Error Trapping

Error Trapping

Table of Contents

Chapter 1. Error Trapping

Actions: When an Error Condition is Detected

When a disconnection error is detected during program execution, the process terminates and the error is reported as a ***SIGHUP*** signal in the log file.

When other error conditions are detected during program execution, the system interrupts execution and performs the following actions:

- Terminates any active **FOR** loops and indirection on the current line.
- Sets **$ZERROR** to reflect the error.
- Appends the appropriate error codes to the **$ECODE** special variable. **NOTE** : If **$ECODE** is updated via the **SET** command, the entire value of **$ECODE** is replaced.
- Updates the **$STACK** system function array to reflect the error, setting the **ECODE** , **MCODE** , and **PLACE** sub-nodes of **$STACK($STACK)** .
- If no error handling is specified at any execution level (**$ETRAP** is null, all **NEW** ed values of **$ETRAP** also are null at all execution levels, and there are no non-null **$ZTRAP** s at any execution level), generates an execution trace-back and terminates all the execution levels. If the user is in programmer mode, the current device is changed to the principal device and a command prompt is issued by the system. If programmer mode is not enabled, the partition is terminated after invoking the default error handler.

Actions: When Error-Handling is Specified

The following steps are performed if error handling is specified at the current or at an earlier execution level.

- If **$ETRAP** is not null, M21 performs an internal **GOTO** to the following two lines of M code:

  ... $ETRAP value ...

  QUIT:$QUIT "" Q

  The second line handles the case in which the **$ETRAP** code completes and execution continues on the second line. The **QUIT** commands are set up so that the current execution level terminates correctly depending on whether or not it was invoked as an extrinsic.

- If **$ZTRAP** is non-null, M21 internally performs the following line of M code:

  GOTO @($ZTRAP)

- As part of the **GOTO** , the value of **$ZTRAP** is reset to the null string. If additional error handling is required at the current execution level, **$ZTRAP** must be re-established. M21 terminates the current execution level. This includes restoring values stacked by the **NEW** command (possibly including **$ESTACK** and **$ETRAP**).

- M21 terminates any active **FOR** loops and indirection in the current execution line. M21 continues with step 1 above until an execution level is reached that does specify error handling via **$ETRAP** or **$ZTRAP** .

To ensure that error handling is uniquely defined, M21 does not allow both **$ETRAP** and **$ZTRAP** to be non-null at the same time. When **$ETRAP** is established via the **SET** command, M21 internally resets **$ZTRAP** to the empty string. When **$ZTRAP** is established via the **SET** command, M21 internally performs a **NEW** command on **$ETRAP** and then sets its current value to the empty string (a **QUIT** from this execution level restores **$ETRAP** to its original value). This interaction between **$ETRAP** and **$ZTRAP** allows existing applications that rely on **$ZTRAP** to coexist with applications that rely on the ANSI standard method of error trapping.

Default Error Processing

If error handling is not defined, the error processing action to be taken when an error occurs includes a display of the execution trace-back on the principal device and an invocation of the M21 default error handler.

The execution trace-back displays the current value of **$ZERROR** , a formatted display of each execution level (from the deepest to the initial), followed by a redisplay of **$ZERROR** (in case the initial value scrolled off the screen). The formatted display of a level includes its nesting level (**$STACK**); the routine reference of the line being performed or ' ***XECUTE*** ' if inside an **XECUTE** string; and a copy of the line of M code that is being performed, if available (p-code-only routines display null text lines).

Following the execution trace-back, the **^%MuErr1** utility is invoked to store the partition's current status for later review. This includes saving a copy of all the local symbols for the partition, **$IO** , **$PRINCIPAL** , **$ZREFERENCE** , and so on. These values can be examined using the **^%MuErr** utility.

User-Defined Error Processing

The programmer can specify the M code that is to be performed when M21 detects an error. The programmer can use either the 1995 ANSI Standard error processing features such as **$ETRAP** or **$ECODE** or the M21-specific features, **$ZERROR** and **$ZTRAP** .

User-Defined Error Processing - $ETRAP

To use the **$ETRAP** mechanism, the value of **$ETRAP** is set to an M string that is executed when an error is detected. For example:

```
SET $ETRAP="S X=$X,Y=$Y G ^ERROR"
```

In this example, when an error is detected, M21 sets the local variables **X** and **Y** to the current values of **$X** and **$Y** respectively, and then transfers control to the first line in routine **^ERROR** . This routine can attempt to recover from the error, record the error and any application-specific information, or allow the error to propagate to the previous execution level where it can be handled, recorded, or continue to be propagated to earlier execution levels. If the error processing routine does not clear **$ECODE** before exiting from the execution level where an error occurred, error processing is still in effect in the new execution level. A common M error handling technique is to **NEW $ESTACK** at the top menu level of the application and to continue in error processing mode until **$ESTACK** returns to zero.

User-Defined Error Processing - $ZTRAP

To use the **$ZTRAP** mechanism, the value of **$ZTRAP** is set to an M entry reference to which control will be transferred when an error is detected. For example:

SET $ZTRAP="ENTRY^ERROR"

When an error is detected, M21 terminates execution levels one at a time until it reaches an execution level at which **$ZTRAP** is not null. M21 then performs a **GOTO** to the entry point contained in **$ZTRAP** and also resets **$ZTRAP** to the empty string. The error trap routine can attempt to recover from the error, record the error and any application-specific information, or force the error to be propagated to an earlier execution level that has a non-null **$ZTRAP,** where the error can be handled, recorded, or continue to be propagated to earlier execution levels.

Because, on error, M21 internally resets **$ZTRAP** to null, **$ZTRAP** must be explicitly **SET** to a new non-null value if additional errors are to be trapped at the current execution level.

Processing Trapped Errors

When a **DO** or **XECUTE** command is processed, the M21 system creates a new execution level. This is done so that on termination of the **DO** or **XECUTE** command, either through an implicit **QUIT** (end-of-routine encountered) or an explicit **QUIT** , the system can return to the point at which the command was initiated.

As a point of reference, each execution level created by a **DO** or **XECUTE** command is numbered. Programmer mode is considered to be execution level 0, the first **DO** or **XECUTE** command is level 1, the second is level 2, and so on. The level that is in the process of execution is generally referred to as the current execution level.

While there is only one **$ETRAP** special variable (there is not a different one at each execution level), it can be stacked using the **NEW** command. This provides a separate **$ETRAP** for each level so that each execution level can customise its error handler independent of other error handling on the execution stack.

Alternatively, the **$ZTRAP** special variable can be set at each execution level. This means that each execution level in an application can establish its own error trapping routine and error recovery procedures.

After an error has been trapped and processed by the specified routine, several techniques can be used to exit from the error trap routine. One method is to use a **GOTO** command so that control passes to a new routine, and program processing remains at the same execution level. This is the most common form of error recovery.

Alternatively, a **QUIT** command can be used to return to the previous execution level. When the **$ETRAP** mechanism is used, execution continues at the next lower level (the current execution level minus one), based on the value of **$ECODE** . If **$ECODE** is null, **QUIT** returns control normally to the previous level, with no further error processing. If **$ECODE** is not null and **$STACK($STACK,"ECODE")** is null (returning from a level with an error to a level at which no error occurred), error processing continues at the new level.

When using the **$ZTRAP** mechanism, a **QUIT** command returns control normally to the previous level, with no additional error processing. If the error condition has not been handled, a **ZQUIT** command can be used to continue error processing on the previous level.

Using the System Error Trap

The standard error-trapping utility programs supplied with the system provide the programmer with an alternative to writing error-trapping routines. These standard utilities include a routine to trap errors **(%MuErr1**) and a routine to report trapped errors **(%MuErr**). These routines are described in this section.

%MuErr1

When an error occurs and the **%MuErr1** utility is invoked, the routine stores the values of the **$HOROLOG** , **$JOB** , **$IO** , **$STORAGE** , **$TEST** , **$ECODE** and **$ZERROR** special variables; the key of the last global referenced (**$ZREFERENCE**) and the contents of the local symbol table. It also stores the UCI number, principle device number, current device number, process ID, M routine name, M routine line number, parent job number and the number of the last job JOBbed by this partition.

%MuErr

This utility reports errors that were trapped using the **%MuErr1** routine. It allows one to display, print, delete, or summarise errors.

Using Old Style Error Trapping

In certain old versions of Digital Standard MUMPS (DSM) and Micronetics Standard MUMPS (MSM), the system discarded the entire contents of the **DO** / **XECUTE** stack when an error occurred. This was done prior to the system passing control to the error-trap routine specified by the **$ZTRAP** special variable. In newer versions of these implementations of M, as in M21, the **DO** / **XECUTE** stack remains at the same level when an error occurs.

Existing applications that rely on the **DO** / **XECUTE** stack being cleared using the error trapping techniques provided in these early versions of M will not work properly. The recommended solution to this problem is to modify the error-trapping code so that it does not rely on these outdated techniques. However, this may not be possible in all cases. As a convenience to our users, a compatibility mode has been provided which enables error processing to behave as it did in these old versions of M.

The compatibility mode has been implemented as an extension to the **BREAK** command. The following describes the implemented compatibility extensions:

BREAK 2 Old mode DSM/MSM error trapping

BREAK -2 Normal M21 error trapping

When a job is started, the default value is normal M21 error trapping (**BREAK -2**). To enable the old style of error trapping, a **BREAK** command with an argument **2** (**BREAK 2**) is inserted at the beginning of the application before any other code is executed. It is strongly recommended that applications be modified to take advantage of the new error processing capabilities.

Format of $ZERROR

The **$ZERROR** special variable contains descriptive information about the most recent error. This information is also provided as a piece of the **$ECODE** special variable when the **$ETRAP** mechanism of error trapping is used and the error is not one of the errors that has an ANSI M code number. The format of this descriptive information is as follows:

<Code>Offset^RoutineName:Command:Argument:Major:Minor:AddInfo

In this display:

Code is the error code

Offset is the location (relative line number) within the routine

RoutineName is the name of the routine

Command is the relative number of the command in the line that is in error

Argument is the number of the argument within the command

Major and **Minor** are the M21 error numbers

AddInfo contains additional information about the error

Refer to ***M21 Error Codes*** and ***M21 Error Numbers*** in this document for additional information.

The M21 system maintains the **Command** and **Argument** parameters only when the debugger (**%DEBUG**) is active. **AddInfo** is not maintained for all error types. For example, when a disk error (<**DKHER**>) occurs, **AddInfo** contains the block number that received the error.

M21 Error Codes

The following table lists the error codes that are produced by the M21 system and describes how the errors are caused.

Table 1-1.

Code	Explanation

<BKERR>	The interpreter encountered a BREAK command while executing a program. The BREAK command is used primarily for program debugging and allows the user to inspect the program, local variables, and global variables at a specified location within the program. Program execution may be resumed by using the ZGO command.
<CLOBR>	An attempt was made to overlay the current routine by issuing a ZLOAD, ZREMOVE, or ZINSERT command from within the routine.
<CMMND>	The interpreter encountered an illegal or undefined command.
<DIVER>	An attempt was made to divide a number by zero.
<DKFUL>	No space is available on the disk within the expansion limits of the current UCI. Information that the system was trying to write to the disk was lost. Database corruption can result.
<DKHER>	The M21 system encountered an unrecoverable error while trying to read from or write to a disk-type device. This is caused by a hardware malfunction or an attempt to reference a block that is outside the physical bounds of the disk.
<DKSER>	A block read in from the disk was not the type expected by the system or the block's internal structure was invalid. This can occur if the database is corrupted by a hard system failure such as a power failure, or a hardware malfunction such as an unrecoverable disk error on a write operation.
<DMAIN>	Invalid use of domain (mnemonic namespace).
<DPARM>	Invalid use of parameter passing.
<DSCON>	The current device has been disconnected from the system.
<DVTRN>	Device translation error.
<ECODE>	$ECODE has been set to a non-null value.

<ESTAP>	The system expression stack overflowed while evaluating a complex expression or saving a large routine.
<EXPER>	An exponentiation error was detected.
<EXSYS>	An error occurred while attempting to send a cross-system request (M21 to M21 or DDP) or receive and process a request from a remote system.
<FORMT>	An attempt to access a global using networking encountered a different collating sequence or encoding for the global than that expected.
<FUNCT>	The interpreter encountered an undefined function or a function that was used improperly.
<INDER>	The interpreter encountered illegal or incorrect use of the indirection operator.
<INHIB>	Access to a database using networking failed because database read or write is disabled on the remote machine.
<INRPT>	The operating system received an interrupt (BREAK) from the terminal while the BREAK function was enabled (a BREAK 1 command was in effect).
<ISYNT>	An attempt was made to insert an illegal line of text or code into the routine buffer. This can occur if the line of code is too long, the line label is invalid, or if the character separating the line label and the line body is incorrect. When an insert is performed in direct mode, the separator character must be a tab. If ZINSERT is used, the separator character must be one or more spaces or tabs.
<LEVEL>	The maximum execution level for the job has been exceeded.
<LFULL>	An attempt to LOCK or ZALLOCATE a variable resulted in the lock table overflowing. The LOCK or ZALLOCATE is not honoured.
<LINER>	A reference was made to a line that does not exist within the body of the routine.
<LKMEM>	An attempt to LOCK or ZALLOCATE a variable resulted in an error trying to allocate shared memory. The LOCK or ZALLOCATE is not honoured.

<LOMEM>	An error occurred when trying to allocate memory to increase the size of the local partition.
<MAPER>	A disk block that is being freed is already marked free in the corresponding MAP block.
<MERGE>	A MERGE command was issued in which either the right hand side or the left hand side of the equals is not a local or global reference, or one operand is a descendant of the other operand.
<MINUS>	The interpreter found a negative number or zero when it expected a positive number.
<MODER>	An attempt was made to access the device in a mode that is not consistent with the parameters specified when the device was opened.
<MMEM1>	An attempt to allocate system memory using ***malloc*** failed.
<MTERR>	An error occurred during an input or output operation to a magnetic tape device.
<MXMEM>	A memory address specified as an argument to the VIEW command or $VIEW function is outside the limits allowed by M21.
<MXNUM>	The value of a number is greater than the largest number allowed by the system.
<MXREC>	An attempt has been made to write a record to a host file, IJC device or tape device that exceeds the record size limit.
<MXSTR>	The value of string exceeds the maximum length allowed by the system. The maximum is 255 characters (or 511 as an option) for global variables, and 4092 characters for local variables.
<NAKED>	Access to a global variable using the naked indicator is invalid. This can occur if the naked indicator was not previously set by a global reference or if the previous global reference did not include subscripts.
<NODEV>	The system intercepted an attempt by the program to OPEN a device that has not been defined to the system.

<NOIMP>	The configuration file parameter to treat VIEW and $VIEW commands as not implemented has been set and a VIEW or $VIEW command has been encountered.
<NOPEN>	The system intercepted an attempt by the program to USE a device that was not previously OPENed by the program.
<NOPGM>	A reference was made to a program that does not exist in the job's routine search path.
<NOSYS>	A reference was made to a non-existent system through Distributed Data Processing (DDP) or to a non-existent volume group through an extended global notation.
<NOUCI>	A reference was made to a non-existent UCI through an extended global notation.
<PCERR>	The interpreter found an illegal post-conditional or the post-conditional argument is invalid.
<PLDER>	The routine that is being loaded has down-level p-code or does not have the correct type of p-code for a remote volume group executing a routine.
<PROT>	An attempt was made to access a protected global that the user is not authorized to access. This error also can occur if an attempt is made to save a program with a name that begins with a percent sign (%) in any UCI other than the Manager's UCI.
<SBSCR>	The subscript used in a local or global variable reference is invalid. This can occur if the subscript is null; the subscript contains the ASCII NULL character; the length of a subscript is greater than 255 characters; or the combined length of the global reference (global name, parentheses, subscripts, and commas) is greater than 255.
<SSVN>	Error involving the use of SSVNs.
<SYNTX>	The interpreter encountered a syntax error in the line that is being executed.

<SYSTM>	The system encountered an internal error. The exact error message, including major/minor error numbers, should be reported to technical support.
<UNDEF>	The operating system intercepted an attempt to reference a non-existent local, global, or structured system variable or a non-existent object method or property.
<VALUE>	An invalid value, which exceeds a system imposed limit, has been specified. The major and minor error codes determine what limit has been exceeded.
<VWERR>	An attempt was made to access a device in shared VIEW buffer mode without ownership of the VIEW device (device 63). This error also occurs if the VIEW device is closed before the device that was opened in shared VIEW buffer mode.
<XCALL>	The function name specified on an external routine reference does not exist, or the parameters specified are invalid.
<ZAP'D>	The job was killed or interrupted by the KILLJOB or ***m21info*** utilities.
<ZSAVE>	One of the lines in the routine that is being compiled is too large to fit in the disk buffer. The line should be split into two or more lines to correct the problem.
<Zxxxx>	A ZTRAP command was issued with "xxxx" as the argument.

M21 Error Numbers

The following table lists the major and minor error numbers generated by M21 and describes how each error is caused.

Table 1-2.

Major	Minor	Explanation
1	-	***Command type error***
	0	Unrecognised comma
2	-	***Argument type error***

	1	Missing parenthesis
	2	Missing or bad colon
	3	Missing or bad equal
	4	Missing or bad local variable
	5	Missing or bad globa variable
	6	Missing or bad functi
	7	Missing or bad routir name
	8	Missing or bad routir label
	9	Missing or bad routir offset
	10	Indirect argument er
	11	Argument condition
	12	Bad argument delimi
	13	Bad command
	14	Missing brace
	15	Missing square brack
	16	Invalid argument cou
3	-	***Expression type error***
	0	Bad special variable r
	1	Bad system function
	2	Bad local variable
	3	Bad global variable
	4	Bad string constant
	5	Bad numeric constan
	6	Unbalanced parenthe
	7	Invalid syntax in terr
	8	Bad operator
	9	Bad delimiter
4	-	***Reference type errors***
	0	Undefined local varia
	1	Undefined global var

	2	Undefined label
	3	Undefined routine
	4	Invalid naked referen
	5	Non-existent device
	6	Unsubscripted local reference required
	7	Variable reference required (no expressio
	8	ZLOAD usage error during routine execut
	9	Undefined UCI refere
	10	Attempted ZINSERT invalid line
	11	Unknown data type
	12	Missing function parameter
	13	Undefined system in cross-system referenc
	14	Global access protect violation
	15	VIEW restriction viol
	16	Reserved for future u
	17	Formal list not entere DO command
	18	QUIT with argument inside FOR scope
	19	QUIT with argument routine not extrinsic
	20	Argumentless QUIT, routine was extrinsic
	21	Extrinsic subroutine ended without Q parameter
	22	Label requires a form
	23	Actual parameters ex number in formal list
	24	Formal list paramete subscripted variable
	25	Duplicate variable na in formal list

	26	Passing value by refe in JOB not allowed
	27	Too many parameter JOB command
	28	A JOB command parameter is too long
	29	Unable to delete rout blocks
	30	Unable to reload call routine
	31	Unable to ZSAVE rou
	32	Invalid first paramete $ZINFO function
	33	Invalid $ZINFO table element
	34	Invalid $ZINFO argu
	35	LHS of MERGE is no local or global
	36	RHS of MERGE is no local or global
	37	Attempt to MERGE w descendent
	38	Device type does not allow the use of doma
	39	USE of domain not specified on OPEN
	40	Undefined domain specified on OPEN
	41	Reference to undefin SSVN
	42	Reference to undefin SSVN node
	43	Attempt to modify re only SSVN
	44	Unknown or unsupp SSVN operation
	45	Unknown device translation table
5	-	***Value-type errors***
	0	Exceeded maximum string length - 255

	1	$SELECT function er
	2	Divide by zero
	3	Negative number
	4	Maximum number
	5	Device not open
	6	Invalid memory add
	7	String value required
	8	Indirection resulted i value
	9	Indirection included than name
	10	Selected partition no active ($VIEW)
	11	Invalid VIEW or $VI argument
	12	Function parameter c range
	13	Invalid subscript
	14	Device not open for a type attempted
	15	$ECODE set to non-n value
	16	Not allowed to write 0
	17	Invalid use of shared mode on VIEW buffe
	18	Raised zero to non-positive power
	19	Raised negative num non-integer power
	20	Spooler I.O error
	21	Spooler error
	22	Routine cache error
	23	Exceeded maximum string length - 1024
	24	Reserved for future u
	25	Exceeded current maximum record leng
	26	Exceeded maximum record length - 255

	27	Exceeded maximum record length - 16384
	28	Value less than minir
	29	Value greater then maximum
	30	Exceeded maximum variable length
	31	Exceeded maximum string length
	32	Exceeded maximum pcode size for a block
	33	Exceeded maximum routine line length
	34	Invalid value
	35	Invalid block type
	36	Invalid data type
6	-	***Environmental errors***
	0	Break key pressed
	1	Unable to malloc a ch of memory
	2	HALT command exe
	3	LOCK table full
	4	BREAK command executed
	5	Expression stack ove
	6	Reserved for future u
	7	Old p-code needs to l re-ZSAVED
	8	Reserved for future u
	9	Reserved for future u
	10	Reserved for future u
	11	Reserved for future u
	12	I/O error on termina operation
	13	I/O error on magneti tape operation
	14	P-code too long to fit one block

	15	ZQUIT error
	16	Reserved for future u
	17	ZTRAP command iss
	18	Job has been killed
	19	Local Symbol Table - of memory
	20	Lock Table - out of memory
	21	Reserved for future u
	22	Maximum execution reached
7	-	***Disk-type errors***
	0-14	Block type mismatch number is expected ty
	20	Bad block type reque
	21	Hardware disk I/O e
	22	Database full conditi
	23	Block number misma
	24	Key/data exceeds maximum length
	25	Cannot open request database
	26	Block being freed is already free
	27	Invalid block numbe
9	-	***Networking errors***
	0	Cross-system request failure
	1	CONFIGUREXSYS n to YES
	2	Could not open local stream socket
	3	Connect to request se failed
	4	Send to request serve failed
	5	Write to request serv failed
	6	Input buffer overflow getting XSYS respons

	7	Sequence no. mismat getting XSYS respons
	8	Request server closec end of socket
	9	Read from request se failed

$ECODE Errors

The following table lists the **$ECODE** errors generated by M21 and explains how the errors are caused.

Table 1-3.

Code	**Explanation**	**Equivalent $ZERRO value**
M1	Naked indicator undefined	<NAKED>:::4:4
M2	Invalid "P" parameter in **$FNUMBER**	<SYNTX>:::5:37
M3	**$RANDOM** seed less than 1	<SYNTX>:::5:24
M4	No true condition in **$SELECT**	<SYNTX>:::5:1
M6	Undefined local variable name	<UNDEF>:::4:0
M7	Undefined global variable name	<UNDEF>:::4:1
M8	Undefined subscripted system variable name	<UNDEF>:::4:41
M9	Divide by zero	<DIVER>:::5:2
M11	No parameters passed	<DPARM>:::4:17
M13	Line reference not found	<LINER>:::4:2
M16	**QUIT** argument not allowed	<DPARM>:::4:19
M17	**QUIT** argument required	<DPARM>:::4:20
M18	Fixed length **READ** not greater than zero	<SYNTX>:::5:38
M19	Cannot copy a tree or subtree into itself	<MERGE>:::4:37
M20	Formal argument list required	<DPARM>:::4:22

M26	Non-existent environment	<NOUCI>:::4:9
M40	Call-by-reference in **JOB** actual	<DPARM>:::4:26
M43	Invalid range value (**$X** , **$Y**)	<MINUS>:::5:3
M58	Too many actual parameters	<DPARM>:::4:23

Chapter 2. $STACK

Returns information about the execution path leading to the current level and about any errors that may have occurred.

Syntax

$ST{ACK}(Level{,StackCode})

Definition

Level An integer expression that specifies the execution level for which information should be returned.

StackCode A string expression that specifies the type of information to be returned.

The single operand **$STACK** function provides the following information about the execution stack.

- If **Level** evaluates to -1, the function returns the current execution nesting level. The **$STACK** special variable returns the same value as **$STACK(-1)** .
- If **Level** evaluates to 0 and if the partition was initiated via the **JOB** command, the function returns 1. Otherwise, it returns 0.
- If **Level** evaluates to a positive integer that is less than or equal to **$STACK(-1)** , the function returns a value that indicates how that execution level was initiated. If initiated by a command, the function returns the name of the command fully spelled out and in uppercase (for example: **DO** or **XECUTE**). Otherwise, if it was initiated by an extrinsic function or variable, then the function returns **$$** .
- If **Level** evaluates to a value greater than **$STACK(-1)** , the function returns an empty string.
- All other values of **Level** are reserved for future extensions.

The two-operand **$STACK** function provides information about the execution level specified using the first operand. The second operand specifies the information to return. This operand can be in uppercase, lowercase, or mixed case, all being equivalent.

- If **StackCode** evaluates to **ECODE** , the function returns the list of error codes added at this level or the empty string if no errors occurred at this level.
- If **StackCode** evaluates to **MCODE** , the function returns the text of the line identified by **$STACK(Level,"PLACE")** .
- If **StackCode** evaluates to **PLACE** , the function returns the last command executed at the specified level. If the location is a routine line, the format of the function value is ***+offset^routine*** . If the location is an **XECUTE** command string, the function returns the at-sign character (**@**).
- All other values of **StackCode** are reserved for future extensions.

Examples

Table 2-1.

Function	Return Value/Description

$STACK(-1)	**10** The current nesting level is 10.
$ST(10,"ECODE")	**,M9,M6,** Both a division by zero and a reference to an undefined local occurred at the same level. This can happen if the second error occurred inside the error trap itself.
$ST(10,"MCODE")	**WRITE !!,A/B** This shows the line executing when the error occurred.
$ST(10,"PLACE")	**+5^ERRHAND** The last error occurred at line **+5** in routine **ERRHAND** .

Chapter 3. $ECODE

Contains a list of error codes encountered by the application.

Syntax

$EC{ODE}

Definition

The **$ECODE** special variable contains a string that identifies the errors encountered by the application. The string value of **$ECODE** is in the following format:

,ErrorCode1,ErrorCode2, ... ,

Note that a comma precedes and follows each error code. Error codes are in one of the following formats:

Mnn where **nn** is an integer specified by the ANSI standard

Uxx where **xx** is any user-defined string not containing a comma

Zxx where **xx** is defined by the version of M21 (M21-specific)

The **M** values are integer numbers specified by the 1995 ANSI M standard (and subsequent Type A amendments) issued by the MUMPS Development Committee (MDC) in an effort to standardise error conditions.

The **U** values are any user-defined strings that do not contain a comma. These are typically application-specific error codes managed by application-specific error handling routines that examine the value of **$ECODE** .

The **Z** values are implementation-specific error codes. For M21, they are the same values that are assigned by M21 to the **$ZERROR** special system variable.

Note that, when an error occurs which has an M error number as defined by the 1995 ANSI M standard, then this error code is appended to **$ECODE** , but the M21 **$ZERROR** equivalent is not. M21 **$ZERROR** messages are only appended when no equivalent ANSI M error code exists.

Refer elsewhere in this document for a list of the ANSI M error numbers and M21-specific error codes.

Considerations

When the value of **$ECODE** is the empty string, normal routine execution is in effect.

Error processing is initiated when:

- **$ECODE** transitions from an empty to a non-empty string. The value of **$ECODE** may change implicitly when M21 detects an error condition (such as an undefined variable), or when explicitly **SET** by the application.
- A **QUIT** returns from an execution level with **$ECODE** non-empty to a level in which no error had occurred. For example, **$STACK(new level,"ECODE")** is the empty string.
- When the value of **$ECODE** is changed via the **SET** command, the new value replaces the existing value. When this happens an error occurs.
- When a partition is initiated, **$ECODE** has the value of the empty string.

Examples

This code displays the individual error codes in **$ECODE** on individual lines:
FOR I=2:1:$L($ECODE,",")-1 WRITE !,$P($ECODE,",",I)

Chapter 4. $ESTACK

Indicates the relative execution nesting level.

Syntax

$ES{TACK}

Definition

The **$ESTACK** special variable contains a non-negative integer specifying the relative nesting of the current execution level. The value is automatically incremented by the **DO** and **XECUTE** commands, and automatically decremented by the **QUIT** command. The **NEW** command may be used to stack the current value and reset **$ESTACK** to **0** . A **QUIT** command (explicitly or implicitly at the end of a routine or an **XECUTE** string) restores the stacked value. A new partition begins with **$ESTACK** set to **0** .

Considerations

An application may stack the value of **$ESTACK** and reset it to **0** via the **NEW** command. Subsequent error handling routines may pop the execution levels until **$ESTACK** returns to **0** , at which point execution returns to its initial starting level for the application. This is useful for nested applications. If **$ESTACK** is not stacked by the **NEW** command, it always equals **$STACK** .

Examples

This command pops the current execution level as long as **$ESTACK** is positive. Note that if **$ECODE** has not been reset to the empty string, returning from an execution level with **$ECODE** not null automatically invokes error handling for the returned-to execution level:

QUIT:$ESTACK>0

This command stacks the current **$ESTACK** value; resets the current value of **$ESTACK** to zero; defines the error handling string; and invokes the application:

NEW $ESTACK SET $ETRAP="D ERROR^ROUTINE" DO APPL

Chapter 5. $ETRAP

Specifies the string to be executed when an error condition is detected.

Syntax

$ET{RAP}

Definition

The **$ETRAP** special variable contains a string of M code to be executed when an error condition is detected. The code is executed at the same execution level at which the error occurs. Prior to execution of the string in **$ETRAP** , any active **FOR** loops and indirection in the current execution level are terminated. Execution behaves as if the contents of **$ETRAP** are appended to the current routine with a temporary but unique label, and an internal **GOTO** to the appended code is performed. Updating **$ETRAP** replaces its previous value.

The current value of **$ETRAP** can be stacked using the **NEW** command. The **NEW** command does not alter the value of **$ETRAP** ; it merely stacks it. New partitions begin with **$ETRAP** set to the empty string. When **$ETRAP** is **SET** , the value of **$ZTRAP** also is reset to the empty string so that at any time, either **$ETRAP** or **$ZTRAP** defines the error handling environment, but not both.

Considerations

Within an application, the **NEW** command can be used to stack the caller's **$ETRAP** until a **QUIT** command (implicit or explicit) is executed at the current execution level.

Unlike **$ZTRAP** , the value of **$ETRAP** is not tied to an execution level unless the **NEW** command is used. Therefore, the application can set **$ETRAP** at the start of the application. The same **$ETRAP** value is used at each nested level if an error condition is detected. When initiating error processing, M21 performs an explicit **GOTO** (without changing the execution level) to the following two lines of code.

...value of $ETRAP...

QUIT:$QUIT "" QUIT

Examples

This example stacks the current value in **$ETRAP** and establishes a new string to be executed if an error condition is detected.

NEW $ETRAP SET $ETRAP="D ERR^ROUTINE"