To: Distribution

From: Bill Silver, T. H. Van Vleck, Dave Vinograd

Date: March 12, 1975

Subject: Proposed Changes to the Syserr Mechanism

INTRODUCTION

This document describes proposed changes to the syserr mechanism. The reader is assumed to have a basic understanding of how the current syserr mechanism functions. Relevant information can be found in the following documents:

MTB-016
MTB-071
MTB-103

The most significant change involves the tabularization of all syserr messages. Calls to syserr will specify a code that identifies an entry in a table of syserr messages. This table will be used in a manner similar to the current error_table_. The table of syserr messages will contain all of the information needed to process, write, and log the syserr message. It will also contain information that can be used to sort and interpret syserr messages that have been saved in the syserr_log. This table will be structured in a way that minimizes the amount of wired main memory used. The tabularization of syserr messages will result in greatly increased administrative control over the use of the syserr mechanism. It will also provide a useful source of documentation for all syserr messages. A new source level language will be used to specify the syserr messages as a series of source statements. A new translator will be developed to process these source statements and produce the actual syserr table.

This document also describes two new capabilities that will be added to the syserr mechanism. One involves the passing of binary data to syserr. This binary data will be put into the syserr_log along with the syserr message. This binary data may then be formatted by programs that process messages from the syserr_log. A second capability involves passing an error_table_ code to syserr. The error_table_ message referenced by this error_table_ code will be appended to the syserr message.

The implementation of these changes involves adding three new entry points to syserr. In order to give the reader an overview of the new syserr capabilities the calling sequences of these new entry points are described below.

```
call syserr$message (syserr_table_code, arg1, ..., argn);

call syserr$binary  (syserr_table_code, data_ptr, data_size,
                     arg1, ..., argn);

call syserr$error_code (syserr_table_code, error_table_code,
                        arg1, ..., argn);
```

This document discusses these new syserr capabilities in detail. There are sections dealing with each of the following subjects:

1. Tabularization of syserr messages.
2. Binary data.
3. Error_table_ messages.
4. A new syserr table source language.
5. An implementation plan for these changes.

THE TABULARIZATION OF SYSERR MESSAGES

Problems with the Current Syserr Mechanism

The present calling sequence to syserr involves passing as arguments a syserr action code, a formline_ control string, and arguments used by formline_ to expand this control string. This calling sequence has major disadvantages in terms of storage usage and administrative control. These disadvantages are discussed below.

1. The calls to syserr generate too many words of object code. Many of the calls to syserr are made from wired programs and thus add to the total wired storage needed by the system. Even syserr calls in paged programs may add to system overhead. The presence of syserr calls in a program may result in adding an extra page to the program or splitting frequently used code over two pages.

2. The calling sequence to syserr is self contained. All of the information needed to process a syserr message is passed in the call to syserr. This may seem to be an advantage in that it allows syserr messages to be easily changed. This ease of modification is, however, a major disadvantage of the current syserr mechanism. Syserr messages are added, deleted, and modified so frequently, and so unnoticed, that it is virtually impossible to know what syserr calls are in the system at any given time. Only by making a pass over the source of the entire hardcore can we now generate a list of all the installed calls to syserr. Even doing this, however, gives us little information about the real meaning and purpose of each call. The syserr mechanism is a critical system function. It is used to crash the system, communicate with the operator, and log system information. From an administrative, marketing, and operations point of view it is unacceptable to have so little control over the use of this critical system function. It is unacceptable that we have no means of generating complete and up-to-date documentation about each syserr message.

3. Although the current syserr interface makes it easy to modify a call to syserr it makes it difficult to change the types of information passed to syserr. Any additional data that might be useful to have associated with a syserr message would have to be passed in the call itself. We would have to modify the syserr calling sequence, or add a new syserr entry point, or squeeze the additional information into the current calling sequence. MTB-071 described a cumbersome attempt to squeeze a sorting code into the action code argument. A better method of specifying a sorting code is presented in this document.

The Syserr Tables

The proposed new calling sequences to syserr are designed to solve the problems discussed above. They involve passing a syserr_table_ code as an argument to syserr. This code is much like an error_table_ code. It should be declared by the calling program as follows:

```
dcl   syserr_table_$nnnnnn  fixed bin(35)  external;
```

The entry name "nnnnnn" is a unique name that identifies this syserr message. For example, the syserr message generated by the iom_manager,

"iom_manager: bad devx X supplied."

might have the name, syserr_table_$bad_devx. Each call to syserr must pass the syserr_table_ code that corresponds to the desired syserr message. The same syserr_table_ code may be referenced by several different calls to syserr.

The syserr_table_ codes will correspond to entries in a source segment named syserr_table_.st. It is similar in function to the source segment, error_table_.et. A special compiler, syserr_table_compiler (stc), will be developed to process the syserr_table_.st source segment. The source language used to define a syserr message will be described in one of the later sections.

The syserr_table_.st source segment will be translated by syserr_table_compiler into two ALM source programs. These two ALM programs must then be assembled into object segments. The two object segments generated will be called syserr_table_ and syserr_info_. The reason that we need two segments will be explained below. An important point to note about these two segments is that both segments are generated from the single syserr_table_.st source segment. The syserr_table_ segment contains references to the syserr_info_ segment. Thus the installed version of both of these segments must have been generated from the same source segment. To ensure this we will require that the installed version of both of these segments be generated by the same invocation of syserr_table_compiler. In order to do this a unique ID will be placed in each of these two segments. At system initialization time, syserr_log_init will check that the variables syserr_table_$uid and syserr_info_$uid are equal. If they are not equal then system initialization will be aborted.

In order to facilitate changing the format of a syserr_table_ or syserr_info_ entry, a version number will be associated with each of these segments. The variables syserr_table_$version_num and syserr_info_$version_num will contain the version number that specifies the format of the entries in their respective segments. These two version numbers do not have to be equal. They will be generated by syserr_table_compiler.

The syserr_table_ codes that will be used in calls to syserr correspond to entries in syserr_table_. Each entry in syserr_table_ will contain all of the information needed by syserr_real to process a syserr call. Since syserr_real must not take a page fault when called by a program that is wired, we must guarantee that syserr_table_ entries referenced by wired programs will themselves be wired. In order to do this syserr_table_compiler will group all of the entries referenced by wired programs at the top of syserr_table_. During system initialization enough pages at the top of this segment to cover all of these entries will be permanently wired. In order to minimize the amount of wired storage needed by syserr_table_ the information kept in each syserr_table_ entry will be only that information absolutely necessary to syserr_real. Below is a description of a syserr_table_ entry.

```
dcl 1 ste    based (ste_ptr)  aligned,
      2 code,                                                /* 1. */
      ( 3 pad          bit(18),                              /* 2. */
        3 table_offset bit(18),                              /* 3. */
      2 info_offset  bit(18),                                /* 4. */
      2 action_code  fixed bin(8),                           /* 5. */
      2 cstring_len  fixed bin(8)) unaligned,                /* 6. */
      2 cstring      char(0 refer(ste.cstring_len));/* 7. */
```

1. table_offset - This word is the value referenced by a syserr_table_ code variable such as syserr_table_$bad_devx.

2. pad - This half of a syserr_table_ code word is reserved for future use. In the initial implementation it will be set to zero.

3. info_offset - This field contains the offset of this entry in syserr_table_, i.e., its own word offset.

4. info_offset - This field contains the offset of the corresponding entry in syserr_info_. There is a one-to-one correspondence between the entries in syserr_table_ and the entries in syserr_info_. This field provides the connection between corresponding entries in these two tables.

5. action_code - This is the syserr action code for this message. (For a list of the valid syserr action codes see the section on the syserr_table_ source language.)

6. cstring_len - This field specifies the length of the formline_ control string for this syserr message.

5. cstring - This field is the formline_ control string for this syserr message.

The syserr_info_ table contains information about syserr messages that is not needed by syserr_real. The reason for splitting up the information about a syserr message into two entries is to minimize the information contained in a syserr_table_ entry. This is important since some syserr_table_ entries are wired. Most syserr_table_ entries will not be wired, but for the sake of consistency it is desirable to make all of the syserr_table_ entries have the same format. The reason for putting the syserr_info_ entries into a separate segment and not putting them in an unwired part of the syserr_table_ segment involves system initialization considerations. That part of the syserr mechanism that writes syserr messages on the operator's console and puts messages into the wired_log is initialized early in collection 1. There is a critical limit to the space that is available during collection 1. The amount of information contained in a syserr_info_ entry may by so great that the syserr_info_ segment would be too large to be used during collection 1. Thus these two segments cannot be combined. The syserr_info_ table will not be used until collection 2 when the syserr logging mechanism is initialized. Below is a description of a syserr_info_ entry.

```
dcl 1 sie  based(sie_ptr)  aligned,
      ( 2 action_code  fixed bin(8),                     /* 1. */
        2 name_len     fixed bin(8),                     /* 2. */
        2 desc_len     fixed bin(17),                    /* 3. */
        2 sort_code    fixed bin(17),                    /* 4. */
        2 format_code  fixed bin(17)) unaligned,         /* 5. */
        2 name         char(0 refer(sie.name_len)),      /* 6. */
        2 description  char(0 refer(sie.desc_len));      /* 7. */
```

1. action_code - The syserr action code is duplicated in this entry for efficiency reasons.

2. name_len - This field contains the length of the syserr message name string.

3. desc_len - This field contains the length of the syserr message description string.

4. sort_code - This field contains a number that is used to sort syserr messages that have been logged. Each class of syserr messages - device errors, audit messages, etc - will be assigned a unique sorting code. For those syserr messages that do not fit into any special class a default value of 0 will be used. (See the section on the user ring processing of syserr messages.)

5. format_code - This field contains a number that can be used to format any binary data associated with this message. It should be very helpful to user ring programs that process syserr messages from the syserr_log. (See the section on the user ring processing of syserr messages.)

6. name - This field specifies the name of the syserr message. It is identical to the entry point name used in the declaration of a syserr_table_ code. Continuing with our example, if this entry corresponds to the syserr_table_ code syserr_table_$bad_devx then this field would contain "bad_devx".

7. description - This string contains a description of this syserr message. It may include a description of the circumstances that cause this message to be used, a description of any variables that may appear in the expanded message string, a description of any action that the operator should take in response to this message, or any other information useful to know about this message.

Ring Zero Syserr Processing

This section describes how the syserr_table_ and syserr_info_ segments are used by syserr_real and syserr_logger to process a syserr message. As an example, the calling sequence to the syserr$message entry point is described in detail below.

syserr$message (syserr_table_code, arg1, ..., argi)

ARGUMENTS:

syserr_table_code (Input) (fixed bin(35)) This argument specifies an offset into the segment syserr_table_. This offset references the entry in syserr_table_ that corresponds to this syserr message.

arg1, ..., argi (Input) These are optional arguments that will be used by formline_ to expand the control string.

The entry point syserr$message is an ALM interface to the entry point syserr_real$message. Using the syserr_table_code argument syserr will reference the syserr_table_ entry for this syserr message. From this entry it will get the action code for this message. All the new syserr entry points will check to see if any stack manipulation is needed. If the action code specifies a fatal error and if other conditions are met then syserr will alter the stack that it is running on so that previous stack history information will be preserved for debugging purposes. Then syserr will call the corresponding entry point in syserr_real using the same argument list that it was called with.

The syserr_real entry point that is called will also use the syserr_table_code argument to make a pointer to the syserr_table_ entry associated with this message. It will get the action code from this entry. It will check that this action code is valid. Contrary to MTB-071 no log code (sorting code) value will be derived from this action code. The control string for this message will be copied from its syserr_table_ entry. Using this control string and the arguments passed by the caller syserr_real will call formline_ to generate an expanded ASCII message. The message will be logged. Based upon the action code, syserr_real will write this message on the operator's console.

This message will be logged by syserr_real in basically the same way that it does now. However, the information put into the wired_log is somewhat different. Below is a description of the new wired_log entry.

```
dcl 1 wmess  based(wmess_ptr)  aligned,
      2 head        like  wmess_header,                /* 1. */
      2 text char(0 refer(wmess.head.text_len)),       /* 2. */
      2 data(0 refer(wmess.head.data_size)) bit(36),   /* 3. */
      2 next_wmess bit(36);                            /* 4. */

dcl 1 wmess_header based  aligned,
      2 seq_num    fixed bin(35),                      /* 5. */
    ( 2 info_off   bit(18),                            /* 6. */
      2 text_len   fixed bin(8),                       /* 7. */
      2 data_size  fixed bin(8),                       /* 8. */
      2 time       fixed bin(71)) unal;                /* 9. */
```

1. head - The header of the wired_log message entry.

2. text - The ASCII message that was expanded from the control string of this syserr message.

3. data - The binary data that is copied into the wired_log by syserr_real. (See the section on binary data.)

4. next_wmess - Used to calculate the address of the next entry in the wired_log.

5. seq_num - The sequence number assigned to this syserr message by syserr_real. The sequence number count is initialized to 1 whenever the syserr_log is reinitialized. Due to the high number of syserr messages that will be generated by the protection audit mechanism this field has been expanded from its previous size.

6. info_off - Offset in syserr_info_ of the entry that corresponds to this syserr message.

7. text_len - Number of characters in the ASCII message string.

8. data_size - Number of words of binary data copied into this message entry. Zero implies that there is no binary data in this entry.

9. time - Raw clock time specifying when the syserr message was put into the wired_log.

When handling the log interrupt, syserr_logger will copy each entry in the wired_log into the syserr_log. It will copy the seq_num, text, data, and time fields from the wired_log. Using the info_off field in the wired_log entry it will generate a pointer to the syserr_info_ entry associated with this syserr message. From this syserr_info_ entry it will get the rest of the data that goes into the syserr_log entry. Below is a description of the new syserr_log entry.

```
dcl 1 smess  based(smess_ptr)  aligned,
      2 head     like  smess_header,                  /*  1. */
      2 name     char(0 refer(smess.head.name_len)),  /*  2. */
      2 text     char(0 refer(smess.head.test_len)),  /*  3. */
      2 data(0 refer(smess.head.data_size)) bit(36), /*  4. */
      2 next_smess   bit(36);                         /*  5. */

dcl 1 smess_header   based  aligned,
    ( 2 next          bit(18),                        /*  6. */
      2 prev          bit(18)) unaligned,             /*  7. */
      2 seq_num       fixed bin(35),                  /*  8. */
    ( 2 action_code   fixed bin(8),                   /*  9. */
      2 name_len      fixed bin(8),                   /* 10. */
      2 text_len      fixed bin(8),                   /* 11. */
      2 data_size     fixed bin(8),                   /* 12. */
      2 time          fixed bin(71)) unal;            /* 13. */
```

1. head - The header of the syserr_log message entry.

2. name - The name of this syserr message.

3. text - The expanded ASCII message.

4. data - The binary data saved for this syserr message.

5. next_smess - Used to calculate the address of the next entry in the syserr_log.

6. next - The offset of the next entry in the syserr_log.

7. prev - The offset of the previous entry in the syserr_log.

8. seq_num - The sequence number of this syserr message.

9. action_code - The action code of this syserr message. It tells how syserr_real processed this message.

10. name_len - Number of characters in the string that specifies the name of of this syserr message.

11. text_len - Number of characters in the ASCII message string.

12. data_size - Number of words of binary data.

13. time - Raw clock time when message logged.

User Ring Syserr Processing

User ring programs may process syserr messages that have been logged. They will be able to get syserr messages directly from syserr_log or from one of the system log segments. They will be able to select syserr messages based on syserr message name, sequence number, action code, the time the message was logged, and sort code. They may print the message text as is since it is already in ASCII and completely expanded. If this message has any binary data they must decide how to format it. This decision can be made using the format code for this message. If the format code is 0 or if it is not known to the program then the binary data may be formatted as if it were an octal dump. However, if the format code is equal to some prearranged value that the user ring programs understand then they will be able to format the binary data in some special way. For example, a format code of 1 may imply that the binary data is SCU data. A format code of 2 may imply that it is history register data, etc.

Neither the sort code nor the format code are found in the syserr_log entry. They are found in the syserr_info_ entry associated with this syserr message. Any other program that wants to find the syserr_info_ entry associated with a syserr_log entry must do the following.

1. Get the syserr message name from the syserr_log entry.

2. Using this entry point name and the segment name "syserr_table_" call hcs_$make_ptr to get a pointer to the syserr_table_ entry that corresponds to this syserr message.

3. Using the info_offset found in the syserr_table_ entry generate a pointer to the corresponding syserr_info_ entry.

It may not be obvious to the reader why the syserr message name is saved in the syserr_log entry instead of the offset of the syserr_info_ entry itself. It is true that if the offset were saved then the algorithm described above would not be necessary. However, this offset is not saved in the syserr_log entry for the following reason. Syserr messages will be saved in the syserr_log and the system log segments for long periods of time, possibly months or even years. During this time it is inevitable that syserr messages will be added and deleted from the system. The syserr_table_ mechanism must be able to process syserr messages that were generated from old versions of syserr_table_ and syserr_info_. Unless these segments are formatted in a very inefficient way the offsets of their syserr message entries will change each time syserr_table_.st is recompiled. Thus we need to put something in the syserr_log entries that will identify a syserr message for all time. The syserr message name is such an entity. If a program is processing a syserr message that has become obsolete, then there will be no corresponding entry in syserr_table_ or syserr_info_. The call to hcs_$make_ptr will not be successful since it uses an unknown syserr_table_ entry point name. The program will know that this syserr message is obsolete and will use default values for the information that it would have found in the syserr_info_ entry.

The ability to add and delete syserr messages from syserr_table_ and syserr_info_ is an important feature. Just as important, however, is the ability to change the information about a syserr message that is kept in these segments. Information relevant to a syserr message at the time it was generated (action code, text, binary data, time) is saved in the syserr_log entry. Information that is used at a later time to process, interpret, and describe this syserr message is kept in its syserr_info_ entry. Changing the description of a syserr message means that the new description will be available for past as well as future instances of that syserr message. At this time, the function of the sort and format codes is not clearly understood. What is understood, however, is that the fields in a syserr_info_ entry such as the description, sort_code, and format_code do not affect the ring 0 processing of syserr messages. They represent a convention understood by the writer of syserr_table_.st source statements, syserr_table_compiler, and user ring programs that process syserr messages from the syserr_log.

<u>Advantages of the Tabularization of Syserr Messages</u>

1. The new syserr calling sequence will generate less object code. The main savings is due to the fact that the formline_ control string is no longer part of the object segment. Also, the most frequently used of the new syserr entry points, syserr$message, has one less argument that the current syserr entry point. Since all calls to syserr are made with descriptors this implies that four words will be saved in each of these calls.

2. One could say that the savings described above is nullified by the space needed by syserr_table_ and syserr_info_ entries. However, this is only partly true. First, the pages used by these two segments are only referenced when a syserr call is actually made. This is an infrequent occurrence. Some syserr messages are almost never used. With the old calling sequence the space used by these calls was in the object text and was therefore active each time the program was executed. Secondly, some syserr messages contain the same message. If called by two separate programs the control string will be duplicated in the object text of each program. The new calling sequence can eliminate this duplication. Many syserr messages have the same control string except for a program name. Such messages could be changed to use the single syserr code by having the program names specified as arguments.

3. The tabularization of syserr messages will result in a significant improvement in the administrative control over the use of this system function. Programmers modifying ring 0 programs will no longer be able to add, delete, or change syserr messages at will. They will have to change syserr_table_.st and this should require an MCR. Changes to syserr_table_.st should be noted on the system change request form.

4. The syserr_table_.st source segment will be an instant source of documentation about syserr messages. The description of the syserr message will be especially helpful. In addition to the source segment itself, a program could be developed that would format this source segment as an actual document. Programs could also be developed that would return selective information about a syserr message.

5. The format and sort codes defined for each syserr message will be very helpful in processing syserr messages that have been logged. New syserr message processing features can easily be added since this whole area of the syserr mechanism is merely a convention among user ring programs.

BINARY DATA

Recently, certain programs have been putting large amounts of binary data into the syserr_log. History registers (128 words) and SCU data (48 words) have been put into the syserr_log. In the future, device status and other information associated with I/O device errors will be logged. The current syserr mechanism does not allow this to be done either conveniently or efficiently.

The major problems involved with logging binary data via the current syserr mechanism are:

1. It is inconvenient for the calling program. It must go through the trouble of breaking up the binary data into pieces that syserr can handle.

2. Because the data must first be broken up, programs usually call syserr with only four words of data at a time. For example, in order to put all of the history register data into the syserr_log 32 calls are made to syserr. The multiplicity of calls that result from giving syserr only a few words at a time is very inefficient.

3. Due to the multiple wired_log entries generated, each of which has header information, and due to the conversion from binary to ASCII, the wired_log entries for 128 words of binary data now uses 648 words. When syserr is called from a program that is masked down to system level the log interrupt is inhibited. If this program repeatedly calls syserr the wired_log will overflow and messages will be lost from the log. Currently, if a program that is masked down to system level attempts to put history register data into the log most of the syserr messages will be lost from the log. With the current implementation, in order to make the wired_log large enough to hold all of the history register data we would have to increase its size to 750 words, five times its current size of 150 words.

4. Since many log entries are needed to put large amounts of data into the syserr_log, it is possible for these entries to be interleaved in the syserr_log with other entries generated by the same program while it is simultaneously running on another processor. It is likely that in such a case the data retrieved from the syserr_log would not be interpretable.

In order to solve these problems the new entry point to syserr described below will be implemented. It is designed to meet the following goals:

1. The calling program must be able to pass binary data to syserr in a convenient manner.

2. A reasonably large amount of data must be processed by a single call to syserr.

3. The binary data should not be converted to ASCII. It should be put into the syserr_log in its original binary format.

```
syserr$binary (syserr_table_code,     data_ptr,       data_size,
               arg1,    ...,    argn)
```

ARGUMENTS:

data_ptr (Input) (ptr) Pointer to the first word of binary data to be logged.

data_size (Input) (fixed bin) The number of words of binary data to be put into the syserr_log. A maximum data size of 128 words will be allowed.

The entry point syserr$binary is an ALM interface to the entry point syserr_real$binary. syserr_real$binary performs all of the functions that are performed by syserr_real$message. It will support all of the defined syserr action codes. It will generate an ASCII string from the formline_ control string found in the syserr_table_ entry for this message. This ASCII string will be placed in the wired_log and later copied into the syserr_log. The ASCII string will be typed on the operator's console if this is specified by the syserr action code.

In addition, syserr_real$binary will copy into the wired log all of the binary data specified by the data_ptr and data_size arguments. This data will not be converted into ASCII. It will not be typed on the operator's console regardless of the action code. When syserr_logger handles the log interrupt it will copy all of this binary data into the corresponding syserr_log entry. The log entry header for both the wired_log and syserr_log will be changed to include the size of the binary data that is contained in the entry. If this value is zero then there is no binary data. The other syserr_real entry points will always set this field to zero.

User ring programs will have to convert any binary data found in a log entry into a printable format. Instead of this conversion being done by syserr_real, a critical ring 0 program, it will be done in a higher ring. The format code found in the syserr_info_ entry can be used to tell user ring programs how this binary data should be formatted. As a default the binary data can be printed as if it were a dump.

ERROR TABLE MESSAGES

Many calls to syserr contain an error_table_ code as one of the arguments to formline_. This code is usually converted and printed as an octal number. This method of using error_table_ message codes within syserr messages has the following major disadvantages.

1. It is not easy for an operator who sees such a message typed on the operator's console to know what error_table_ message is being referenced. He must look in the source listing of error_table_.alm. Using the octal entry offset obtained from the syserr message he can then find the error_table_ message.

2. Finding the error_table_ message from these syserr messages once they have been logged may often be impossible. The error_table_ entry offsets that reference a previous version of the error_table_ will not be valid.

The new syserr entry point described below is intended to improve the use of error_table_ messages with syserr messages. Since this entry point will reference the unwired segment, error_table_, it must not be called by any programs that cannot take page faults.

```
syserr$error_code (syserr_table_code, error_table_code, arg1,
                   ..., argn)
```

ARGUMENTS:

error_table_code (Input) (fixed bin(35)) A standard error_table_ code.

The entry point syserr$error_code is an ALM interface to the entry point syserr_real$error_code. It performs all of the functions that are performed by syserr_real$message. In addition, it will use the error_table_code argument to obtain a message string from the system error_table_. This message string will be appended to the expanded syserr message string. The concatenated string will be logged. If appropriate, the concatenated string will be typed on the operator's console.

SYSERR SOURCE LANGUAGE

This section discusses the source language used to define syserr messages. The definition of all of the syserr messages will be combined in the single segment, syserr_table_.st. The definition of a syserr message is comprised of several statements. An informal description of these statements is given below.

General Statement Syntax

```
<statement>::= <statement name>: <statement variable>;
```

NAME STATEMENT

```
<name statement>::= name: <syserr message name>;
```

A name statement must be the first statement in the definition of a syserr message. The statement variable is the name of this syserr message. This name will become an entry point in the segment syserr_table_. Each syserr message name must be unique within syserr_table_.st.

END STATEMENT

```
<end statement>::= end: <syserr message name>;
```

An end statement must be the last statement in the definition of a syserr message. The statement variable is the name of this syserr message. It must match the name specified on the preceding name statement. Between the name statement and the end statement will be all of the other statements that define this syserr message. These statements may be in any order. Only one statement of each type is allowed in any one syserr message definition. The main purpose of this statement is for the convenience of those perusing a listing of the syserr_table_.st source segment. it identifies a syserr message definition that has spanned one or more pages of the listing.

ACTION STATEMENT

```
<action statement>::= action: <action code>;
<action code>::= {fatal|write|write_alarm|log|log_only}
```

This statement defines the syserr action code for this syserr message. This statement is not optional. Syserr messages that use variable action codes (for example those that use DEBG card values) must now be specified as separate messages. There must be one message for each possible action code. This is a rare case and its use should be discouraged. The meaning of the various action codes is given below. The reader should note that the previously supported syserr action involving the termination of a process is no longer supported.

fatal - The message will be logged and then typed on the operator's console with alarm. Then a "Multics Not In Operation" message will be typed. Then Multics will be crashed.

write - The message will be logged and then typed on the operator's console without alarm.

write_alarm - The message will be logged and then typed on the operator's console with alarm.

log - The message will be logged. The message will not be written on the operator's console. However, if the message could not be logged due to a lack of space in the wired_log buffer then the message will be typed on the operator's console without alarm. The string "*LOST" will be prefixed to the syserr message.

log_only - The message will be logged. The message will not be written on the operator's console. If the message cannot be logged it will be lost without notification to the operator.

CONTROL STATEMENT

<control statement>::= control: "<control string>";

The control statement is used to define the formline_ control string that is to be used to expand this syserr message. The variables defined within this formline_ control string must match the arguments passed in the call to syserr. The control string variable must be within quotes. Any quotes within the control string itself must be expressed as double quotes. This statement is not optional.

STATUS STATEMENT

<status statement>::= status: <status variable>;
<status variable>::= {wired|active|paged|init}

The status statement is used to define where this syserr message is to placed in the syserr_table_. All syserr message entries are grouped into one of four status classes. status classes. All of the entries from the same status class will be packed together in syserr_table_. The wired status class entries will be placed at the top of syserr_table_, followed by the active status class entries, followed by the paged status class entries, and lastly followed by the init status class entries. The number of pages in syserr_table_ used by the wired status class entries will be calculated by syserr_table_compiler. These pages will be permanently wired at system initialization time. This is done in order to fulfill the requirement that all entries in syserr_table_ that are referenced by syserr_real on behalf of wired programs must themselves be wired. This statement is optional. If it is missing a default status class of wired will be assumed. The exact meaning of these four syserr message status variables is:

wired - One of the calls to syserr that references this syserr message comes from a program that is wired.

active - This syserr message will be referenced only by paged programs. This syserr message is frequently used. The purpose of this status class is to hopefully put some of these frequently used paged syserr message entries into any unused space in the last page used by the wired syserr message entries.

paged - This syserr message will be referenced solely by paged programs.

init - This syserr message will be referenced solely during system initialization. By isolating this type of syserr messages we can place them in pages at the end of syserr_table_. These pages will never be referenced after system initialization. Some calls to syserr come from initialization programs that are wired. However, this case occurs only during collection 1 when all of syserr_table_ is wired. Thus syserr messages whose status is both wired and init should be defined as init.

DESCRIPTION STATEMENT

<description statement>::= description: "<description string>";

This statement is used to specify a description of the syserr message. The description string may contain any characters suitable for printing. Quotes within this string must be expressed as double quotes. The description string may be as long as necessary to completely describe the meaning and reason for this syserr message. If appropriate, it should include a description of the action to be taken by the operator in response to this syserr message. This is an optional statement. If it is missing a null string will be used as a default.

SORT STATEMENT

<sort statement>::= sort: <sort code>;

The sort code variable specified in this statement must be a non-negative decimal number. This variable specifies that this syserr message belongs to a particular sort class. The user ring programs may support an option that enables them to process only those syserr messages that belong to a particular sort class. The sort class numbers must be used according to conventions agreed upon by the writers of syserr_table_ statements. This statement is optional. If it is missing, a default sort code value of 0 will be used.

FORMAT STATEMENT

<format statement>::= format: <format code>;

The format code variable specified in this statement must be a non-negative decimal number. This variable specifies the format to be used when printing binary data. The format codes must be used according to conventions understood by the writers of syserr_table_ statements and the user ring programs. This statement is optional. If it is missing a default format code value of 0 will be used. A format code of 0 implies that the binary data is to be printed as an octal dump.

NOTES

Spacing characters (blanks, tabs, new line characters, new page characters) may appear between any statement and between any element of a statement. PL/I type comments strings, /*...*/, may appear anywhere that a spacing character may appear. (See Appendix A for sample definitions of syserr messages using this source language.)

IMPLEMENTATION PLAN

1. The syserr_table_compiler must be implemented. It probably should be coded using the reduction_compiler.

2. The syserr_table_.st source segment must be generated. As calls to syserr are converted their syserr messages must be defined in syserr_table_.st.

3. The programs syserr and syserr_real must be changed. The current syserr entry point must be maintained until all calls to syserr have been converted to use one of

the three new entry points. Since there is no syserr_table_code argument passed to the current syserr entry point, default table information will be used. In order to work with the new wired_log and syserr_log entry formats this entry point will use dummy syserr message entries that will be defined in syserr_table_.st. There will be one dummy message entry for each action code. The control strings in the syserr_table_ entries for these dummy messages will not be used by syserr_real. The three new syserr entry points and their corresponding syserr_real entry points must be implemented.

4. A change should be made to the syserr message text that is typed on the operator's console. The sequence number of the message should be included in the message text. This sequence number may then be used by the operator to obtain more information about the message.

5. The program syserr_logger must be changed to work with the new wired_log and syserr_log entry formats.

6. The program init_collections must be changed. The program syserr_log_init must be moved from collection 1 to collection 2.

7. The syserr_data data base must be changed. The size of the wired_log buffer should be doubled to 300 words. This will allow at least two syserr messages with the maximum amount of binary data to be logged.

8. All user ring programs that process syserr messages from the syserr_log must be changed to use the new syserr_log entry format. They must be changed to process binary data and to use the format and sort codes that are available in syserr_info_.

9. A new program should be implemented that would print selective information from a syserr_info_ entry. It should be able to find this entry given either a syserr message name or a valid syserr message sequence number.

10. A new program should be implemented that can generate a formal document from the syserr_table_.st source segment.

11. A new program should be implemented that would merge private versions of the syserr_table_.st source segment into one source segment. Any number of source segments of the type aaaaaa.st could be used as input. The result would be a new source segment with the name syserr_table_.st.

12. The programs that call syserr will have to be changed. They do not all have to be changed at once. The syserr calls that involve binary data should be changed first. Then as ring 0 programs are added or changed we can require that they use the new syserr calling sequence in order to be installed.

Appendix A
Sample Syserr Message Definitions

```
/*   This is a sample definition of a syserr message
 *   using the syserr_table_compiler source language.
 *   call syserr$message (syserr_table_$bad_devx,devx);
*/
name:       bad_devx;
 action:    fatal;      /* Crash the system. */
 control:   "iom_manager: bad devx ^o supplied.";
 status:    wired;
 format:    0;          /* No binary data. */
 sort:      1;          /* Sort class 1. */
 description:
          "This syserr message is generated when
           iom_manager is called with a bad device index.
           There is nothing the operator can do. ";
end:        bad_devx;


/*   call syserr$message (syserr_table_$mylock,"name",lockp);
*/
name:       mylock;
 action:    fatal;      /* Fatal error. */
 control:   "^a: mylock error on ^p.";
 status:    paged;      /* Fatal action => not active. */
 format:    0;
 sort:      2;          /* File system error. */
 description:
          "This syserr message is generated by file
           system programs that find a lock already
           locked to a process. ";
end:        mylock;
```