TO: C. T. Clingen
F. J. Corbató
R. Freiburghouse
J. W. Gintell
N. I. Morris
M. Padlipsky
R. Roach
J. H. Saltzer
T. H. Van Vleck
S. Webber

FROM: V. Voydock

DATE: October 27, 1971

SUBJECT: Coding and Design Standards

Enclosed is a revision of the three part standards document which was published as MCB-576. I would appreciate hearing any comments and suggestions you might have, in particular suggestions for new standards. In addition, please notify me if I have failed to mention something which is currently being enforced as a standard. There will be a meeting to discuss coding standards on Thursday, November 4 at 10:30 a.m. in the Honeywell Conference Room. Another issue which should be discussed at this meeting is whether we should have more strict rules about programming style. For example, should we require that every program contain a paragraph describing its calling sequence.

Should not pass pointers as arguments
if a simple word do.
Publish — MPM

Identification

Design and Coding Standards for Non-hardcore System Programs

V. L. Voydock

In order to insure that system programs are of a uniformly high quality in design, coding and user interface, the following standards have been established. These standards apply to all programs in the Limited Service, Standard Service, Maintenance Tools and Development Systems. It is impossible, of course, to exhaustively list everything one must do to produce well-designed programs. Designing a good program is still more an art than a science. Therefore, the project management reserves the right, in certain cases, to require further modification to a program even though it meets the standards below. Similarly, exceptions will, of course, exist only with the explicit approval of the project management.

Coding Standards

1. All system subroutines must be pure, so that a single copy may be shared by all users. The Multics PL/I and FORTRAN compilers produce only pure subroutines.

2. All system subroutines must be written in the PL/I language. Explicit permission of the project management is required to use any other language. To aid others in understanding a program, the program listing should be well commented. This includes explaining the meaning of important variables.

3. Only Standard Service System and Development System subroutines may be called.

4. The names of all system programs that are not commands or active functions must end with an underscore ("_"). The names of all temporary segments and all non-standard I/O streams and condition names used by system modules must also end in an underscore. This is to avoid naming conflicts with the user.

5. All variables used, including called subroutines, must be declared. This is done to increase program readability and reduce the confusion introduced by default or implicit declarations. For called subroutines, the parameter list must be fully declared, unless, of course, the subroutine accepts a variable number of arguments (e.g., a free format output subroutine). For readability, declarations should be collected together in a logical way (e.g., at the beginning of the subroutine or block for which they apply, or at the end) rather than being scattered throughout the program.

6. Special characters should be placed in the program directly. To lessen dependencies on the character code being used, the built in function unspec should not be used for this purpose. For example,

```
dcl nl char(1) initial ("
");
```

declares "nl" to be a one character string whose value is the new line character. The statement

```
unspec(nl) = "000001010"b
```

should not be used.

7. Use of external static variables which do not contain a "$" (e.g., "dcl x external static") is prohibited since this data type cannot be efficiently implemented in the current Multics environment. External references of the form "a$b" are allowed. If the programmer needs to have an external data base which is shared among many subroutines, he may either create a segment by an appropriate file system call and reference it using based structures or use the assembler to create a data segment by appropriate use of the "segdef" pseudo operation. The programmer wishing to do this should consult with a knowledgeable member of the Multics Development Group.

8. All variables should be of the automatic storage class unless there is a good reason for them to be internal static, i.e., they are static by nature. See also rule 9 below.

9. In PL/I programs to avoid having to initialize variables whose values are constant every time the subroutine containing them is entered, and to avoid having copies of these variables made for every user of the subroutine, one should use internal static and initialize the variables using the initial attribute. The PL/I compiler will allocate space for these variables in the text section of the subroutine being compiled and will initialize them. Since the text section is pure, one copy of these variables will be used by all users of the subroutine. Unfortunately, if a variable of this type is passed as a parameter to another subroutine, the compiler has no way of knowing whether or not that variable is to be

changed by that subroutine and it, therefore, puts the variable into the linkage section. Therefore, if one has a large number of "constant" variables that are also passed as parameters, one should put them in the text portion of an assembly language program and initialize them using the appropriate data generating pseudo operations and reference them using either based structures or the "a$b" notation. This will assure that only one copy of these variables is used by all users of the subroutine. The programmer wishing more clarification of this point should consult with a knowledgeable member of the Multics Development Group. This rather messy problem will disappear if and when the PL/I language is changed to allow parameters to be declared "read_only".

10. Use of the PL/I allocate and free statement should be cleared in advance with project management, since there often exists more efficient ways to accomplish the same task. Subroutines that do perform allocations (or call subroutines which do) must establish a cleanup handler to free the storage in the event processing is aborted.

11. The programmer should avoid writing PL/I functions with multiple entry points which return different data types unless there is a good reason to do so, since this generates extra code at each return statement.

Programming Style

1. The most common route through a program should be the most efficient. More exotic facilities which are inherently expensive should be separated from the simple facilities so that a casual user need not pay for the exotic every time he uses the simple.

2. System programs should, in general, use one of the two standard I/O streams, user_input and user_output. Only special I/O service programs should issue I/O attach or detach calls for these streams. Commands should not, in general, have an "offline output" option. The file_output command is provided for this purpose.

3. All programs that are not commands or active functions should return a status code indicating successful completion or occurrence of an unexpected event, unless they are programs for which errors are unrecoverable or extremely rare, e.g., console output subroutines. This type of program should make use of the Multics signalling facility to signal that one in a million error. In general, because of the higher overhead involved, programs should not make use of the Multics signalling facility for "routine" errors and status conditions.

4. In most cases, programs that are not commands or active functions should not print error messages, but should allow a higher level subroutine decide on the seriousness of errors and what to do about them. In general, it is wise to let the most qualified subroutine give the message. A good rule of thumb to determine the "most qualified" subroutine is to ask whether anything could be learned by reflecting the error to a higher level subroutine. If the answer is no, then we have found the most qualified subroutine.

5. All programs that are not commands or active functions should assume they are called with the correct number and type or arguments and should not make checks for this. This is to avoid continually paying the cost of argument checking in programs which call the subroutines correctly. This does mean that the programmer must be careful to call subroutines correctly.

6. System programs should be prepared to execute properly even if they did not complete execution during a previous invocation because of a "quit" or a fault.

7. System programs should never call a command if there is a subroutine which does almost the same thing. Commands are inherently more expensive since they are designed to interact directly with a human user.

8. System programs should not use a subroutine to do something which can be done reasonably easily in a few PL/I statements. The purpose of this rule is to avoid the proliferation of unnecessary system subroutines. The exceptions to this rule are input/output (see paragraph 1 under "Error Handling and I/O") and implicit conversion from character to numeric data types. The reason for the latter exception is that this type of conversion is inherently more expensive than calling a specialized subroutine.

9. Calls to subroutines which require descriptors should be minimized when this does not conflict with program readability or degrade the user interface. This is because of the higher overhead involved in setting up argument lists with descriptors. For example, one should try to minimize the number of ioa_ calls in a program. This should not be interpreted to mean that one should remove all error messages from his program or make their output so terse as to be unreadable. It simply means that if, subject to the constraints mentioned above, it is possible to use one ioa_ call rather than two then one should do so.

Data Base Management

Designing a program for a virtual memory environment requires a new outlook on program and data organization. Though the programmer is freed from the onerous task of allocating physical storage for his programs and data (storing intermediate results on secondary storage, overlaying parts of his programs with other parts to fit into core memory, etc.) he cannot ignore the issues of data management and program organization if he wants his program to be reasonably efficient. This is especially true for programs which manipulate large amounts of data. The attitude, "I have an infinite virtual memory. If I need more room I'll create another segment", may be all right for the casual user building a one-shot program but not for the systems programmer. A major aim of the programmer should be to minimize the working set of his programs, i.e., his programs should create as few segments as is practical, reuse the ones they do create and should avoid unnecessary moving of data. In Multics it generally pays to spend cpu time (within reason) to save space. This principle should not, of course, be taken to an extreme. It does not mean, for instance, that one should not use a hash table. It is true that a hash table takes up more space than an equivalent linear list but a program will take less page faults referencing the former than searching the latter. In this case, the actual working set of the former is smaller even though its potential working set is larger. In all cases, the programmmer must exercise his judgement as to the proper trade-off between working set size and cpu usage, always avoiding the temptation to allow his working set to expand to infinity.

In addition to this basic principle, the following guidelines apply.

1. System programs must leave system data bases in a consistent state, e.g., a program which changes the contents of a segment should reset the bit count of that segment when it is done with it. Programs should make any period of inconsistency as short as possible. They must also clean up after themselves, e.g., free storage should be released.

2. In order to assure consistent behavior, all conventional translators must use the subroutine ti_ to interface with the file system. It might not make sense for non-standard translators such as BASIC to use ti_. Exceptions of this sort should be cleared in advance with the project management.

3. System programs should initiate the segments they access by a null reference name and should subsequently access those segments via a pointer. In general, segments initiated by a module should be terminated by that module (see (4) below).

4. In general, the process directory should be used to hold temporary segments. Programs should clean up after themselves by either truncating

or deleting their temporary segments. If the temporary segment can be reused the next time the program is invoked, it should be truncated. Otherwise, it should be deleted. As stated above, the names of temporary segments must be intelligible and must end in an underscore ("_").

5. Any system program which creates new segments should put them into the user's current working directory unless the program explicitly makes provision for the user to provide a target directory. (The moveb and copy commands fall into this latter category.) The aim of this rule is to avoid "messing up" another directory, such as the directory from which a source segment was obtained.

6. System programs which create new segments must set access control lists according to the conventions enumerated below. If a segment is being replaced instead of being newly created, the command must leave the access control list as it was before the command acted. For instance, a compiler finds that an object segment already exists with effective access RE for this user, with other access for other users. The compiler must obviously add W access to change the segment contents, but should restore the entire access control list to its former value when the compilation is completed. The file system interface subroutine ti_ does this automatically for the translator writer. The access to be given to the user creating a segment is:

Segment Type	Access	Ring Brackets
directory segment	SUMA	v, v, v
object segment	RE	v, v, v
data segment	RW	v, v, v

where V is the current validation level of the user.

Additional Standards for Commands and Subsystems

Through the mechanism of the command processor any program -- system subroutine, system command, user subroutine -- can be invoked from the console. System commands are a special class of subroutines that are explicitly programmed with the console user in mind. They must check carefully for argument validity; they must warn the user of possible misunderstandings; they must be very reliable. They must to the greatest possible extent be a self-consistent set, i.e., the behavior of a command should be predictable from that of other commands.

For these reasons a number of additional standards are necessary for system commands and subsystems.

Naming Conventions

1. For ease of typing, all commands must have an abbreviated name consisting of the first letter of the first two or three syllables or first two or three words of its name (e.g., rename rn, unlink ul, print_attach_table pat).

2. All command names and abbreviations must be cleared in advance with the project management.

Programming Style and User Interface

1. If a command would also be useful as a subroutine, break it apart into a command which interfaces with the user (processes multiple arguments, handles the star and equals convention, interprets control arguments, etc.) and a subroutine which does the work. This subroutine, like all subroutines, should return a status code rather than printing an error message. The outputing of error messages like all other user interface problems should be handled by the command.

2. Any command for which the star convention makes sense should use the star convention. Any command for which the equals convention makes sense should use the equals convention.

3. Characters which have special meanings to commands (e.g., "*", "=", ">", "<") should not be used in any context other than their standard one. For example, the "glorph" command should not interpret an argument of "*" as meaning that the user wants to be logged out.

4. Commands should not be "too powerful", that is, typing errors should not cause disastrous results. For example, with the old remove command, "remove a>b" would delete the segment b in directory a, whereas "remove a> b" (i.e., one accidentally typed a space in front of the "b") would remove the directory a. To remedy this, there are now two commands "delete", which deletes only non-directory branches and "deletedir" which deletes only directory branches.

5. Unless the purpose of a command is to produce some sort of output, it should produce no output during normal operation, i.e., it does not need to tell the user that it is doing its job. For example, if one enters the command "delete x y" the delete command produces output only if it has trouble deleting x or y. It does not type "deleting segment x", "deleting segment y". Commands which take a long time to execute (e.g., PL/I) should print a short message when they are entered to indicate they are functioning. The general idea here is to reassure the user that he has not done something wrong. After more than a couple of seconds wait, the user, particularly a novice user, begins to worry that perhaps the computer is waiting for him.

6. Commands which take the segment names as arguments should accept pathnames not reference names, unless they explicitly deal with reference names (e.g., terminate_refname). The user who has a reference name he wishes to pass to a command may use the "get_pathname" active function to convert this reference name to a pathname (e.g., "status [get_pathname x]" will cause the status command to be called with the pathname of the segment whose reference name is "x").

7. Commands which are really subsystems should be prepared to handle the "program_interrupt" condition which is signalled by the program_interrupt the current edm request is aborted and edm is ready to accept a new request from the console.

8. We come now to a standard that is difficult to express with any degree of exactness. The phrase "Commands should be designed with the

user in mind" expresses the spirit of the standard. What follows is a series of examples designed to sensitize the reader to some of the issues involved in designing a command. Calling sequences should be logical (e.g., the user should not have to remember that a "%" as a third argument to command "farfal" causes all segments with second component names "fred" to be deleted, whereas a "?" in the same position suppresses this feature). Commands should allow user intervention when appropriate. For example, the delete command should allow the user to decide whether a protected segment should be deleted, rather than forcing him to make the segment deleteable and re-submit the delete request (or worse, delete the segment without warning). Judicious use of red console output is encouraged. It should be used to call attention to important or unusual occurrences. Remember, over-use destroys the whole purpose of red output -- a command which outputs everything in red may as well output everything in black. "Canned" messages printed by commands should not contain characters which come out as escape characters on IBM model 1050 and model 2741 consoles and on model 37 teletypes (e.g., "ϕ<segment ϕ> not found" is not an acceptable message).

Argument Handling

1. Commands, whenever possible, must accept pathnames (not just entry names) as arguments. The subroutine expand_path_ should be called to convert a relative pathname into an absolute pathname.

2. Commands which deal with segments whose names have a fixed suffix should not force the user to type that suffix. Rather, they should append that suffix to their arguments if it is not given. For example, "pℓ1 x" and "pℓ1 x.pℓ1" should be equivalent.

3. Any command for which multiple arguments make sense should accept multiple arguments. This is especially true for commands that operate on single arguments (rather than pairs, triples, etc.)

4. All commands which accept a variable number of arguments should declare themselves as having no arguments (i.e., "command_name: proc;") and should obtain their arguments using the procedure cu_$arg_ptr.

5. Commands must obey Multics control argument conventions as described in the MSPM Section "Control Argument Conventions".

6. In general, for the convenience of the user, command arguments should be order independent unless the order dependency serves a useful purpose (as in the various options to dprint).

Error Handling and I/O

1. The input/output facilities of the PL/I language must not be used in system programs since they are inherently more expensive than system-provided subroutines.

2. To read a line from the input stream "user_input" use the subroutine (ios_$read_ptr". To read a line with appropriate data type conversion (i.e., the user is typing in pointers, floating point numbers, etc.) use the subroutine "read_list_".

3. Output lines fall into three distinct classes:

   a. "unusual status" messages

   b. questions

   c) everything else

Line of type a) should be outputted using the subroutines "com_err_" and active_fnc_err_ (active functions should use active_fnc_err_, all other modules should use com_err_), lines of type b) using the subroutine command_query_. These two subroutines are provided in order to centralize the processing of lines of type a) and b) so that changes in system conventions in this area may easily be made. For lines of type c) the subroutine "ioa_" should be used when it is necessary to format an output line; otherwise, use the subroutine "ios_$write_ptr".

4. Commands should check for status codes which have special meaning to them and either print appropriate error messages or, if the error is easily recoverable, allow for user intervention using command_query_. In all cases, messages must contain the name of the command which generated them since if this were not done, the user would have no way of knowing which command generated a given message if he had issued several at once or was running an exec_com segment. Complex programs such as compilers may output diagnostics by standard output subroutines but should have at least one call to com_err_ to notify the system that an error has occured.

Appendix to Coding Standards

This appendix contains some more explicit standards that are currently in effect. The standards below are less general and (possibly) more subject to change than those in the body of the document.

1. If the allocate and free statements are used, the allocation should be in "system_free_n", i.e., listen_$get_area and a based area should be used. Things allocated in system free_n should be freed as soon as they are no longer needed; cleanup handlers should be established before allocation to guarantee the freeing.
2. The second argument to com_err_ must be the full name of the command calling com_err_.
3. Commands that expect "yes" or "no" as answers to a question should use command_query_ with the "yes_or_no" switch set.
4. Currently check_star_ should be called if the star convention is used.
5. Efficient calls to an internal subroutine are generated if that subroutine is called only by its containing block and if it contains no adjustable automatic storage. The programmer, therefore, should use this construct for his internal subroutines whenever possible.
6. Implicit conversion from arithmetic to string types and vice versa is not currently allowed since it is inefficiently implemented.

7. The subroutines change_wdir_ and get_wdir_ should be used to change and get the working directory. Currently they merely call the appropriate file system subroutine, but if the modification to the search rules proposed by C. Clingen is adopted, they will need to do more.

8. Automatic label arrays should be declared label(...) so as to avoid unnecessary transfers to the pL1 unwinder. (see Multics PL/I Implementation Notes)

9. Nothing should be declared fixed dec or float dec by default unless it is used only with the addr function.

10. The table option should not be used when compiling segments for installation as it significantly increases the size of the object segment.