by * are completely ignored and serve only as remarks that the user wants to associate with his prototype.

Comments starting with $ are printed on the user's console, (or in the MESPØT file) at the point of execution where they are encountered.

As a matter of fact, RUNCØM replaces $ by the name of a procedure, say CØMMENT, which prints the concatenation of the BCD strings given as arguments.

Neither * nor $ are substitutable, but words of the text going with $ are substitutable.

One may use the special concatenation $- to mean that no substitution be performed in this comment.

* Comments may appear anywhere in the protype, but $ comments may not appear before CHAIN or MACRØ.

Blank requests are always ignored, and may appear anywhere.

4.5.7 - The control word (END), when encountered, forces the end of the RUNCØM processing. The list of requests created by the expansion is closed, and the SHELL is called for execution. Thus (END) need not be the last request in the prototype, although it is a natural place to put it.

(END) may be substituted as actual values of a formal parameter. It may appear anywhere in the prototype, including before CHAIN or MACRØ.

The text of the prototype need be bounded in order not to run away when reading it. Either an end of file or a single word request ENDMACRO may be used.

ENDMACRØ will never be ignored, even in a (SKIP) phase, and it may not be substituted.

5 - Making of macro-procedures

5.1 - As we have said, the prototype is contained in a regular BCD file. Consequently it can be created by any of the common procedures designed for handling BCD files. A private program can do the same. Furthermore, all input media are suitable, either by direct typing from a console, or by card reader, punched tape, or any arbitrary external device.

6 - Calling a macro-procedure

6.1 - A prototype is not an executable program in terms of machine instructions. Therefore one cannot transfer control to a prototype. Instead, the procedure RUNCØM is systematically invoked as following:

RUNCØM macro-name arguments . . .

Same conventions as for any procedure apply to a call upon RUNCØM; in particular with respect to the arguments delimiters.

6.2 - The first argument of RUNCØM is the name of the macro-procedure. It is usually the name of a BCD file containing the prototype. This file, if it exists, will be read by RUNCØM, and processed according to the following rules:

6.2.1 - Some executable request encountered, but no CHAIN and no MACRØ. Expansion is performed, and must hit an (END) control word. or the end of the file. Occurences of CHAIN or MACRØ creates an error and stops the process.

6.2.2 - CHAIN is encountered. Expansion is performed under same conditions as in 6.2.1.

6.2.3 - MACRØ is encountered. If the name following MACRØ does not match the one following RUNCØM, reading the file continues until another CHAIN or MACRO is encountered, and the process is recycled. If the names match, expansion is performed under the same conditions as in 6.2.1.

6.2.4 - (If the end of file is encountered before any expansion got started, or if the BCD file does not exist), and if no library is specified (see below 6.3), an appropriate comment is printed, and RUNCØM returns to its calling procedure, via the error return meaning: need more arguments.

6.3 - Libraries of macro may be specified to be searched for the requested prototype. Should a condition mentioned in 6.2.4 occur, RUNCØM would then carry out the searching.

6.3.1 - The ØPTIONS segment (see SHELL description may contain an entry MACLIB, which points to a list of pointers to file names. Thus, successive files may be searched in the order they appear in the user's ØPTIONS segment.

6.3.2 - The argument list to RUNCØM may specify some libraries to be searched before those specified in the permanent options. The meta-argument (MACLIB) followed by the name of a BCD file of macros, is recognized by RUNCØM, and both are stripped off the list of arguments to be fed to the macro. Such a pair may occur several times in the argument list.

6.4 - As we have already shown in 4.5.4, a single formal argument can be replaced by a list in a (LØØP) scope. If no (LØØP) is specified for the particular argument, it is still possible to substitute a list where a single value is expected .

E.g. MACRØ SPLIT A B C
SPLIT A B C

called by RUNCØM SPLIT ALFA GAP (ØNE TWØ THREE) will expand as: SPLIT ALFA GAP ØNE TWØ THREE

In other words the contents of a pair of parentheses is substituted as is to the formal argument. It may contain any BCD string, including inner sets of nested parentheses. The outer set of parentheses is removed during the substitution. Meta-arguments to RUNCØM, like (MACLIB), will not be interpreted if embedded in parentheses, nor will the macro-name. The following example is not legal:

RUNCØM (MAC1 (MACLIB) MACRØS ARG1) ARG2

The result would be: MACRØ (NØT FØUND.

Parentheses used as delimiters for a group of arguments must be written as separate characters, (by blanks or tabs).

6.5 - It may happen that a macro procedure be called with less actual arguments than specified in the list of formal arguments. In this case, all formal arguments for which no substitution is provided are given explicity the value (NIL).

Needless to say, there may be automatic substitution of preset BCD values for those arguments missing, but this has to be done through external procedures.

E. g. let us assume that a procedure IFEQUAL A B C D E works as follows

.IF. A .EQUAL. B .THEN. C=D .ELSE. C=E

we may build a macro such as:

```
MACRØ   CØMPRES   A  B  C  D
(NØW)
IFEQUAL  B (NIL)   B  BCD  B
IFEQUAL  C (NIL)   C  A  C
IFEQUAL  D (NIL)   D  B  D
(LATER)
SQUEEZE  A  B  C  D
```

one may call this macro as follows:

```
RUNCØM CØMPRES ALFA
   file ALFA BCD is compressed into ALFA BCD
RUNCØM CØMPRES ALFA GAP
   file ALFA GAP is compressed into ALFA GAP
```

RUNCØM CØMPRES ALFA GAP BETA

file ALFA GAP is compressed into BETA GAP

RUNCØM CØMPRES ALFA GAP BETA GAPSQZ

file ALFA GAP is compressed into BETA GAPSQZ

The above example shows clearly enough that all desirable gimmicks are possible, as long as one prepares appropriate procedures. E.g. automatic generation of symbols, checking for generated symbols, etc...

7 - Expansion mechanism

7.1 - Any reader familiar with macro-assembler may already have gathered how RUNCØM proceeds to expand a prototype. Nevertheless, some further information may help to understand the internal organization.

7.2 - First RUNCØM gets its argument list from its calling program, SHELL orother, and sets the names of the file(s) to be read in order to find out the prototype. If this step succeeds, the next step is entered.

7.3 - The prototype is read entirely until an end of file or an ENDMACRO is encountered. This allows building a table of labels associated to pointers into the prototype. The prototype and all necessary information are stored in the stack (for recursive properties). Some syntax checking is performed during this phase, like CHAIN and MACRØ procedures.

7.4 - Then RUNCØM starts creating requests out of the prototype, the list of formal arguments, and the list of actual arguments. It has to be understood that this phase is basically dynamic. Switches as (NØW)-(LATER), (SKIP)-(STØP) are set at the time they are encountered, from then and on. There is no attempt to assign a mode to a scope of requests bracketed by a switch; only their dynamic succession matters.

E. g. (NØW)
- (LABEL) HERE
PROC1
(LATER)
PROC2
...
(GØTØ) HERE

When expansion proceeds from the top down, PROC1 is executed in the (NØW) mode, but a subsequent transfer to HERE may expand PRØC 1 in the (LATER) mode in the following part of the prototype.

Similarly, any (LØØP) occurence sets a switch whereby the next encountered (LØØP) will jump back to the previous encountered (LØØP) for another repetition with successive arguments. Use of (GØTØ)'s may result in inserting in the loop some parts of the prototype which are not bracketted (syntactically speaking) by (LØØP)'s.

In brief, the expansion phase is primarily an interpretative process, not a compilation.

7.5 - ENDMACRØ or end of file occurence is replaced by an (END) control word, so that eventually the expansion stops by encountering an (END). The list of requests expanded by RUNCØM is stored in the stack, and returned to the calling program at the end of the process, by a RETURN GAP macro-instruction

RETURN (@expanded list)

7.6 - It should be noticed here that RUNCØM <u>does not start explicitly the list of requests</u> ready for execution. RUNCØM initiates only those requests for which execution is requested (NØW). For that, it calls the SHELL for every single request. Since the SHELL is recursive, it does not matter if it were already the RUNCØM's calling program.

When RUNCØM returns with the expanded list, the calling program is usually the SHELL, which will take care of the execution. But any other program may call upon RUNCØM, considered as merely a macro-expander, in order to get a ready-to-use list of requests, the execution of which may be postponed arbitrarily.

7.8 - RUNCØM, as any other procedure, is sensitive to the user's permanent options. In (LIST) mode, it will print a list of generated requests, either for (NØW) or (LATER). In (DEBUG) mode it will leave in the stack both the prototype and complementary information, such as label table, formal and actual arguments lists, etc... In (LØUD) mode, it will print messages bracketing the execution:

MACRØ XXX STARTED AT 1302.6
MACRØ XXX TERMINATED AT 1325.0

7.9 - When the user wants an immediate execution of the macro-procedure, he may well specify the mode (NØW) for the whole macro, even if everything could be executed the same way in the mode (LATER). Results may be identical, but the system overhead will be more important, since RUNCØM and its associated segments are to be included as parts of the user process through the end, instead of being released as soon as the list has been expanded.

8. - Optional Meta-Language

8.1 - RUNCØM, as the SHELL, and as will eventually many common procedures, use certain graphical conventions, as meta-arguments, concatenation or grouping characters, control words, etc... It is certainly desirable for documentation, teaching and general understanding purposes, to have a list of conventional graphics accepted by all users as system conventions. However, conventions may have to be changed, and some users may have unsolvable problems of compatibility with existing conventions. Therefore, it seems a necessity to provide a general way of anticipating the problem.

8.2 - All words used as conventional notations could be gathered into a common segment; evidently users could not modify it. Whenever a common procedure of the system needs to refer to a particular graphic, it would pick it up from the common segment through an entry name, which can be conveniently the same as its usual contents.

E. g. (END) (MANUAL) (LIST) ' * $ could be entry names to a segment called SYSGRAPHICS.

Users could provide their own segment to be used by a particular procedure, or systematically, as a permanent own language.

From the point of view of system maintenance, we would get rid of this tedious affair of combing through program listings for chasing wired-in symbolisms.