GT.M

Programmers Guide

V7.1-004

GT.M Programmer's Guide

This document contains a description of GT.M and the operating instructions pertaining to the various functions that comprise the system. This document does not contain any commitment of FIS. FIS believes the information in this publication is accurate as of its publication date; such information is subject to change without notice. FIS is not responsible for any errors or defects.

Revision History		
Revision V7.1-004	27 June 2024	Updated the following chapters for V7.1-004: • Chapter 4: "Operating and Debugging in Direct Mode" (page 50) • Chapter 5: "General Language Features of M" (page 68) • Chapter 6: "Commands" (page 108) • Chapter 7: "Functions" (page 212) • Chapter 8: "Intrinsic Special Variables" (page 291) • Chapter 10: "Utility Routines" (page 448) • Chapter 13: "Error Processing" (page 562) • Chapter 14: "Triggers" (page 591)
Revision V7.1-003	23 November 2023	Updated the following chapters for V7.1-003: • Chapter 6: "Commands" (page 108) • Chapter 7: "Functions" (page 212) • Chapter 8: "Intrinsic Special Variables" (page 291) • Chapter 9: "Input/Output Processing" (page 340) • Chapter 10: "Utility Routines" (page 448)
Revision V7.1-002	19 September 2023	Updated the following chapters for V7.1-002: • Chapter 4: "Operating and Debugging in Direct Mode" (page 50) • Chapter 5: "General Language Features of M" (page 68) • Chapter 6: "Commands" (page 108)

		• Chapter 8: "Intrinsic Special Variables" (page 291) • Chapter 10: "Utility Routines" (page 448) • Chapter 13: "Error Processing" (page 562) • Chapter 14: "Triggers" (page 591)
Revision V7.1-001	26 June 2023	Updated the following chapters for V7.1-001: • Chapter 2: "GT.M Language Extensions" (page 5) • Chapter 3: "Development Cycle" (page 32) • Chapter 4: "Operating and Debugging in Direct Mode" (page 50) • Chapter 5: "General Language Features of M" (page 68) • Chapter 6: "Commands" (page 108) • Chapter 7: "Functions" (page 212) • Chapter 8: "Intrinsic Special Variables" (page 291) • Chapter 13: "Error Processing" (page 562)
Revision V7.1-000	04 April 2023	Updated the following chapters for V7.1-000: • Chapter 7: "Functions" (page 212)
Revision V7.0-005	02 December 2022	Updated the following chapters for V7.0-005: • Chapter 5: "General Language Features of M" (page 68) • Chapter 6: "Commands" (page 108) • Chapter 7: "Functions" (page 212)
Revision V7.0-004	20 September 2022	Updated the following chapters for V7.0-004: • Chapter 7: "Functions" (page 212)
Revision V7.0-003	24 June 2022	Updated the following chapters for V7.0-003: • Chapter 5: "General Language Features of M" (page 68) • Chapter 6: "Commands" (page 108) • Chapter 7: "Functions" (page 212) • Chapter 9: "Input/Output Processing" (page 340)
Revision V7.0-002	23 March 2022	Updated the following chapters for V7.0-002: • Chapter 5: "General Language Features of M" (page 68) • Chapter 6: "Commands" (page 108)

		• Chapter 7: "Functions" (page 212)
		• Chapter 8: "Intrinsic Special Variables" (page 291)
		• Chapter 9: "Input/Output Processing" (page 340)
		• Chapter 10: "Utility Routines" (page 448)
		• Chapter 13: "Error Processing" (page 562)
Revision V7.0-001	24 November 2021	Updated the following chapters for V7.0-001: • Chapter 2: "GT.M Language Extensions" (page 5) • Chapter 5: "General Language Features of M" (page 68) • Chapter 6: "Commands" (page 108) • Chapter 7: "Functions" (page 212) • Chapter 8: "Intrinsic Special Variables" (page 291) • Chapter 9: "Input/Output Processing" (page 340) • Chapter 10: "Utility Routines" (page 448)
Revision V7.0-000	12 February 2021	Updated the following chapters for V7.0-000: • Chapter 10: "Utility Routines" (page 448)
Revision V6.3-014	06 October 2020	Updated the following chapters for V6.3-014: • Chapter 3: "Development Cycle" (page 32) • Chapter 6: "Commands" (page 108) • Chapter 11: "Integrating External Routines" (page 521)
Revision V6.3-013	30 June 2020	Updated the following chapters for V6.3-013: • Chapter 6: "Commands" (page 108)
Revision V6.3-012	08 April 2020	Updated the following chapters for V6.3-012: • Chapter 7: "Functions" (page 212) • Chapter 8: "Intrinsic Special Variables" (page 291) • Chapter 9: "Input/Output Processing" (page 340) • Chapter 10: "Utility Routines" (page 448) • Chapter 11: "Integrating External Routines" (page 521)
Revision V6.3-011	20 December 2019	Updated the following chapters for V6.3-011:

		• Chapter 6: "Commands" (page 108) • Chapter 9: "Input/Output Processing" (page 340) • Chapter 10: "Utility Routines" (page 448) • Chapter 12: "Internationalization" (page 543)
Revision V6.3-010	31 October 2019	Updated the following chapters for V6.3-010: • Chapter 6: "Commands" (page 108) • Chapter 7: "Functions" (page 212) • Chapter 12: "Internationalization" (page 543)
Revision V6.3-009	27 June 2019	Updated the following chapters for V6.3-009: • Chapter 8: "Intrinsic Special Variables" (page 291) • Chapter 10: "Utility Routines" (page 448) • Chapter 11: "Integrating External Routines" (page 521)
Revision V6.3-008	24 April 2019	Updated the following chapters for V6.3-008: • Chapter 2: "GT.M Language Extensions" (page 5) • Chapter 6: "Commands" (page 108) • Chapter 8: "Intrinsic Special Variables" (page 291) • Chapter 10: "Utility Routines" (page 448)
Revision V6.3-007	04 February 2019	Updated the following chapters for V6.3-007: • Chapter 6: "Commands" (page 108) • Chapter 4: "Operating and Debugging in Direct Mode" (page 50) • Chapter 13: "Error Processing" (page 562) • Chapter 7: "Functions" (page 212) • Chapter 5: "General Language Features of M" (page 68) • Chapter 8: "Intrinsic Special Variables" (page 291) • Chapter 9: "Input/Output Processing" (page 340) • Chapter 14: "Triggers" (page 591) • Chapter 10: "Utility Routines" (page 448)
Revision V6.3-006	26 October 2018	Updated the following chapters for V6.3-006:

Revision V6.3-005	29 June 2018	Updated the following chapters for V6.3-005:
Revision V6.3-004	23 March 2018	Updated the following chapters for V6.3-004:

		• Chapter 10: "Utility Routines" (page 448)
Revision V6.3-003	12 December 2017	Updated the following chapters for V6.3-003: • Chapter 1: "About GT.M" (page 1) • Chapter 6: "Commands" (page 108) • Chapter 4: "Operating and Debugging in Direct Mode" (page 50) • Chapter 13: "Error Processing" (page 562) • Chapter 7: "Functions" (page 212) • Chapter 5: "General Language Features of M" (page 68) • Chapter 8: "Intrinsic Special Variables" (page 291) • Chapter 9: "Input/Output Processing" (page 340) • Chapter 14: "Triggers" (page 591)
Revision V6.3-002	22 August 2017	Updated the following chapters for V6.3-002: • Chapter 1: "About GT.M" (page 1) • Chapter 6: "Commands" (page 108) • Chapter 5: "General Language Features of M" (page 68) • Chapter 8: "Intrinsic Special Variables" (page 291) • Chapter 10: "Utility Routines" (page 448)
Revision V6.3-001	20 March 2017	Updated the following chapters for V6.3-001: • Chapter 6: "Commands" (page 108) • Chapter 13: "Error Processing" (page 562) • Chapter 7: "Functions" (page 212) • Chapter 5: "General Language Features of M" (page 68) • Chapter 2: "GT.M Language Extensions" (page 5) • Chapter 12: "Internationalization" (page 543) • Chapter 8: "Intrinsic Special Variables" (page 291) • Chapter 9: "Input/Output Processing" (page 340) • Chapter 3: "Development Cycle" (page 32) • Chapter 14: "Triggers" (page 591)

		• Chapter 11: "Integrating External Routines" (page 521)
		• Chapter 10: "Utility Routines" (page 448)
Revision V6.1-000	28 August 2014	Updated for V6.1-000. For chapter-specific revisions, refer to Chapter 2: "GT.M Language Extensions" (page 5), Chapter 6: "Commands" (page 108), Chapter 7: "Functions" (page 212), Chapter 8: "Intrinsic Special Variables" (page 291), Chapter 9: "Input/Output Processing" (page 340), and Chapter 11: "Integrating External Routines" (page 521).
Revision V6.0-003	24 February 2014	Updated for V6.0-002 and V6.0-003. For chapter-specific revisions, refer to Chapter 3, Development Cycle [32], Chapter 6: "Commands" (page 108), Chapter 7: "Functions" (page 212), Chapter 8: "Intrinsic Special Variables" (page 291), Chapter 9: "Input/Output Processing" (page 340), Chapter 11: "Integrating External Routines" (page 521) and Chapter 13: "Error Processing" (page 562).
Revision V6.0-001	21 March 2013	Updated for V6.0-001. For chapter-specific revisions, refer to Chapter 3, Development Cycle [32], Chapter 5: "General Language Features of M" (page 68), Chapter 6: "Commands" (page 108), Chapter 7: "Functions" (page 212), Chapter 9: "Input/Output Processing" (page 340), Chapter 10: "Utility Routines" (page 448), Chapter 11: "Integrating External Routines" (page 521), and Chapter 13: "Error Processing" (page 562).
Revision V6.0-000	19 November 2012	Updated for V6.0-000. For chapter-specific revisions, refer to Chapter 6: "Commands" (page 108) and Chapter 9: "Input/Output Processing" (page 340).

Table of Contents

About This Manual

The GT.M Programmer's Guide describes how to develop and maintain applications using GT.M. For information on how to install the GT.M software and maintain the user environment, refer to the **GT.M Administration and Operations Guide**.

Intended Audience

This manual is intended for programmers who develop and/or maintain M applications in the GT.M environment. This manual assumes that the programmers have no previous knowledge of GT.M. However, it does assume that the programmers have access to the UNIX documentation that supplements the limited UNIX information in this manual.

Purpose of the Manual

The GT.M Programmer's Guide documents all aspects of programming with M in the GT.M environment.

How to Use This Manual

To assist you in locating information, the flow of the chapters moves from more general usage information to more specific reference and utilization information.

Chapter 1: "About GT.M" (page 1) gives an overview of the features of the GT.M programming system.

Chapter 2: "GT.M Language Extensions" (page 5) provides summary tables of the GT.M language extension, grouped by purpose.

Chapter 3: "Development Cycle" (page 32) gives an introduction to program development in the GT.M environment.

Chapter 4: "Operating and Debugging in Direct Mode" (page 50) describes basic elements of operating in Direct Mode and the features available for debugging in Direct Mode.

Chapter 5: "General Language Features of M" (page 68) describes features of M as a programming language that are important in using the reference information given in the "Commands" [108], "Functions" [212], and "Intrinsic Special Variables" [291] chapters.

Chapter 6: "Commands" (page 108) is a comprehensive description of each GT.M command. Entries are in alphabetical order.

Chapter 7: "Functions" (page 212) is a comprehensive description of each GT.M function. Entries are in alphabetical order.

Chapter 8: "Intrinsic Special Variables" (page 291) is a comprehensive description of each GT.M intrinsic special variable. Entries are in alphabetical order.

Chapter 9: "Input/Output Processing" (page 340) describes input/output facilities available in GT.M.

Chapter 10: "Utility Routines" (page 448) describes library utilities provided with GT.M for performing frequently used M tasks.

Chapter 11: "Integrating External Routines" (page 521) describes how to call GT.M routines from routines created in other programming languages and how to call out of GT.M routines to these external programs.

Chapter 12: "Internationalization" (page 543) describes the facilities available for using GT.M successfully with languages other than American English.

Chapter 13: "Error Processing" (page 562) describes methods for error handling in GT.M.

Chapter 14: "Triggers" (page 591) describes the trigger facility available with GT.M.

Chapter 1. About GT.M

Revision History		
Revision V6.3-005	29 June 2018	• In "Input-Output Processing" (page 3), remove reference to magnetic tape.
Revision V6.3-003	12 December 2017	• In "Managing Source Code" (page 2), add explanation for the handling of the source line limit.
Revision V6.3-002	22 August 2017	• In "Managing Source Code" (page 2), add source line max length statement

GT.M runs on a wide variety of computer platforms. Consult FIS for the current list of Supported platforms.

In addition to preserving the traditional features of M, GT.M also offers an optimized compiler that produces object code that does not require internal interpreters during execution.

On all platforms, the GT.M dynamic linking mechanism activates compiled objects. On some platforms, you can link the object modules into shared object libraries.

In keeping with the focus on creating fully compiled code, GT.M is tightly integrated with the operating system environment and permits the use of operating system utilities for program development.

GT.M also provides a full complement of M tools for creating, compiling, and debugging source code. Many of these tasks are accomplished from the GT.M facility called Direct Mode, which offers the look and feel of an interpreted language that is familiar to the traditional M programmer.

Programming Environment

The GT.M Programming Environment is described in the following sections.

Managing Data

The scope of M data is either process local or global.

- Local variables last only for the duration of the current session; GT.M deletes them when the M process terminates.

- Global variables contain data that persists beyond the process. GT.M stores global variables on disk. A Global Directory organizes global variables and describes the organization of a database. The GT.M administrator uses the Global Directory Editor (GDE) to create and manage Global Directories. A Global Directory maps global names to a database file. GT.M uses this mapping when it stores and retrieves globals from the database. Several Global Directories may refer to a single database file.

For more information about the GT.M data management system, refer to the "Global Directory Editor", "MUPIP" and "GT.M Journaling" chapters in the **GT.M Administration and Operations Guide**.

Database Management Utilities

The Global Directory Editor (GDE) creates, modifies, maintains, and displays the characteristics of Global Directories. GDE also maps LOCKs on resource names to the region of the database specified for the corresponding global variables.

The M Peripheral Interchange Program (MUPIP) creates database files and provides tools for GT.M database management and database journaling.

For more information regarding GT.M database utilities, refer to the "Global Directory Editor", "MUPIP" and "GT.M Journaling" chapters in the **GT.M Administration and Operations Guide**.

Managing Source Code

In the GT.M programming environment, source routines are generated and stored as standard UNIX files. They are created and edited with standard UNIX text editors. GT.M accepts source lines of up to 8192 bytes. When GT.M encounters a line with a length greater than 8192 bytes in a source file, it emits an LSEXPECTED warning. This warning identifies cases where a line greater than 8192 bytes is split into multiple lines, which causes statements beyond the character prior to the limit to execute irrespective of any starting IF, ELSE or FOR commands. The 8192 byte limit applies to XECUTE command arguments and Direct Mode input as well.

GT.M is designed to work with the operating system utilities and enhances them when beneficial. The following sections describe the process of programming and debugging with GT.M and from the operating system.

Source File Management

In addition to standard M "percent" utilities, GT.M permits the use of the standard UNIX file manipulation tools, for example, the diff, grep, cp, and mv commands. The GT.M programmer can also use the powerful facilities provided by the UNIX directory structure, such as time and date information, tree-structured directories, and file protection codes.

GT.M programs are compatible with most source management software, for example, RCS and SCCS.

Programming and Debugging Facilities

The GT.M programmer can use any UNIX text editor to create M source files. If you generate a program from within the Direct Mode, it also accesses the UNIX text editor specified by the environment variable EDITOR and provides additional capabilities to automate and enhance the process.

The GT.M programmer also uses the Direct Mode facility to interactively debug, modify, and execute M routines. In Direct Mode, GT.M executes each M command immediately, as if it had been in-line at the point where GT.M initiated Direct Mode.

The following is a list of additional enhancements available from the Direct Mode:

- The capability to issue commands from Direct Mode to the shell

- A command recall function to display and reuse previously entered commands

- Many language extensions that specifically optimize the debugging environment

TheGT.M Compiler

The GT.M compiler operates on source files to produce object files consisting of position-independent, native object code, which on some platforms can be linked into shared object libraries. GT.M provides syntax error checking at compile-time and

allows you to enable or disable the compile-as-written mode. By default, GT.M produces an object file even if the compiler detects errors in the source code. This compile-as-written mode facilitates a flexible approach to debugging.

The Run-Time System

A GT.M programmer can execute an M routine from the shell or interactively, using the M commands from Direct Mode.

The run-time system executes compile-as-written code as long as it does not encounter the compile-time errors. If it detects an error, the run-time system suspends execution of a routine immediately and transfers control to Direct Mode or to a user-written error routine.

Automatic and Incremental Linking

The run-time system utilizes a GT.M facility called ZLINK to link in an M routine. When a program or a Direct Mode command refers to an M routine that is not part of the current process, GT.M automatically uses the ZLINK facility and attempts to link the referenced routine (auto-ZLINK). The ZLINK facility also determines whether recompilation of the routine is necessary. When compiling as a result of a ZLINK, GT.M typically ignores errors in the source code.

The run-time system also provides incremental linking. The ZLINK command adds an M routine to the current image. This feature facilitates the addition of code modifications during a debugging session. The GT.M programmer can also use the feature to add patches and generated code to a running M process.

Error Processing

The GT.M compiler detects and reports syntax errors at the following times:

- Compile-time - while producing the object module from a source file

- Run-time - while compiling code for M indirection and XECUTEs

- Run-time - when the user is working in Direct Mode.

The compile-time error message format displays the line containing the error and the location of the error on the line. The error message also indicates what was incorrect about the M statement.

GT.M can not detect certain types of errors associated with indirection, the functioning of I/O devices, and program logic until run-time.

The compile-as-written feature allows compilation to continue and produces an object module despite errors in the code. This permits testing of other pathways through the code. The errors are reported at run-time, when GT.M encounters them in the execution path.

The GT.M run-time system recognizes execution errors and reports them when they occur. It also reports errors flagged by the compiler when they occur in the execution path.

For more information, see Chapter 13: "Error Processing" (page 562).

Input-Output Processing

GT.M supports input and output processing with the following system components:

- Terminals

- Sequential disk files

- Mailboxes

- FIFOs

- Null devices

- Socket devices

GT.M input/output processing is device-independent. Copying information from one device to another is accomplished without reformatting.

GT.M has special terminal-handling facilities. GT.M performs combined QIO operations to enhance terminal performance. The terminal control facilities that GT.M provides include escape sequences, control character traps, and echo suppression.

GT.M supports RMS sequential disk files that are accessed using a variety of deviceparameters.

GT.M supports block I/O with fixed and variable length records for file-structured (FILES-11) tapes and non-file-structured unlabeled (FOREIGN) tapes. GT.M supports the ASCII character set for unlabeled FOREIGN and FILES-11 tapes. GT.M supports the EBCDIC character set for FOREIGN tapes only. GT.M also supports FOREIGN DOS-11 and ANSI labelled tapes or stream format records. It also supports ASCII and EBCDIC character sets.

GT.M uses permanent or temporary mailboxes fifos for interprocess communication. GT.M treats mailboxes as record-structured I/O devices.

GT.M provides the ability to direct output to a null device. This is an efficient way to discard unwanted output.

GT.M provides device-exception processing so that I/O exception handling need not be combined with process-related exception conditions. The OPEN, USE, and CLOSE EXCEPTION parameters define an XECUTE string as an error handler for an I/O device.

Integrating GT.M with Other Languages

GT.M offers capabilities that allow you to optimize your programming environment. These include allowing you to call into M routines from programs written in other programming languages, access your M databases with interfaces that provide functionality equivalent to M intrinsic database functions, and to alter your programming environment when working with languages other than American English. These include allowing you to call programs written in other programming languages that support C-like interfaces and to alter your programming environment when working with languages other than American English. This capability is described in more detail in chapters throughout this manual.

Access to Non-M Routines

GT.M routines can call external (non-M) routines using the external call function. This permits access to functions implemented in other programming languages. For more information, see Chapter 11: "Integrating External Routines" (page 521).

Internationalization

GT.M allows the definition of alternative collation sequences and pattern matching codes for use with languages other than English. Chapter 12: "Internationalization" (page 543) describes the details and requirements of this functionality.

Chapter 2. GT.M Language Extensions

Revision History		
Revision V7.1-001	26 June 2023	• In "Debugging Facilities" (page 7), add ZLINK and auto-ZLINK to the $ZCstatus table entry • In "Exception Handling Facilities" (page 8), add ZLINK and auto-ZLINK to the $ZCSTATUS table entry
Revision V7.0-001	24 November 2021	• In "Operating System Interface Facilities" (page 6), add note on <NUL> characters
Revision V6.3-008	24 April 2019	• In "ICU" (page 28), fix minor typo.
Revision V6.3-006	26 October 2018	• In "Discussion and Best Practices" (page 29), UTF-8 mode tweaks. • In "ICU" (page 28), UTF-8 mode tweaks. • In "Extensions for the support for the Unicode® standard " (page 21), should points -> should point • In "Philosophy of GT.M's support for the Unicode® standard" (page 25), minor corrections to use UTF-8 except when discussing the Unicode standard or multiple encoding (beyond UTF-8).
Revision V6.3-001	20 March 2017	• In "ICU" (page 28), removed reference to instructions on building ICU. • In "Extensions for the support for the Unicode® standard " (page 21), corrected $ZPIECE() function abbreviation
Revision V6.2-001	27 February 2015	In "Journaling Extensions" (page 9), replaced descriptions of the deprecated ZTSTART and ZTCOMMIT with a discussion of how TSTART and TCOMMIT interact with Journaling.
Revision V6.1-000	28 August 2014	• In "Alias Variables Extensions" (page 11), added the "SET * and QUIT * Examples" (page 15) section. • In "Extensions for the support for the Unicode® standard " (page 21), improved the description on GT.M's use of Unicode Byte Order Market (BOM).

In addition to providing all of the ANSI standard M features, GT.M offers a number of language extensions. In this chapter, the language extensions are grouped by intended function to demonstrate their relationships to each other and to the programming

process. A summary table is provided in each section. For a full description of a particular extension, refer to its complete entry in the "Commands" [108], "Functions" [212], or "Intrinsic Special Variables" [291] chapter.

The following sections describe the GT.M language extensions listed below:

- UNIX interface facilities

- Debugging tools

- Exception-handling extensions

- Journaling extensions

- Extensions providing additional capability

- Device Handling Extensions

- Alias Variables Extensions

- Extensions for Unicode Support

Operating System Interface Facilities

To improve efficiency and reduce duplication and inconsistency, GT.M is closely integrated with the host operating system environment. With GT.M you can gain access to the operating system facilities to examine:

- System information, such as quotas and SIDs

- Jobs and processes

- Directories and files

- Devices

- Messages

- Privileges

The following table summarizes the GT.M operating system interface facilities.

Operating System Interface Facilities	
EXTENSION	**EXPLANATION**
ZSYstem	Provides access to the shell.
$ZMessage()	Translates an error condition code into text form.
$ZCMdline	Contains a string value specifying the "excess" portion of the command line that invoked the GT.M process.
$ZJob	Holds the pid of the process created by the last JOB command performed by the current process.
$ZPARSE()	Parses a UNIX filename.

Operating System Interface Facilities	
EXTENSION	**EXPLANATION**
$ZSEARCH()	Searches for one or more UNIX files.
$ZSYstem	Contains the status code of the last ZSYSTEM.
$ZTRNLNM()	Translates an environment variable.
$ZDIRectory	Contains current working directory.

> **NOTE:**
>
> The OS services accessed by GT.M commonly treat a <NUL> character as a terminator therefore any in cases where application information passes to the OS as arguments an embedded <NUL> may cause non-obvious behavior.

Debugging Facilities

GT.M provides a number of debugging features. These features include the ability to:

- Interactively execute routines using M commands.

- Display lines that may contain errors using the ZPRINT command and the $ZPOSITION special variable.

- Redisplay error messages using the $ZSTATUS special variable and the ZMESSAGE command.

- Set breakpoints and actions to bypass an error using the ZBREAK command.

- Execute a line at a time using the ZSTEP command.

- Display information about the M environment using the ZSHOW command.

- Modify the invocation stack with QUIT and ZGOTO.

- Incrementally add or modify code using the ZLINK and ZEDIT commands.

- Continue execution using the ZCONTINUE command.

- Establish "watch points" with triggers to trap incorrect accesses on global variable updates.

The following table summarizes the GT.M language extensions that facilitate debugging.

GT.M Debugging Tools	
EXTENSION	**EXPLANATION**
ZBreak	Establishes a temporary breakpoint, with optional M action and/or activation count.
ZContinue	Continues routine execution from a break.
ZEDit	Invokes the UNIX text editor specified by the EDITOR environment variable.
ZGoto	Removes multiple levels from the M invocation stack and transfers control.

GT.M Debugging Tools	
EXTENSION	**EXPLANATION**
ZLink	Includes a new or modified M routine in the current M image; automatically recompiles if necessary.
ZMessage	Signals the specified condition.
ZPrint	Displays lines of source code.
ZSHow	Displays information about the M environment.
ZSTep	Incrementally executes a routine to the beginning of the next line of the same type.
ZWRite	Displays all or some local or global variables.
$ZCSTATUS	Holds the value of the status code for the last compile performed by a ZCOMPILE, ZLINK or auto-ZLINK.
$ZEDit	Contains the status code for the last ZEDit.
$ZJOBEXAM()	Performs a ZSHOW "*" to a default file location and name, or the one optionally specified by the argument.
$ZLEVel	Contains the current level of DO/XECUTE nesting.
$ZMessage()	Translates an error condition code into text form.
$ZPOSition	Contains a string indicating the current execution location.
$ZPROmpt	Controls the symbol displayed as the direct mode prompt.
$ZROutines	Contains a string specifying a directory list containing the object, and optionally the source, files.
$ZSOurce	Contains name of the M source program most recently ZLINKed or ZEDITed; default name for next ZEDIT or ZLINK.
$ZStatus	Contains error condition code and location of the last exception condition occurring during routine execution.
$ZSTep	Controls the default ZSTep action.

Exception Handling Facilities

The GT.M exception trapping allows you to do the following:

- DO a recovery routine and resume the original command stream.

- GOTO any special handling; an extended ZGOTO provides for context management.

- Report an error and enter Direct Mode for debugging.

- OPEN Input/Output devices with specific traps in addition to the main trap.

- Trap and process an exception based on a device error.

- Trap and process an exception based on terminal input.

The following table summarizes the GT.M language extensions that facilitate exception handling.

GT.M Exception Handling Extensions	
EXTENSION	**EXPLANATION**
ZGoto	Removes zero or more levels from the M Invocation stack and, optionally, transfers control.
ZMessage	Signals the specified condition.
$ZCSTATUS	Holds the value of the status code for the last compile performed by a ZCOMPILE, ZLINK or auto-ZLINK.
$ZEOF	Contains indication of whether the last READ reached end-of-file.
$ZMessage()	Translates an error condition code into text form.
$ZLevel	Contains current level of DO/XECUTE nesting.
$ZStatus	Contains error condition code and location of last exception condition occurring during routine execution.
$ZSYstem	Contains the status code of the last ZSYSTEM.
$ZTrap	Contains an XECUTE string or entryref that GT.M invokes upon encountering an exception condition.
EXCEPTION	Provides a deviceparameter specifying an XECUTE string or entryref that GT.M invokes upon encountering a device-related exception condition.

Journaling Extensions

Journaling records redundant copies of database update information to increase protection against loss of information due to hardware and software failure. In GT.M, TSTART and TCOMMIT mark the beginning and end of an application (logical) transaction, which may consist of multiple global variable updates. When a TCOMMIT takes $TLEVEL from one (1) to zero (0), it transfer all of the transaction updates to the journal file, and, except if TRANSACTIONID="BATCH", returns control to the application only after the associated records reach the secondary storage holding the journal file.

The following table summarizes the GT.M language extensions for journaling.

Journaling Extensions	
EXTENSION	**EXPLANATION**
View	Extended to ensure that GT.M has transferred all updates to the journal file.
$View()	Extended for examining journaling status.

Extensions ForAdditional Capability

For ways to adjust some process operating characteristics, see the command description "View" (page 151). For ways to get information about certain process operating characteristics, see the function description "$View()" (page 243).

In GT.M, support of environment specification for global names and resource names is possible. It is possible to excercise user code to customize interpretation of the environment specification. See Chapter 5: "General Language Features of M" (page 68) for details.

The following table summarizes GT.M extensions that increase general capability.

GT.M Extensions for Additional Capability	
EXTENSION	EXPLANATION
View	Modifies the environment.
ZAllocate*	Facilitates incremental locking by locking a name without unlocking previously locked names.
ZDeallocate*	Unlocks one or more names without necessarily unlocking other names.
ZHelp	Provides access to on-line help.
ZWIthdraw	"Kills" data in a node without affecting the node's descendants.
$Order()	Enhanced to return the next unsubscripted variable in collating sequence from the current environment. Name-level $ORDER() always returns an empty string when used with extended references.
$View()	Examines the GT.M environment.
$ZCStatus	Returns the status from the last compile.
$ZDate()	Converts a date and/or time in $HOROLOG format into formatted text, using a user-specified format string.
$ZPrevious()**	Returns the previous element in a collating sequence, at the current level of a local or global array.
$ZA,$ZB, $ZEOF	Return device dependent I/O status information.
$ZCOmpile	Maintains the compiler qualifiers to be used on automatic compilation.
$ZBIT functions	A series of functions beginning with the characters $ZBIT that allow manipulation of bits.
$ZGBLdir	Maintains the name of the current global directory; may be set to switch this process to a new database.
$ZIO	Contains translated name of current I/O device.
$ZINTerrupt	Specifies the code to be XECUTE'd when an interrupt is processed.
$ZKEY	SD: Returns current position in the sequential file based on last read.
$ZMAXTPTIme	Contains an integer value indicating the time duration GT.M should wait for the completion of all activities fenced by the current transaction's outermost TSTART/TCOMMIT pair.
$ZROutines	Maintains the list of directories to search during look-ups of object and source files.
$ZSYstem	Returns the status code for the last subprocess invoked with the ZSYSTEM command.
$ZVERsion	Contains a designation of the current version name, level, and operating system.

*The ZALLOCATE and ZDEALLOCATE commands are provided for compatibility with other M systems. However, FIS recommends use of the standard LOCK command, which provides an incremental locking facility. The incremental lock provides both flexibility and greater compatibility with the M language standard.

**The $ZPREVIOUS function is provided for compatibility with previous versions of GT.M and other M systems. However, FIS recommends use of the standard two-argument form for the $ORDER function.

GT.M Device Handling Extensions

In the earlier versions of the M standard, device behavior was defined as a framework, with the details left to the implementations. GT.M supports Terminals, Sequential Disks, FIFOs, PIPEs and a Null device under this model. Subsequently device mnemonicspaces were added to the standard and some of them defined. GT.M supports the SOCKET device under this model with some extensions identified with controlmnemonics starting with the letter "Z."

For details of GT.M device handling see Chapter 9: "Input/Output Processing" (page 340).

Alias Variables Extensions

Alias variables provide a layer of abstraction between the name of a local variable and an array analogous to that provided by M pass by reference in routines and function calls. Multiple local variables can be aliased to the same array, and a SET or KILL to one acts as a SET or KILL to all. Alias container variables provide a way using a subscripted local to store a reference to an entire local variable array, which protects the associated array even when it's not accessible through any current local variable name.

GT.M aliases provide low level facilities on which an application can implement object-oriented techniques. An object can be mapped onto, and stored and manipulated in an array, then saved in an alias container variable whence it can be retrieved for processing. The use of appropriate subscripts in the array used for a container, provides a way to organize the stored objects and retrieve them by using the $ORDER() function to traverse the container array. The use of alias variables to implement objects provides significant efficiencies over traditional local variables because alias variables and alias container variables eliminate the need to execute MERGE commands to move objects.

Example:

```
GTM>kill A,B

GTM>set A=1,*B=A ; B & A are aliases
GTM>write B
1
GTM>
```

Within the context of Alias Variables extensions:

1. array is very similar to its definition in the M standard, and means an entire tree of nodes, including the root and all descendants, except that it only applies to local variables and not to global variables.

2. "Associated alias variables" means all alias variables and all alias container variables associated with an array.

3. lvn is very similar to its definition in the M standard except that in the context of alias variables lvn is used to refer to a local variable name with a subscript.

4. lname is very similar to its definition in the M standard, except that in the context of alias variables, lname is just the name of an unsubscripted local variable (root of an array).

5. "Data cell" and "node" are synonyms.

The following table summarizes Alias Variables extensions.

GT.M Extensions for Alias Variables	
EXTENSION	**EXPLANATION**
Set *	Explicitly creates an alias. For more information, refer to the description of SET * in "Set" (page 144)
Kill *	Removes the association between its arguments, and any associated data cells. For more information, refer to the description of KILL * in "Kill" (page 129)
Quit *	When QUIT * terminates an extrinsic function or an extrinsic special variable, it always returns an alias container. For more information, refer to the description of QUIT * in "Quit" (page 142).
ZWrite / ZSHow "V"	Produces Alias Variables format output. For more information, refer to "ZWRITE Format for Alias Variables" (page 210)
New	For the scope of the NEW, a NEW of a name suspends its alias association. For more information, refer to "New" (page 138).
Exclusive New	Create a scope in which some associations between an lname or an lvn and an array may be invisible. For more information, refer to "New" (page 138).
$ZAHandle()	returns a unique identifier (handle) for the array associated with an lname or an alias container; for an subscripted lvn that is not an alias container, it returns an empty string. For more information, refer to "$ZAHandle()" (page 249)
$ZDATA()	Extends $DATA() to reflect the current alias state of the lvn or lname argument in order to identify alias and alias container variables. For more information, refer to "$ZDATA()" (page 261).
View and $View()	• VIEW provides LV_GCOL, LV_REHASH, and STP_GCOL to perform garbage collection and local variable lookup table reorganization operations which normally happen automatically at appropriate times. For more information on the keywords of the VIEW command, refer to "Key Words in VIEW Command" (page 152). • $VIEW() provides LV_CREF, LV_GCOL, and LV_REF. FIS uses the LC_CREF, LV_GCOL, LV_REF keywords in testing and is documenting them to ensure completeness in product documentation. They may (or may not) be useful during application development for debugging or performance testing implementation alternatives. For more information the keywords of $VIEW(), refer to "Argument Keywords of $VIEW()" (page 244).
TSTART, RESTART, and ROLLBACK	TSTART command can optionally list names whose arrays are restored on a transaction RESTART. If any of these are alias variables or have nodes which are alias container variables, their associations are also restored on transaction RESTART. For more information, refer to Chapter 6: "Commands" (page 108).

Definitions

Alias Variables

Alias Variables provide access to an array through multiple names. Conceptually an alias variable is the same as a pass-by-reference joining of multiple variable names, except that the joining of alias variables is explicit, whereas that of variables

passed by reference is implicit. Indeed, the underlying implementation of alias variables and pass-by-reference within GT.M is the same.

- All alias variables associated with the same array are equivalent in their access to its nodes - for example, a SET of a node in an array via one name is no different than a SET to that node using any other name of which it is an alias. Nothing about the order of their creation or association has any significance.

- Once an array becomes accessible via only a single unsubscripted name, GT.M treats that name as a traditional local variable.

- GT.M treats variables joined through pass-by-reference as a special variant of an alias variable. Pass-by-reference relates to the M stack model with aliasing implicit as a side effect of invocation with DO or $$ and unaliasing implicit as a side effect of QUIT. In the broader alias case, program commands directly alias and unalias names without any binding to the M stack.

- GT.M treats the state of a TP (Transaction Processing) RESTART variable as an internal alias, which it only exposes if the transaction creating it RESTARTs.

- GT.M treats variables hidden by exclusive NEW as a type of alias.

- Owing to their implicit behavior, under certain circumstances, pass-by-reference aliases, RESTART variable and exclusive NEW aliases are not entirely symmetrical with respect to explicitly created alias variables (that is, they may come and go at different times, whereas alias variables come and go under application program control).

Alias Container Variables

Alias container variables are subscripted lvns that protect arrays for subsequent access by an alias variable. Since accessing an array requires a name, aliasing a name with the alias container regains access to an array stored in a container. For example:

```
GTM>kill A,B,C

GTM>set A=1,*C(2)=A ; C(2) is a container
GTM>zwrite
A=1 ;*
*C(2)=A
GTM>set *B=C(2) ; B is now an alias
GTM>write B,":",$length(C(2)),":" ; An alias variable provides access but a container doesn't
1:0:
GTM>
```

- The value of an alias container is the empty string.

- Use the SET * command to associate an lname with the container to obtain an alias that provides access to the array in a container.

- SET with an alias container as left-hand side target replaces the value at that node of the container variable and destroys any prior alias association with an array.

- References to descendants of an alias container variable refer to nodes of the named parent array and have no relationship to any alias container held by a parent node.

- An alias container variable serves as a way to organize and manage entire arrays.

- While it takes two alias variables for an array to be considered aliased, it only takes one alias container variable to do so.

Performance

With two exceptions, alias and alias container variables add no overhead to normal local variable performance:

1. Complex patterns of aliases layered onto TSTART RESTART variables.

2. Complex patterns of aliases intermixed with NEW scope management, particularly when using exclusive NEW.

There is no reason to avoid aliases in any situation, but in those two contexts, GT.M rewards attention to tidy design. GT.M uses garbage collection to manage the storage used for local variables. Increasing the use of local variables, for example, to implement objects, will increase the need for garbage collection, even though the garbage collector and storage management are designed to be light weight and self-tuning. The use of alias variables to implement objects, however, is as efficient as any other method is likely to be, and except for the normal admonition to not keep arrays and local variables around when they are not needed, and to not create levels of contexts over and above those actually needed by application logic, use alias variables as liberally as your application needs dictate.

ZWRITE / ZSHOW "V" format

ZWRITE as applied to local variables and ZSHOW "V" are conceptually similar, with two differences:

• ZWRITE allows the use of patterns to specify variables and subscripts to display whereas ZSHOW "V" applies to all local variables.

• ZSHOW "V" optionally allows the output to be directed to a global or local variable, whereas ZWRITE always directs its output to the current output device.

For more information on the ZWRITE / ZSHOW "V" format for alias variables, refer to "ZWRITE Format for Alias Variables" (page 210).

Pass-by-reference

GT.M's underlying implementation of pass-by-reference and alias variables is the same. As illustrated by the program killalias above, ZWRITE displays variables joined though pass-by-reference using alias conventions. Pass-by-reference is distinguished from alias variables by its implicit creation and elimination. Note the interaction between pass by reference and alias variables when the association of a formallist parameter in a subprogram is changed:

```
$ /usr/lib/fis-gtm/V5.4-002B/gtm -run ^switchalias
switchalias ; Demonstrate Set * on formalist parameter
  zprint ; Print this program
  set A=1,B=2
  write "------------",!
  write "Initial Values:",!
  zwrite
  do S1(.A)
  write "------------",!
  write "On return:",!
  zwrite
  quit
  ;
S1(X) ;
  set X=3
  write "------------",!
  write "Inside call - note alias association for formallist parameter:",!
```

```
  zwrite
  set *X=B,X=4 ; Change association of formallist parameter
  write "------------",!
  write "Note changed association",!
  zwrite
  quit
------------
Initial Values:
A=1
B=2
------------
Inside call - note alias association for formallist parameter:
A=3 ;*
B=2
*X=A
------------
Note changed association
A=3
B=4 ;*
*X=B
------------
On return:
A=3
B=4
$
```

SET * and QUIT * Examples

The following table show the type of data movement of alias and alias container variables from QUIT * in a function to a SET * target:

	QUIT *	SET *	Result	ZWRITE
set *a=$$makealias(.c)	Creates an alias container	Dereferences the alias container	Same as **set *a=c**	***c=a**
set *a(1)=$ $makealias(.c)	Creates an alias container	Dereferences the alias container	Same as **set *a(1)=c**	***a(1)=c**
set *a=$$makecntnr(.c)	Returns an alias container	Copies the alias container	Same as **set *a=c(1)**	***c=a**
set *a(1)=$ $makecntnr(.c)	Returns an alias container	Copies the alias container	Same as **set *a(1)=c(1)**	***a(1)=c**

The makealias function returns an alias of the argument:

```
makealias(var)
 quit *var
```

The makecntr function returns an alias container of the argument:

```
makecntnr(var)
 new cont
 set *cont(1)=var
 quit *cont(1)
```

KILL * Examples

Example

```
GTM>Set A=1,*B=A ; Create an array and an association

GTM>ZWRite ; Show that the array and association exist
A=1 ;*
*B=A
GTM>Kill *A ; Remove the association for A - it now has no association and no array
GTM>ZWRite ; B is a traditional local variable
B=1
Example:
GTM>Set A=2 ; add a value for A

GTM>ZWRite ; A and B have different values and both are traditional local variables
A=2
B=1
GTM>
```

KILL on the other hand, removes data in the array (and possibly the array itself) without affecting any alias association.

```
GTM>Set A=2,*B=A ; Create an array and an association

GTM>ZWRite ; Both array and association exist
A=2 ;*
*B=A
GTM>Kill A ; Kill the array
GTM>ZWRite ; There's no data to show - only the association
*B=A
GTM>Set B=3 ; Create a new value
GTM>ZWRite ; The association was unaffected by the Kill
A=3 ;*
*B=A
GTM>
```

Example:

```
$ /usr/lib/fis-gtm/V5.4-002B_x86/gtm -run ^killalias
killalias ; Demonstrate Kill * of pass-by-reference
        ZPrint ; Print this program
        Set A=1,C=3
        Write "------------",!
        Write "Initial Values:",!
        ZWRite
        Do K1(.A,.C) ; Pass A & C by reference
        Write "-----------",!
        Write "Value of A is unchanged because of Kill *B, but C has changed: ",!
        ZWRite
        Quit
;
K1(B,D) ; A & C are bound to B & D respectively
        Write "------------",!
        Write "A & B are aliases, as are C & D:",!
        ZWRite
        Kill *B
```

```
        Set B=2,D=4
        Write "------------",!
        Write "After Kill *B, A & B are different but C & D remain associated:",!
        ZWrite
        Quit
------------
Initial Values:
A=1
C=3
------------
A & B are aliases, as are C & D:
A=1 ;*
*B=A
C=3 ;*
*D=C
------------
After Kill *B, A & B are different but C & D remain associated:
A=1
B=2
C=4 ;*
*D=C
------------
Value of A is unchanged because of Kill *B, but C has changed:
A=1
C=4
Example:
GTM>Set A=1,*B=A ; Create an array and association
GTM>ZWRite ; Verify that it's there
A=1 ;*
*B=A
GTM>Kill (A) ; Kill everything except A
GTM>ZWRite ; Demonstrate that A also has no array
GTM>Set A=2 ; Create an array
GTM>ZWRite ; The association survived the Kill
A=2 ;*
*B=A
GTM>
```

Annotated Alias Examples

Example:

```
$ /usr/lib/fis-gtm/V5.4-002B/gtm -run ^tprestart
tprestart ; Transaction restart variable association also restored on restart
  zprint ; Print this program
  set A="Malvern",C="Pennsylvania",E="USA"
  set *B=C,*D(19355)=E
  write "------------",!
  write "Initial values & association",!
  zwrite
  tstart (B,D) ; On restart: A not restored, B,D restored, C,E restored by association
  if '$TRestart Do  ; Change C,E if first time through
  .set C="Wales",E="UK"
  .kill *D(19355)
  .write "------------",!
```

```
  .write "First time through transaction; B,C,D,E changed",!
  .zwrite
  .set A="Brynmawr"
  .kill *B
  .write "------------",!
  .write "A changed; association between B & C and D & E killed; B,D have no value",!
  .zwrite
  .trestart
  else  Do  ; Show restored values on restart
  write "------------",!
  write "Second time through transaction; B,C,D,E & association restored",!
  zwrite
  tcommit ; No global updates in this transaction!
  quit
------------
Initial values & association
A="Malvern"
B="Pennsylvania" ;*
*C=B
*D(19355)=E
E="USA" ;*
------------
First time through transaction; B,C,D,E changed
A="Malvern"
B="Wales" ;*
*C=B
E="UK" ;*
------------
A changed; association between B & C and D & E killed; B,D have no value
A="Brynmawr"
C="Wales" ;*
E="UK" ;*
------------
Second time through transaction; B,C,D,E & association restored
A="Brynmawr"
B="Pennsylvania" ;*
*C=B
*D(19355)=E
E="USA" ;*
```

Note that TROLLBACK does not restore alias variables:

```
/usr/lib/fis-gtm/V5.4-002B_x86/gtm -run ^tprollback
tprollback ;
  zprint ; Print this program
  set A(1)=1,A(2)=2,A(3)=3
  set B(1)="1b",*B(2)=A,B(3)=3 ; B includes a container for A
  set *C(1)=B    ; C includes a container for B
  kill *A,*B   ; C is the only way to the data
  write "------------",!
  write "Only containers before transaction:",!
  zwrite
  tstart (C)
  if '$trestart
  .set *D=C(1) ; D is now an alias for what used to be B
  .set D(3)=-D(3)
```

```
  .set *D=D(2) ; D is now an alias for what used to be A
  .set D(1)=-D(1)
  .kill *D  ; Kill D after is used to manipulate the arrays
  .write "------------",!
  .write "Changed values before restart:",!
  .zwrite
  .trestart
  write "------------",!
  write "Restored values restart:",!
  zwrite
  kill C ; Kill only handle to arrays
  write "------------",!
  write "No local arrays left:",!
  zwrite
  trollback  ; Rollback transaction, don't commit it
  write "------------",!
  write "Rollback doesnt restore names and local arrays",!
  zwrite
  quit
------------
Only containers before transaction:
$ZWRTAC=""
*C(1)=$ZWRTAC1
$ZWRTAC1(1)="1b"
*$ZWRTAC1(2)=$ZWRTAC2
$ZWRTAC2(1)=1
$ZWRTAC2(2)=2
$ZWRTAC2(3)=3
$ZWRTAC1(3)=3
$ZWRTAC=""
------------
Restored values restart:
$ZWRTAC=""
*C(1)=$ZWRTAC1
$ZWRTAC1(1)="1b"
*$ZWRTAC1(2)=$ZWRTAC2
$ZWRTAC2(1)=1
$ZWRTAC2(2)=2
$ZWRTAC2(3)=3
$ZWRTAC1(3)=3
$ZWRTAC=""
------------
No local arrays left:
------------
Rollback doesnt restore names and local arrays
```

Example:

```
$ /usr/lib/fis-gtm/V5.4-002B_x86/gtm -run ^aliasexample; Extended annotated alias example
   zprint
   write "------------",!
   set x="name level",x(1)=1,x(1,2)="1,2",x("foo")="bar"
   write $ZDATA(x),! ; x is a conventional lvn - output 11
   set *y=x ; x an y are now alias variables
   write $ZDATA(x),! ; output appears as 111
   set *a(1)=y ; a(1) is now an alias container variable
```

```
    set b="bness",b("b")="bbness" ; b is a conventional lvn
    set *b=a(1) ; b joins x and y as alias variables for the same data
    ; prior b values are lost
    ; set *<name> is equivalent to Kill *<name> Set *<name>
    set y("hi")="sailor" ; Assignment applies to all of {b,x,y}
    kill b("foo") ; Kill applies to all of {b,x,y}
    kill *x ; x is undefined and no longer an alias variable
    ; b and y still provide access to the data
    write a(1),"<",! ; output appears as <
    write a(1)*3,! ; output appears as 0
    write $length(a(1)),! ; output appears as 0
    set c=y,c("legs")="tars" ; c is conventional lvn with value "name level"
    do sub1
    write $Data(c),! ; output is 1
    do sub2(.c)
    set a(1)="" ; a(1) ceases to be an alias container variable
    ; has the value ""
    write $D(i),! ; output is 0
    kill *c,*y ; c and y become undefined lvns
    zwrite b ; output is b("got")="a match"
    ; it's no longer an alias variable
    ; as everything else has gone
    quit
sub1
    new y ; in this scope y is no longer an alias for b
    set *y=c ; in this scope c and y are alias variables
    kill y("legs") ; Kill apples to all of {c,y}
    kill *y ; in this scope y is no longer an alias for c
    ; this is really redundant as
    ; the Quit implicitly does the same thing
    quit
sub2(i) ; i and c are joined due to pass-by-reference
    write $ZAHandle(c)=$ZAHandle(i),! ; output appears as 1
    kill b ; data for {b,y} is gone
    ; both are undefined, but remain alias variables
    set *c=a(1) ; c joins {b,y} as alias variable; prior value of c lost
    ; c is no longer alias of i
    write $ZAHandle(c)=$ZAHandle(i),! ; output appears as 0
    set i=a(1) ; Assignment applies to i - value is ""
    wet c("got")="a match" ; Assignment applies to all of {b,c,y)
    quit


------------
11
111
<
0
0
1
1
0
0
b("got")="a match"
```

Extensions for the support for the Unicode® standard

To represent and process strings that use international characters, GT.M processes can use the UTF-8 encoding defined by the Unicode® standard.

If the environment variable gtm_chset has a value of UTF-8 and either LC_ALL or LC_CTYPE is set to a locale with UTF-8 support (for example, zh_CN.utf8), a GT.M process interprets strings as containing characters encoded in the UTF-8 representation. In the UTF-8 mode, GT.M no longer assumes that one character is one byte, or that the glyph display width of a character is one. Depending on how ICU is built on a computer system, in order to operate in UTF-8 mode, a GT.M process may well also need a third environment variable, gtm_icu_version set appropriately.

If the environment variable gtm_chset has no value, the string "M", or any value other than "UTF-8", GT.M treats each 8-bit byte as a character, which suffices for English, and many single-language applications.

All GT.M components related to the M mode reside in the top level directory in which a GT.M release is installed and the environment variable gtm_dist should point to that directory for M mode processes. All GT.M components related to the UTF-8 mode reside in the utf8 subdirectory and the environment variable gtm_dist should point to that subdirectory for UTF-8 mode processes. So, in addition to the values of the environment variables gtm_chset and LC_ALL/LC_CTYPE, gtm_dist for a UTF-8 process should also point to the utf8 subdirectory.

M mode and UTF-8 mode are set for the process, not for the database. As a subset of UTF-8 characters, ASCII characters ($CHAR() values 0 through 127) are interpreted identically by processes in M and UTF-8 modes. The indexes and values in the database are simply sequences of bytes and therefore it is possible for one process to interpret a global node as encoded in UTF-8 and for another to interpret the same node as bytecodes. Note that such an application configuration would be extremely unusual, except perhaps during a transition phase or in connection with data import/export.

In UTF-8 mode, string processing functions (such as $EXTRACT()) operate on strings of multi-byte characters, and can therefore produce different results in M and UTF-8 modes, depending on the actual data processed. Each function has a "Z" alter ego (for example, $ZEXTRACT()) that can be used to operate on sequences of bytes identically in M and UTF-8 modes (that is, in M mode, $EXTRACT() and $ZEXTRACT() behave identically).

In M mode, the concept of an illegal character does not exist. In UTF-8 mode, a sequence of bytes may not represent a valid character, and generates an error when encountered by functions that expect and process UTF-8 strings. During a migration of an application to add the support for UTF-8 mode, illegal character errors may be frequent and indicative of application code that is yet to be modified. VIEW "NOBADCHAR" suppresses these errors at times when their presence impedes development.

In UTF-8 mode, GT.M also supports IO encoded in UTF-16 variants as well as in the traditional one byte per character encoding from devices other than $PRINCIPAL.

The following table summarizes GT.M's support for the Unicode® standard.

EXTENSION	EXPLANATION
$ASCII()	IN UTF-8 mode, the $ASCII() function returns the integer UTF-8 code-point value of a character in the given string. Note that the name $ASCII() is somewhat anomalous for UTF-8 data but that name is the logical extension of the function from M mode to UTF-8 mode. For more information and usage examples, refer to "$ASCII()" (page 215).
$Char()	In UTF-8 mode, $CHAR() returns a string composed of characters represented by the integer equivalents of the UTF-8 code-points specified in its argument(s). For more information and usage examples, refer to "$Char()" (page 216).
$Extract()	The $EXTRACT() function returns a substring of a given string. For more information and usage examples, refer to "$Extract()" (page 218).

EXTENSION	EXPLANATION
$Find()	The $FIND() function returns an integer character position that locates the occurrence of a substring within a string. For more information and usage examples, refer to "$Find()" (page 220).
$Justify()	The $JUSTIFY function returns a formatted string. For more information and usage examples, refer to "$Justify()" (page 224).
$Length()	The $LENGTH() function returns the length of a string measured in characters, or in "pieces" separated by a delimiter specified by its optional second argument. For more information and usage examples, refer to "$Length()" (page 226).
$Piece()	The $PIECE() function returns a substring delimited by a specified string delimiter made up of one or more characters. For more information and usage examples, refer to "$Piece()" (page 231).
$TRanslate()	The $TRANSLATE() function returns a string that results from replacing or dropping characters in the first of its arguments as specified by the patterns of its other arguments. For more information and usage examples, refer to "$TRanslate()" (page 242).
$X	For UTF-8 mode and TRM and SD output, $X increases by the display-columns (width in glyphs) of a given string that is written to the current device. For more information and usage examples, refer to "$X" (page 301).
$ZASCII()	The $ZASCII() function returns the numeric byte value (0 through 255) of a given sequence of octets (8-bit bytes). For more information and usage examples, refer to "$ZAscii()" (page 250).
$ZCHset	The read-only intrinsic special variable $ZCHSET takes its value from the environment variable gtm_chset. An application can obtain the character set used by a GT.M process by the value of $ZCHSET. $ZCHSET can have only two values –"M", or "UTF-8" and it cannot appear on the left of an equal sign in the SET command. For more information and usage examples, refer to "$ZCHset" (page 303).
$ZCHar()	The $ZCHAR() function returns a byte sequence of one or more bytes corresponding to numeric byte value (0 through 255) specified in its argument(s). For more information and usage examples, refer to "$ZCHar()" (page 259).
$ZCOnvert()	The $ZCONVERT() function returns its first argument as a string converted to a different encoding. The two argument form changes the encoding for case within a character set. The three argument form changes the encoding scheme. For more information and usage examples, refer to "$ZCOnvert()" (page 260).
$ZExtract()	The $ZEXTRACT() function returns a byte sequence of a given sequence of octets (8-bit bytes). For more information and usage examples, refer to "$ZExtract()" (page 265).
$ZFind()	The $ZFIND() function returns an integer byte position that locates the occurrence of a byte sequence within a sequence of octets(8-bit bytes). For more information and usage examples, refer to "$ZFind()" (page 266).
$ZJustify()	The $JUSTIFY() function returns a formatted and fixed length byte sequence. For more information and usage examples, refer to "$ZJustify()" (page 269).
$ZLength()	The $ZLENGTH() function returns the length of a sequence of octets measured in bytes, or in "pieces" separated by a delimiter specified by its optional second argument. For more information and usage examples, refer to "$ZLength()" (page 270).
$ZPATNumeric	ZPATN[UMERIC] is a read-only intrinsic special variable that determines how GT.M interprets the patcode N used in the pattern match operator. With $ZPATNUMERIC="UTF-8", the patcode N matches any numeric character as defined by the Unicode standard. By default patcode N only matches the ASCII digits, which are the only digits which M actually treats as numerics. For more information and usage examples, refer to "$ZPATNumeric" (page 315).

EXTENSION	EXPLANATION
$ZPIece()	The $ZPIECE() function returns a sequence of bytes delimited by a specified byte sequence made up of one or more bytes. In M, $ZPIECE() typically returns a logical field from a logical record. For more information and usage examples, refer to "$ZPIece()" (page 273).
$ZPROMpt	$ZPROM[PT] contains a string value specifying the current Direct Mode prompt. By default, GTM> is the Direct Mode prompt. M routines can modify $ZPROMPT by means of a SET command. $ZPROMPT cannot exceed 31 bytes. If an attempt is made to assign $ZPROMPT to a longer string, GT.M takes only the first 31 bytes and truncates the rest. With character set UTF-8 specified, if the 31st byte is not the end of a valid UTF-8 character, GT.M truncates the $ZPROMPT value at the end of last character that completely fits within the 31 byte limit. For more information and usage examples, refer to "$ZPROMpt" (page 318).
$ZSUBstr()	The $ZSUBSTR() function returns a properly encoded string from a sequence of bytes. For more information and usage examples, refer to "$ZSUBstr()" (page 284).
$ZTRanslate()	The $ZTRANSLATE() function returns a byte sequence that results from replacing or dropping bytes in the first of its arguments as specified by the patterns of its other arguments. $ZTRANSLATE() provides a tool for tasks such as encryption.For more information and usage examples, refer to "$ZTRanslate()" (page 285).
$ZWidth()	The $ZWIDTH() function returns the numbers of columns required to display a given string on the screen or printer. For more information and usage examples, refer to "$ZWidth()" (page 289).
%HEX2UTF	The GT.M %HEX2UTF utility returns the GT.M encoded character string from the given bytestream in hexadecimal notation. This routine has entry points for both interactive and non-interactive use. For more information and usage examples, refer to "%HEX2UTF" (page 518).
%UTF2HEX	The GT.M %UTF2HEX utility returns the hexadecimal notation of the internal byte encoding of a UTF-8 encoded GT.M character string. This routine has entry points for both interactive and non-interactive use. For more information and usage examples, refer to "%UTF2HEX" (page 518).
[NO]WRAP (USE)	Enables or disables automatic record termination. When the current record size ($X) reaches the maximum WIDTH and the device has WRAP enabled, GT.M starts a new record, as if the routine had issued a WRITE ! command. For more information and usage examples, refer to "WRAP" (page 433).
DSE and LKE	In UTF-8 mode, DSE and LKE accept Unicode characters in all their command qualifiers that require file names, keys, or data (such as DSE -KEY, DSE -DATA and LKE -LOCK qualifiers). For more information, refer to the LKE and DSE chapter For more information and usage examples, refer to **GT.M Administration and Operations Guide**.
GDE Objects	GDE allows the name of a file to include UTF-8 characters In UTF-8 mode, GDE considers a text file to be encoded in UTF-8 when it is executed via the "@" command. For more information, refer to the GDE chapter in **GT.M Administration and Operations Guide**.
FILTER[=expr]	Specifies character filtering for specified cursor movement sequences on devices where FILTER applies. In UTF-8 mode, the usual Unicode line terminators (U+000A (LF), U+0000D (CR), U+000D followed by U+000A (CRLF), U+0085 (NEL), U+000C (FF), U+2028 (LS) and U+2029 (PS)) are recognized. If FILTER=CHARACTER is enabled, all of the terminators are recognized to maintain the values of $X and $Y. For more information, refer to "FILTER" (page 425).
Job	The Job command spawns a background process with the same environment as the M process doing the spawning. Therefore, if the parent process is operating in UTF-8 mode, the Job'd process also operates in UTF-8 mode. In the event that a background process must have a different mode from the parent, create a shell script to alter the environment as needed, and spawn it with a ZSYstem command, for example, ZSYstem "/path/to/shell/script &", or start it as a PIPE device. For more information and UTF-8 mode examples, refer "Job" (page 125).

EXTENSION	EXPLANATION
MUPIP	MUPIP EXTRACT In UTF-8 mode, MUPIP EXTRACT, MUPIP JOURNAL -EXTRACT and MUPIP JOURNAL -LOSTTRANS write sequential output files in the UTF-8 character encoding form. For example, in UTF-8 mode if ^A has the value of 主要雨在西班牙停留在平原, the sequential output file of the MUPIP EXTRACT command is: 09-OCT-2006 04:27:53 ZWR GT.M MUPIP EXTRACT UTF-8 ^A="主要雨在西班牙停留在平原" MUPIP LOAD MUPIP LOAD command considers a sequential file as encoded in UTF-8 if the environment variable gtm_chset is set to UTF-8. Ensure that MUPIP EXTRACT commands and corresponding MUPIP LOAD commands execute with the same setting for the environment variable gtm_chset. The M utility programs %GO and %GI have the same requirement for mode matching. For more information on MUPIP EXTRACT and MUPIP LOAD, refer to the General Database Management chapter in **GT.M Administration and Operations Guide**.
Open	In UTF-8 mode, the OPEN command recognizes ICHSET, OCHSET, and CHSET as three additional deviceparameters to determine the encoding of the input / output devices. For more information and usage examples, refer to "Open" (page 142).
Pattern Match Operator (?)	GT.M allows the pattern string literals to contain UTF-8 characters. Additionally, GT.M extends the M standard pattern codes (patcodes) A, C, N, U, L, P and E to the UTF-8 character set. For more information, refer to "Pattern Match Operator" (page 85) and "$ZPATNumeric" (page 315).
Read	In UTF-8 mode, the READ command uses the character set value specified on the device OPEN as the character encoding of the input device. If character set "M" or "UTF-8" is specified, the data is read with no transformation. If character set is "UTF-16", "UTF-16LE", or "UTF-16BE", the data is read with the specified encoding and transformed to UTF-8. If the READ command encounters an illegal character or a character outside the selected representation, it triggers a run-time error. The READ command recognizes all Unicode line terminators for non-FIXED devices. For more information and usage examples, refer to "Read" (page 143).
Read #	When a number sign (#) and a non-zero integer expression immediately follow the variable name, the integer expression determines the maximum number of characters accepted as the input to the READ command. In UTF-8 or UTF-16 modes, this can occur in the middle of a sequence of combining code-points (some of which are typically non-spacing). When this happens, any display on the input device, may not represent the characters returned by the fixed-length READ (READ #). For more information and usage examples, refer to "Read" (page 143).
Read *	In UTF-8 or UTF-16 modes, the READ * command accepts one Unicode character of input and puts the numeric UTF-8 code-point value for that character into the variable. For more information and usage examples, refer to "Read" (page 143).
View "[NO]BADCHAR"	As an aid to migrating applications to using the Unicode standard, this UTF-8 mode VIEW command determines whether UTF-8 enabled functions trigger errors when they encounter illegal strings. For more information and usage examples, refer to "View" (page 151).
User-defined Collation	For some languages (such as Chinese), the ordering of character strings encoded with UTF-8 may not be the linguistically or culturally correct ordering. Supporting applications in such languages requires development of collation modules - GT.M natively supports M collation, but does not include pre-built collation modules for any specific natural language. Therefore, applications that use UTF-8 characters may need to implement their

EXTENSION	EXPLANATION
	own collation functions. For more information on developing a collation module for the Unicode® standard, refer to "Implementing an Alternative Collation Sequence for Unicode® characters" (page 558).
Unicode® Byte Order Marker (BOM)	When ICHSET is UTF-16, GT.M uses BOM (U+FEFF) to automatically determine the endianess. For this to happen, the BOM must appear at the beginning of the file or data stream. If BOM is not present, GT.M assumes big endianess. SEEK or APPEND operations require specifying the endianess (UTF-16LE or UTF-16BE) because they do not go to the beginning of the file or data stream to automatically determine the endianess. When endianess is not specified, SEEK or APPEND assume big endianess. If the character set of a device is UTF-8, GT.M checks for and ignores a BOM on input. If the BOM does not match the character set specified at device OPEN, GT.M produces an error. READ does not return BOM to the application and the BOM is not counted as part of the first record. If the output character set for a device is UTF-16 (but not UTF-16BE or UTF-16LE,) GT.M writes a BOM before the initial output. The application code does not need to explicitly write the BOM.
WIDTH=intexpr (USE)	In UTF-8 mode and TRM and SD output, the WIDTH deviceparameter specifies the display-columns and is used with $X to control truncation and WRAPing of the visual representation of the stream. For more information and usage examples, refer to "WIDTH" (page 432).
Write	In UTF-8 mode, the WRITE command uses the character set specified on the device OPEN as the character encoding of the output device. If character set specifies "M" or "UTF-8", GT.M WRITEs the data with no transformation. If character set specifies "UTF-16", "UTF-16LE" or "UTF-16BE", the data is assumed to be encoded in UTF-8 and WRITE transforms it to the character encoding specified by character set device parameter. For more information and usage examples, refer to "Write" (page 439).
Write *	When the argument of a WRITE command consists of a leading asterisk (*) followed by an integer expression, the WRITE command outputs the character represented by the code-point value of that integer expression. For more information and usage examples, refer to "Write" (page 439).
ZSHow	In UTF-8 mode, the ZSHOW command exhibits byte-oriented and display-oriented behavior as follows: 1. ZSHOW targeted to a device (ZSHOW "*") aligns the output according to the numbers of display columns specified by the WIDTH deviceparameter. 2. ZSHOW targeted to a local (ZSHOW "*":lcl) truncates data exceeding 2048KB at the last character that fully fits within the 2048KB limit. 3. ZSHOW targeted to a global (ZSHOW "*":^CC) truncates data exceeding the maximum record size for the target global at the last character that fully fits within that record size. For more information and usage examples, refer to "ZSHOW Destination Variables" (page 202).

Philosophy of GT.M's support for the Unicode® standard

With the support of the Unicode® standard, there is no change to the GT.M database engine or to the way that data is stored and manipulated. GT.M has always allowed indexes and values of M global and local variables to be either canonical numbers or any arbitrary sequence of bytes. There is also no change to the character set used for M source programs. M source programs have always been in ASCII (standard ASCII - $C(0) through $C(127) - is a proper subset of the UTF-8 encoding specified by the Unicode standard). GT.M accepts some non-ASCII characters in comments and string literals.

The changes in GT.M to support the Unicode standard are principally enhancements to M language features. Although conceptually simple, these changes fundamentally alter certain previously ingrained assumptions. For example:

1. The length of a string in characters is not the same as the length of a string in bytes. The length of a UTF-8 string in characters is always less than or equal to its length in bytes.

2. The display width of a string on a terminal is different from the length of a string in characters - for example, with UTF-8, a complex glyph may actually be composed of a series of glyphs or component symbols, each in turn a UTF-8 encoded character in a Unicode string.

3. As a glyph may be composed of multiple characters, a UTF-8 string can have canonical and non-canonical forms. The forms may be conceptually equivalent, but they are different strings of encoded characters.

> ## ⚠ Important
>
> GT.M treats canonical and non-canonical versions of the same string as different and unequal. FIS recommends that applications be written to use canonical forms. Where conformance to a canonical representation of input strings cannot be assured, application logic linguistically and culturally correct for each language should convert non-canonical strings to canonical strings.

Applications may operate on a combination of character and binary data - for example, some strings in the database may be digitized images of signatures and others may include escape sequences for laboratory instruments. Furthermore, since M applications have traditionally overloaded strings by storing different data items as pieces of the same string, the same string may contain both UTF-8 data and binary data. GT.M has functionality to allow a process to manipulate UTF-8 strings as well as binary data.

The GT.M design philosophy is to keep things simple, but no simpler than they need to be. These typically arise where interpretations of lengths and interpretations of characters interact. For example:

1. A sequence of bytes is never illegal when considered as binary data, but can be illegal when treated as a UTF-8 string. The detection and handling of illegal UTF-8 strings adds complexity, especially when binary and UTF-8 data reside in different pieces of the same string.

2. Since binary data may not map to graphic UTF-8 characters, the ZWRite format must represent such characters differently. A sequence of bytes that is output by a process interpreting it as UTF-8 data may require processing to form correctly input to a process that is interpreting that sequence as binary, and vice versa. Therefore, when performing IO operations, including MUPIP EXTRACT and MUPIP LOAD operations in ZWR format, ensure that processes have the compatible environment variables and /or logic to generate the desired output and correctly read and process the input.

3. Application logic managing input / output that interacts with human beings or non-GT.M applications requires even closer scrutiny. For example, fixed length records in files are always defined in terms of bytes. In Unicode support related operations, an application may output data such that a character would cross a record boundary (for example, a record may have two bytes of space left, and the next UTF-8 character may be three bytes long), in which case GT.M fills the record with one or more pad bytes. When a padded record is read as UTF-8, trailing pad bytes are stripped by GT.M and not provided to the application code.

For some languages (such as Chinese), the ordering of strings according to UTF-8 code-points (character values) may not be the linguistically or culturally correct ordering. Supporting applications in such languages requires development of collation modules - GT.M natively supports M collation, but does not include pre-built collation modules for any specific natural language.

Glyphs and Unicode® characters

Glyphs are the visual representation of text elements in writing systems and UTF-8 code-points are the underlying data. Internally, GT.M stores UTF-8 encoded strings as sequences of bytes. A Unicode® compatible output device - terminal, printer

or application - renders the characters as sequences of glyphs that depict the sequence of code-points, but there may not be a one-to-one correspondence between characters and glyphs.

For example, consider the following word from the Devanagari writing system.

अच्छी

On a screen or a printer, it is displayed in 4 columns. Internally GT.M stores it as a sequence of 5 UTF-8 code-points:

#	Character	UTF-8 code-point	Name
1	अ	U+0905	DEVANAGARI LETTER A
2	च	U+091A	DEVANAGARI LETTER CA
3	`	U+094D	DEVANAGARI SIGN VIRAMA
4	छ	U+091B	DEVANAGARI LETTER CHA
5	ी	U+0940	DEVANAGARI VOWEL SIGN II

The Devanagari writing system (U+0900 to U+097F) is based on the representation of syllables as contrasted with the use of an alphabet in English. Therefore, it uses the half-form of a consonant to represent certain syllables. The above example uses the half-form of the consonant (U+091A).

Although the half-form form consonant is a valid text element in the context of the Devanagari writing system, it does not map directly to a character in the Unicode® standard. It is obtained by combining the DEVANAGARI LETTER CA, with DEVANAGARI SIGN VIRAMA, and DEVANAGARI LETTER CHA.

च		+		`		+		छ		=		च्छ

On a screen or a printer, the terminal font detects the glyph image of the half-consonant and displays it at the next display position. Internally GT.M uses ICU's glyph-related conventions for the Devanagari writing system to calculate the number of columns needed to display it. As a result, GT.M advances $X by 1 when it encounters the combination of the 3 UTF-8 code-points that represent the half-form consonant.

To view this example at GT.M prompt, type the following command sequence:

```
GTM>write $ZCHSET
UTF-8
GTM>set DS=$char($$FUNC^%HD("0905"))_$char($$FUNC^%HD("091A"))_$char($$FUNC^%HD("094D"))
GTM>set DS=DS_$char($$FUNC^%HD("091B"))_$char($$FUNC^%HD("0940"))
GTM>write $zwidth(DS); 4 columns are required to display local variable DS on the screen.
4
GTM>write $length(DS); DS contains 5 characters or UTF-8 code-points.
5
GTM>
```

For all writing systems supported by the Unicode standard, a character is a code-point for string processing, network transmission, storage, and retrieval of Unicode data whereas a character is a glyph for displaying on the screen or printer.

This holds true for many other popular programming languages. Keep this distinction in mind throughout the application development life-cycle.

ICU

ICU is a widely used, defacto standard package (see http://icu-project.org for more information) that GT.M relies on for most operations that require knowledge of the Unicode® character sets, such as text boundary detection, character string conversion between UTF-8 and UTF-16, and calculating glyph display widths.

> ⚠️ **Important**
>
> Unless the support for the Unicode standard is sought for a process (that is, unless the environment variable gtm_chset is UTF-8"), GT.M processes do not need ICU. In other words, existing applications that are not based on the Unicode standard continue to work on supported platforms without ICU.

An ICU version number is of the form major.minor.milli.micro where major, minor, milli and micro are integers. Two versions that have different major and/or minor version numbers can differ in functionality and API compatibility is not guaranteed. Differences in milli or micro versions are maintenance releases that preserve functionality and API compatibility. ICU reference releases are defined by major and minor version numbers. Note that display widths for some characters changed in ICU 4.0 and may change again in the future, as both languages and ICU evolve.

An operating system's distribution generally includes an ICU library tailored to the OS and hardware, therefore FIS does not provide any ICU library. In order to support UTF-8 functionality, GT.M requires an appropriate version of ICU to be installed on the system - check the release notes for your GT.M release for supported ICU versions.

GT.M expects ICU to be compiled with symbol renaming disabled and will issue an error at startup if the available version of ICU is built with symbol renaming enabled. To use a version of ICU built with symbol renaming enabled, the gtm_icu_version environment variable indicates the MAJOR VERSION and MINOR VERSION numbers of the desired ICU formatted as MajorVersion.MinorVersion (for example "3.6" to denote ICU-3.6). When $gtm_icu_version is so defined, GT.M attempts to open the specific version of ICU. In this case, GT.M works regardless of whether or not symbols in this ICU have been renamed. A missing or ill-formed value for this environment variable causes GT.M to only look for non-renamed ICU symbols. The release notes for each GT.M release identify the required reference release version number as well as the milli and micro version numbers that were used to test GT.M prior to release. In general, it should be safe to use any version of ICU with the specific ICU reference version number required and milli and micro version numbers greater than those identified in the release notes for that GT.M version.

ICU supports multiple threads within a process, and an ICU binary library can be compiled from source code to either support or not support multiple threads. In contrast, GT.M does not support multiple threads within a GT.M process. On some platforms, the stock ICU library, which is usually compiled to support multiple threads, may work unaltered with GT.M. On other platforms, it may be required to rebuild ICU from its source files with support for multiple threads turned off. Refer to the release notes for each GT.M release for details about the specific configuration tested and supported. In general, the GT.M team's preference for ICU binaries used for each GT.M version are, in decreasing order of preference:

1. The stock ICU binary provided with the operating system distribution.

2. A binary distribution of ICU from the download section of the ICU project page.

3. A version of ICU locally compiled from source code provided by the operating system distribution with a configuration disabling multi-threading.

4. A version of ICU locally compiled from the source code from the ICU project page with a configuration disabling multi-threading.

GT.M uses the POSIX function dlopen() to dynamically link to ICU. In the event you have other applications that require ICU compiled with threads, place the different builds of ICU in different locations, and use the dlopen() search path feature (for example, the LD_LIBRARY_PATH environment variable on Linux) to enable each application to link with its appropriate ICU.

Discussion and Best Practices

Data interchange

GT.M's support for the Unicode® standard only affects the interpretation of data in databases, and not databases themselves. A simple way to convert from a ZWR format extract in one mode to an extract in the other is to load it in the database using a process in the mode in which it was generated, and to once more extract it from the database using a process in the other mode.

If a sequence of 8-bit octets contains bytes other than those in the ASCII range (0 through 127), an extract in ZWR format for the same sequence of bytes is different in M and UTF-8 modes. In M mode, the $[Z]CHAR() values in a ZWR format extract are always equal to or less than 255. In UTF-8 mode, they can have larger values - the code-points of UTF-8 characters can be far greater than 255.

Note that the characters written to the output device are subject to the OCHSET transformation of the controlling output device. If OCHSET is "M", the multi-byte characters are written in raw bytes without any transformation.

1. Each multi-byte graphic character (as classified by $ZCHSET) is written directly to the device converted to the encoding form specified by the OCHSET of the output device.

2. Each multi-byte non-graphic character (as classified by $ZCHSET) is written in $CHAR(nnnn) notation, where nnnn is the decimal character code (that is, code-point up to 1114111 if $ZCHSET="UTF-8" or up to 255 if $ZCHSET="M").

3. If $ZCHSET="UTF-8" and a subscript or data contains a malformed UTF-8 byte sequence, ZWRITE treats each byte in the sequence as a separate malformed character. Each such byte is written in $ZCHAR(nn[,...]) notation, where each nn is the corresponding byte in the illegal UTF-8 byte sequence.

Note that attempts to use ZWRITE output from a system as input to another system using a different character set may result in errors or not yield the same state as existed on the source system. Application developers can deal with this by defining and using one or more pattern tables that declare all non-ASCII characters (or any useful subset thereof) to be non-graphic. For more details on defining pattern tables, please refer to "Pattern Code Definition" section of Chapter 12: "Internationalization" (page 543).

Limitations

User-defined pattern codes are not supported

Although the M standard patcodes (A,C,L,U,N,P,E) are extended to work in UTF-8 mode, application developers can neither change their default classification nor define the non-standard patcodes ((B,D,F-K,M,O,Q-T,V-X) beyond the ASCII subset. This means that the pattern tables cannot contain characters with codes greater than the maximum ASCII code 127.

String Normalization

In GT.M, strings are not implicitly normalized. Unicode® normalization is a method of computing canonical representation of the character strings. Normalization is required if the strings contain combination characters (such as accented characters consisting of a base character followed by an accent character) as well as precomposed characters. The Unicode standard

assigned code-points to such precomposed characters for backward compatibility with legacy code sets. For the applications containing both versions of the same character (or combining characters), the Unicode standard recommends one of the normal forms. Because GT.M does not normalize strings, the application developers must develop the functionality of normalizing the strings, as needed, in order for string matching and string collation to behave in a conventional and wholesome fashion. In such a case, edit checks can be used that only accept a single representation when multiple representations are possible.

UTF-16 is not supported for $PRINCIPAL device

In GT.M does not support UTF-16, UTF-16LE and UTF-16BE encodings for $PRINCIPAL I/O devices (including Terminal, Sequential and Socket devices). In order to perform Unicode®-related I/O with the $PRINCIPAL device, application developers must use "UTF-8" for the ICHSET or OCHSET deviceparameters.

UTF-16 is not supported for Terminal Devices

Due to the uncommon usage and lack of support for UTF-16 by UNIX terminals and terminal emulators, GT.M does not support UTF-16, UTF-16LE and UTF-16BE encodings for Terminal I/O devices. Note that UNIX platforms use UTF-8 as the defacto character encoding for the Unicode® standard. The terminal connections from remote hosts (such as Windows) must communicate with GT.M in UTF-8 encoding.

Error messages are in [American] English

GT.M has no facility for a translation of product error messages or on-line help into languages other than [American] English. All error message text (except the messages arguments that could include Unicode® data) is in the [American] English language.

Performance and Capacity

With the use of UTF-8 as GT.M's internal character encoding, the additional requirements for CPU cycles, excluding collation algorithms, should not increase significantly compared with the identical application using the M character set. Additional memory requirements for UTF-8 vary depending on the application as well as the actual character set used. For example, applications based on Latin-1 (2-byte encoded) characters may require up to twice the memory and those based on Chinese/Japanese (3-byte encoded) characters may require up to three times the memory compared to an identical application using "M" (ASCII) characters. The additional disk-space and I/O performance trade-offs for UTF-8 also vary based on the application and the characters used.

Characters in arguments exchanged with external routines must be validated by the external routines

GT.M does not check for illegal characters in a string before passing it to an external routine or in a returned value before assigning it to a GT.M variable. This is because such checks add parameter-processing overhead. The application must ensure that the strings are in the encoding form expected by the respective routines. More robustly, external routines must interpret passed strings based on the value of the intrinsic variable $ZCHSET or the environment variable gtm_chset. The external routines can perform validation if needed.

Maximums

In older versions of GT.M which did not support the Unicode® standard, the restrictions on certain objects were put in place with the assumption that a character is represented by a single byte. With support for the Unicode standard in GT.M, the following restrictions are in terms of bytes- not characters.

M Name Length

The maximum length of an M identifier is restricted to 31 bytes. Since identifier names are restricted to be in ASCII, programmers can define M names up to 31 characters long.

M String Length

The maximum length of an M string is restricted to 1,048,576 bytes (1MiB). Therefore, depending on the characters used, the maximum number of characters could be reduced from 1,048,576 characters to as few as 262,144 (256K) characters.

M Source Line Length

The maximum length of a program or indirect source line is restricted to 2,048 bytes. Application developers must be aware of this byte limit if they consider using multi-byte source comments or string literals in a source line.

Database Key and Record Sizes

The maximum allowed size for database keys (both global and nref keys) is 1019 bytes and for database records is 1MiB. Application developers must be aware that the keys or data containing multi-byte UTF-8 characters are limited at a smaller number of characters than the number of available bytes.

Ten Golden Rules

Adhere to the following rules of thumb to design and develop applications based on the Unicode® standard for deployment on GT.M.

1. GT.M functionality related to the Unicode standard and characters becomes available only in UTF-8 mode.

2. [At least] in UTF-8 mode, byte manipulation must use Z* equivalent functions.

3. In M mode, standard functions are always identical to their Z equivalents.

4. Use the same character set for all globals names and subscripts in an instance.

5. Define a collation system according to the linguistic and cultural tenets of the language used.

6. Create the application logic to ensure strings used as keys are canonical.

7. Specify CHSET="M" or otherwise handle illegal characters during the I/O operations.

8. Communicate with any external routines using a compatible character encoding form.

9. Compile and run programs in the same setting of $ZCHSET and "BADCHAR".

Chapter 3. Development Cycle

Revision History		
Revision V7.1-001	26 June 2023	• In "Compiling from the Shell" (page 42), add file extension info
Revision V6.3-014	06 October 2020	• In "Processing Errors from Direct Mode and Shell" (page 48), add case for informational messages
Revision V6.3-006	26 October 2018	• In "Compiling from the Shell" (page 42), add that if you do not specify a .m file extension, the mumps command assumes the .m file extension. • In "MUMPS Command Qualifiers Summary" (page 46), add entry for -nameofrtn. • In "-[no]o[bject][=filename]" (page 45), add a note about compilation with -NAMEOFRTN. • In "Qualifiers for the mumps command" (page 43), add information about the -nameofrtn qualifier.
Revision V6.3-001	20 March 2017	• In "Defining Environment Variables" (page 34), specified the environment variable length limit and mentioned the differences in shell syntax. • In "-[no]w[arning]" (page 45), added clarification to the note. • In "Preparing the Database" (page 37), removed HP-UX
Revision V6.0-003	24 February 2014	In "Qualifiers for the mumps command" (page 43), added the descriptions of -DY[NAMIC_LITERALS] and -NOIN[LINE_LITERALS] qualifiers.
Revision V6.0-001	21 March 2013	Added a new section called "Processing Errors from Direct Mode and Shell" (page 48).

This chapter introduces program development in the GT.M environment. The GT.M environment differs from other M implementations in a number of ways. These differences, which include maintaining data and code in separate files and compiling rather than interpreting source code, allow greater programmer control over the development cycle.

In contrast to M environments that interpret M code, GT.M compiles M code from source files into the target machine language. The GT.M compiler produces object files, which are dynamically linked into an image. Source files and object files may be managed independently, or placed together in a specific directory. GT.M permits access to source and object files in multiple directories.

GT.M databases are UNIX files identified by a small file called a Global Directory. Global Directories allow management of the database files to be independent of the placement of files containing M routines. By changing the Global Directory, you can use the same programs to access different databases.

Program development may utilize both GT.M and UNIX development tools. The development methodology and environment chosen for a particular installation, and tailored by the individual user, determines the actual mix of tools. These tools may vary from entirely GT.M with little UNIX, to mostly UNIX with a modest use of GT.M.

Direct Mode serves as an interactive interface to the GT.M run-time environment and the compiler. In Direct Mode, the user enters M commands at the GT.M prompt, and GT.M compiles and executes the command. This feature provides immediate turnaround for rapid program development and maintenance.

This chapter is based on the tasks that a programmer might perform while developing an application. It provides a "road map" for programmers of varying levels. Some steps may be unnecessary in your environment, so feel free to skip sections that do not apply to your situation.

Overview of the Program Development Cycle

This section provides an overview of the steps involved in generating executable programs in GT.M.

The steps begin with your initial use of GT.M. The first two steps are part of your initial setup and will generally be performed only the first time you use GT.M. The remaining steps are those you will use regularly when generating your programs.

Each of these remaining steps can be performed either from the GT.M prompt or the shell prompt. To clearly describe the two ways to perform each step, this section is set up in the format of a table with one column illustrating the GT.M method, and one column illustrating the shell method.

Creating a GT.M Routine		
1) Define environment variables (shell)	define gtm_dist gtmgbldir gtmroutines	
2) Prepare database (GT.M)	define Global Directory with GDE, create database with MUPIP CREATE	
-	SHELL	GT.M
3) Create/Edit routine	Create file with UNIX editor; assign .m extension	ZEDIT "routine" .m extension added by GT.M
4) Compile routine	invoke mumps routine.m	ZLINK "routine"
5) Execute routine	invoke mumps -run routine calls from other routines invoke auto-ZLINK	Do ^routine calls from other routines invoke auto-ZLINK
6) Debug routine	edit file with UNIX editor; repeat steps 4, 5	utilize GT.M debug commands such as: ZGOTO ZLINK

Creating a GT.M Routine		
		ZMESSAGE
		ZPRINT
		ZSHOW
		ZSTEP
		ZSYSTEM
		ZWRITE
		repeat steps 4, 5

The table is presented as an overview of the GT.M routine generation process, and as a comparison of the available methods. More complete information on each of the steps can be found in the following parts of this manual set.

1. Debugging routines: Chapter 4: "Operating and Debugging in Direct Mode" (page 50).

2. Defining environment variables: "Defining Environment Variables" (page 34).

3. Defining/creating Global Directories: "Preparing the Database" (page 37) and **GT.M Administration and Operations Guide**, "Global Directory Editor" and "MUPIP" chapters.

4. Creating/editing routines: "Creating and Editing a Source Program" (page 39).

5. Compiling routines: "Compiling a Source Program" (page 40).

6. Executing routines: "Executing a Source Program" (page 46).

Defining Environment Variables

GT.M requires the definition of certain environment variables as part of setting up the environment. These environment variables are used for the following purposes:

- To locate the files that FIS provides as part of GT.M

- To hold some user-controlled information which GT.M uses for run-time operation

GT.M limits environment variables to 8192 bytes, but items they specify such as a path may have a lower limit.

The procedure below describes how to define an environment variable. Use this procedure to define an environment variable either at the shell prompt or in your shell startup file. If you define the variable at the shell prompt, it will be effective only until you logout. If you define it in your .profile file (.cshrc, if using a C shell variant), it will be in effect whenever you log in. Your system manager may have already defined some of these variables.

Note

Each environment variable required by GT.M is described and illustrated in individual sections following the procedure. Only gtm_dist, and in some cases gtmgbldir, gtm_principal and gtmroutines, are required by users who do not perform programming activities.

To define an environment variable type the following commands:

```
$ env_variable=env_variable_value
$ export env_variable
```

The example above may differ from the syntax supported by some shells

The following environment variables hold information that determines some details of GT.M run-time operation, over which the user has control.

gtm_dist

gtm_dist is used to establish the location of the installed GT.M program and support files.

The syntax for gtm_dist is as follows:

```
$ gtm_dist=<GT.M-distribution-directory>
```

The standard installation places these files in /usr/lib/fis-gtm.

Example:

```
$ gtm_dist=/usr/lib/fis-gtm/V6.0-002_x86_64
$ export gtm_dist
```

This identifies /usr/lib/fis-gtm/V6.0-002_x86_64 as the location of the installed GT.M files.

Add gtm_dist to your PATH environment variable to have UNIX search the GT.M installation directory (when processing a command to activate or run an image). This allows you to activate GT.M and the utilities without explicitly specifying a path.

To add gtm_dist to your PATH type the following commands:

```
$ PATH=$PATH:$gtm_dist
$ export PATH
```

> ### Note
>
> Most of the examples in this manual assume that you have added gtm_dist to your PATH.

gtmgbldir

gtmgbldir defines the path to a Global Directory. A Global Directory maps global variables to physical database files, and is required to locate M global variables. gtmgbldir provides the initial value for $ZGBLDIR, the intrinsic special variable that connects the GT.M run-time system to the Global Directory. It also connects the Global Directory to the utilities requiring one.

If you maintain multiple global directories, define gtmgbldir to the Global Directory you currently want to use.

The syntax of a gtmgbldir definition is:

```
$ gtmgbldir=/directory/filename.gld
```

Example:

```
$ gtmgbldir=/usr/staff/mumps.gld
```

```
$ export gtmgbldir
```

This specifies /usr/staff as the directory containing the Global Directory file named mumps.gld.

gtm_principal

The gtm_principal environment variable specifies the value for $principal, which designates the absolute pathname of the principal $IO device. This is an MDC Type A enhancement to standard M.

The following is an example of gtm_principal definition:

```
$ gtm_principal=/usr/filename
$ export gtm_principal
```

This specifies the /usr/filename as the principal $IO device, effective until changed further or until you logout of the particular session.

gtmroutines

The gtmroutines environment variable specifies a search list of possible locations for M routines. This value is used to initialize $ZROUTINES, (the intrinsic special variable that enables GT.M to find the routine (program) you want to run). gtmroutines is required for ZLINKing. gtmroutines is particularly helpful in calling percent utilities and the Global Directory Editor (GDE), which are in gtm_dist.

```
$ gtmroutines="directories in search list"
```

The directories in the search list must be separated by a space and enclosed in quotation marks (" "). Environment variables are accepted in the search list.

The following is an example of gtmroutines definition:

```
$ gtmroutines=". $gtm_dist"
$ export gtmroutines
```

This specifies that GT.M search for a routine first in the current directory (.), then in the distribution directory (which is identified by the environment variable gtm_dist). The distribution directory is included in the list because it contains the percent routines. You will probably want the search list to contain these two items at a minimum. In addition, you may want to add directories of your own.

For additional information about how GT.M uses the routine search list, see "$ZROutines" (page 319).

Editor

The EDITOR environment variable specifies the UNIX text editor used when editing a routine either from the shell or with ZEDIT. Since this is a standard part of establishing your UNIX environment, you will probably only need to define this when you want to use a different editor than the one defined in your shell startup file.

Example:

```
$ EDITOR=/usr/bin/vi
$ export EDITOR
```

This defines the current text editor to vi.

Preparing the Database

GT.M databases consist of one or more UNIX files. Most database files have a UNIX file structure externally and a GT.M Database Structure (GDS) internally. Management of the GDS files by the GT.M run-time system assures high performance and integrity. GT.M database files are coordinated by a Global Directory. The Global Directory identifies which global names belong in which files, and specifies the creation characteristics for each file. To specify access to a database, each M process must define the gtmgbldir environment variable to point to the associated Global Directory.

To define and maintain a Global Directory, use the Global Directory Editor (GDE) utility. The GDE utility automatically upgrades existing global directories to the current format. The MUPIP command CREATE uses the characteristics as defined in the Global Directory to create the associated database. In a production environment, the system manager typically maintains Global Directories.

For more information on GDE and MUPIP refer to the "Global Directory Editor" and "MUPIP" chapters in the **GT.M Administration and Operations Guide**.

Example:

This example is a sequence of events that illustrate steps you might typically perform in creating a new global directory, in our example PAYROLL.GLD. To assist you in following the sequence, each actual step appears in typewriter font, as you might see on your terminal screen, followed by an explanation in normal text font.

```
$ ls payroll.gld
payroll.gld not found
```

The ls command verifies that there are no existing files with the name payroll.gld.

```
$ gtmgbldir=payroll.gld
$ export gtmgbldir
```

This establishes the current value of the environment variable gtmgbldir as payroll.gld. GT.M uses gtmgbldir to identify the current Global Directory. When defined at the shell prompt, gtmgbldir maintains the defined value only for the current login session. The next time you log into UNIX, you must again define the value of gtmgbldir as payroll.gld to use it as the current Global Directory.

This example defines gtmgbldir without a full pathname. The environment variable points to the payroll.gld file in the current working directory. Therefore if the default directory changes, GT.M attempts to locate the Global Directory in the new default directory and cannot use the original file. If you intend for the Global Directory to consistently point to this file, even if the default directory changes, use a full file-specification for gtmgbldir.

```
$ /usr/lib/fis-gtm/V6.0-0001_x86/gtm
GTM>do ^GDE
%GDE-I-LOADGD, Loading Global Directory file
        /home/jdoe/.fis-gtm/V6.0-001_x86/g/payroll.gld
%GDE-I-VERIFY, Verification OK
GDE>
```

This invokes the Global Directory Editor by entering GDE from the GT.M prompt and produces an informational message.

```
GDE> show -all
                        *** TEMPLATES ***
                              Def   Rec   Key Null      Standard
 Region                       Coll  Size  Size Subs     NullColl  Journaling
-------------------------------------------------------------------------------
 <default>                      0   4080   255 NEVER     Y         Y
            Jnl File (def ext: .mjl)  Before  Buff      Alloc Exten
-------------------------------------------------------------------------------
```

```
<default>        <based on DB file-spec>  Y        128      2048  2048
Segment          Active             Acc Typ Block      Alloc Exten Options
-----------------------------------------------------------------------
<default>          *               BG  DYN  4096      5000 10000 GLOB =1000
                                                                 LOCK = 40
                                                                 RES  =   0
                                                                 ENCR = OFF
<default>                          MM  DYN  4096      5000 10000 DEFER
                                                                 LOCK = 40
       *** NAMES ***
Global                             Region
-----------------------------------------------------------------------
*                                  DEFAULT
                         *** REGIONS ***
                         Dynamic                     Def   Rec   Key Null      Standard
Region                   Segment                     Coll  Size  Size Subs     NullColl  Journaling
-------------------------------------------------------------------------------------------------
DEFAULT                  DEFAULT                       0   4080    255 NEVER     Y          Y
                 *** JOURNALING INFORMATION ***
Region                   Jnl File (def ext: .mjl)  Before Buff      Alloc Exten
------------------------------------------------------------------------------
DEFAULT                  $gtmdir/$gtmver/g/payroll.mjl
                                               Y       128      2048  2048
                 *** SEGMENTS ***
Segment                  File (def ext: .dat)Acc Typ Block      Alloc Exten Options
------------------------------------------------------------------------------------
DEFAULT                  $gtmdir/$gtmver/g/dayroll.dat
                                      BG  DYN  4096      5000 10000 GLOB=1000
                                                                   LOCK=  40
                                                                   RES =   0
                                                                   ENCR=OFF
                 *** MAP ***
 - - - - - - - - - - Names - - - - - - - - - -
From                     Up to                        Region / Segment / File(def ext: .dat)
----------------------------------------------------------------------------------------------------------
%                        ...                          REG = DEFAULT
                                                      SEG = DEFAULT
                                                      FILE = $gtmdir/$gtmver/g/payroll.dat
LOCAL LOCKS                                           REG = DEFAULT
                                                      SEG = DEFAULT
                                                      FILE = $gtmdir/$gtmver/g/payroll.dat
```

The GDE SHOW command displays the default Global Directory.

```
GDE> change -segment default -allocation=1000 file=payroll.dat
```

The GDE CHANGE command sets the database file name to payroll.dat, and specifies a file size of 1000 blocks (of 1024 bytes).

```
GDE>exit
%GDE-I-VERIFY, Verification OK
%GDE-I-GDCREATE, Creating Global Directory file /usr/lib/fis-gtm/V6.0-001_x86/payroll.gld
%GDE-I-GDEIS, Global Directory
```

The GDE EXIT command terminates GDE. The Global Directory Editor creates a default Global Directory and displays a confirmation message.

```
$ ls payroll.gld
payroll.gld
```

This ls command shows the new Global Directory has been created.

In the simplest case, running the Global Directory Editor and immediately EXITing creates a Global Directory with a default single file database.

To create the database file payroll.dat, use the GT.M MUPIP CREATE utility.

Example:

```
$ mupip create
```

```
Created file payroll.dat
```

The MUPIP CREATE command generates the database file. Notice that the MUPIP CREATE syntax does not include the file name. MUPIP uses the environment variable gtmgbldir to find the Global Directory payroll.dat and obtains the file name from that Global Directory. MUPIP then checks to make sure that payroll.dat does not already exist and creates payroll.dat with the characteristics described in payroll.dat.

Example:

```
$ mupip load payroll.gld
GT.M MUPIP EXTRACT
02-MAY-2013  22:21:37 ZWR
Beginning LOAD at record number: 3
LOAD TOTAL              Key Cnt: 279  Max Subsc Len: 28  Max Data Len: 222
Last LOAD record number: 281
```

This uses the MUPIP LOAD command to load a sequential file into the database.

Because MUPIP uses the environment variable gtmgbldir to locate a Global Directory, which identifies the database file(s), the LOAD command does not require any information about the target database. With few exceptions, the GT.M utilities work in the same way.

Creating and Editing a Source Program

The first step in developing a GT.M program is to create a source file. In most cases, the user can create and modify GT.M source programs using UNIX text editors.

When the program is very simple (and its lines do not need revision after they are entered), you can use the cat command to direct input from your terminal to your source file.

Editing from GT.M

If you focus on program development outside the GT.M environment, skip this section and continue with the section "Editing from the Shell".

ZEDIT <filename>

Invoke Direct Mode to create and edit a source program in GT.M. At the GTM> prompt, invoke the editor by typing:

ZEDIT <filename>

ZEDIT invokes the editor specified by the EDITOR environment variable, which creates a seperate file for each M source module.

The GT.M environment works most efficiently if the file has the same name as the M routine it contains, and has an .m extension. Since ZEDIT automatically defaults the.m extension, it is not necessary to specify an extension unless you require a different one. If you use another extension, you must specify that extension with every reference to the file. Multiple character file extensions are permitted for M source file names.

Example:

```
$ /usr/lib/.fis-gtm/V5.4-002B_x86/gtm
```

```
GTM>ZEDIT "payroll"
```

This syntax uses the gtm script to enter GT.M from the shell, and uses ZEDIT to initiate an editing session on payroll.m Because ZEDIT defaults the extension to .m, it is not necessary to provide an extension. If payroll.m does not already exist, GT.M creates it in the first source directory identified by $ZROUTINES. If $ZROUTINES is null, ZEDIT places the source file in the process's current working directory.

$ZROUTINES is a read-write special variable containing an ordered list of directories that certain GT.M functions use to locate source and object files. Generally, a system manager sets up the environment to define the environment variable gtmroutines. At image invocation, GT.M initializes $ZROUTINES to the value of gtmroutines. Once you are running M, you can SET and refer to $ZROUTINES using the format:

```
GTM>SET $ZROUTINES=expr
```

Where:

- The expression may contain a list of UNIX directories and/or file-specifications delimited by spaces.

- The expression specifies one or more directories to search.

- An element of the expression contains an environment variable evaluating to a directory specification.

- If $ZROUTINES contains an environment variable that evaluates to a list, GT.M uses the first name in that list.

For more information on $ZROUTINES, see Chapter 8: "Intrinsic Special Variables" (page 291).

Editing from the Shell

To create and edit a source program from the shell, invoke any text editor at the shell prompt and specify a UNIX file as the source. The GT.M environment works best when you give a file the name of the M routine that it contains, and an .m extension.

Example:

```
$ vi payroll.m
```

The vi command initiates an editing session for payroll.m from the shell prompt. If payroll.m does not already exist, vi creates it. Because this example uses UNIX rather than GT.M tools, we must specify the .m file extension.

Compiling a Source Program

If you wish to focus on program development outside the GT.M environment, skip the next section and continue with the section "Compiling from the Shell".

GT.M compiles M source code files and produces object files for complete integration into the UNIX enviroment. The object modules have the same name as the compiled M source file with an .o file extension, unless otherwise specified. The object files contain machine instructions and information necessary to connect the routine with other routines, and map it into memory. An M routine source file must be compiled after it is created or modified. You can compile explicitly with the ZLINK command or implicitly with auto-ZLINK. At the shell command line, compile by issuing the mumps command.

The compiler checks M code for syntax errors and displays error messages on the terminal, when processing is complete. Each error message provides the source line in error with an indicator pointing to the place on the line where the error is occurring. For a list and description of the compiler error messages, refer to the GT.M Message and Recovery Procedures Reference Manual.

You can generate a listing file containing the compile results by including the -list qualifier as a modifier to the argument to the ZLINK command in Direct Mode. This can also be done by redirecting the compiler messages to a file by adding >filename 2>&1 to the mumps command when compiling a program from the shell. See "Compiling from the Shell" (page 42) for an explanation of the M command describing -list, and other valid qualifiers for the M and ZLINK commands.

The compiler stops processing a routine line when it detects an error on that line. Under most conditions the compiler continues processing the remaining routine lines. This allows the compiler to produce a more complete error analysis of the routine and to generate code that may have valid executable paths. The compiler does not report multiple syntax errors on the same line. When it detects more than 127 syntax errors in a source file, the compiler ceases to process the file.

Compiling from GT.M

In Direct Mode, GT.M provides access to the compiler explicitly through the ZLINK and ZCOMPILE commands, and implicitly through automatic invocation of ZLINK functionality (auto-ZLINK) to add required routines to the image. ZCOMPILE is a GT.M routine compilation command, it compiles the routine and creates a new object module. The primary task of ZLINK is to place the object code in memory and "connect" it with other routines. However, under certain circumstances, ZLINK may first use the GT.M compiler to create a new object module.

The difference between ZCOMPILE and ZLINK is that ZCOMPILE creates a new object module on compiling, whereas the ZLINK command links the object module with other routines and places the object code in memory.

ZLINK compiles under these circumstances:

*   ZLINK cannot locate a copy of the object module but can locate a copy of the source module.

*   ZLINK can locate both object and source module, and finds the object module to be older than the source module.

*   The file-specification portion of the ZLINK argument includes an explicit extension of .m.

Auto-ZLINK compiles under the first two circumstances, but can never encounter the last one.

When a command refers to an M routine that is not part of the current image, GT.M automatically attempts to ZLINK and, if necessary, compile that routine. In Direct Mode, the most common method to invoke the compiler through an auto-ZLINK is to enter DO ^routinename at the GTM> prompt. When the current image does not contain the routine, GT.M does the following:

*   Locates the source and object

*   Determines whether the source has been edited since it was last compiled

*   Compiles the routine, if appropriate

*   Adds the object to the image

By using the DO command, you implicitly instruct GT.M to compile, link, and execute the program. With this method, you can test your routine interactively.

For complete descriptions of ZLINK and auto-ZLINK, see Chapter 6: "Commands" (page 108) .

Example:

```
GTM>do ^payroll
GTM>do ^taxes
```

This uses the M DO command to invoke the GT.M compiler implicitly from the GTM> prompt if the routine requires new object code. When the compiler runs, it produces two object module files, payroll.o and taxes.o.

If you receive error messages from the compilation, you may fix them immediately by returning to the editor and correcting the source. By default, the GT.M compiler operates in "compile-as-written" mode, and produces object code even when a routine contains syntax errors. This code includes all lines that are correct and all commands on a line with an error, up to the error. Therefore, you may decide to tailor the debugging cycle by running the program without removing the syntax errors.

> ⚠️ **Caution**
>
> The DO command does not add an edited routine to the current image if the image already includes a routine matching the DO argument routine name. When the image contains a routine, GT.M simply executes the routine without examining whether a more recent version of the module exists. If you have a routine in your image, and you wish to change it, you must explicitly ZLINK that routine.

Example:

```
GTM>zlink "payroll"
GTM>zlink "taxes.m"
```

The first ZLINK compiles payroll.m if it cannot locate payroll, or if it finds that payroll.m has a more recent date/time stamp than payroll.o. The second ZLINK always compiles taxes.m producing a new taxes.o.

For more information on debugging in GT.M Direct Mode, see Chapter 4: "Operating and Debugging in Direct Mode" (page 50).

Compiling from the Shell

From the shell, invoke the compiler by entering mumps file-name at the shell prompt.

Example:

```
$ mumps payroll.m
$ mumps taxes.m
```

This uses the mumps command to invoke the GT.M compiler from the shell prompt, and creates .o versions of these files.

Use the mumps command at the shell prompt to:

- Check the syntax of a newly entered program.

- Optionally, get a formatted listing of the program.

- Ensure that all object code is up to date before linking.

The mumps command invokes the compiler to translate an M source file into object code.

The format for the MUMPS command is:

```
MUMPS [-qualifier[...]] pathname
```

- Source programs typically have an extension of .m. GT.M accepts other file extenions for explicit compilation commands, but not for auto-ZLINK.

- Each pathname identifies an M source program to compile. If you do not specify a .m file extension, the mumps command assumes the .m file extension.

- Qualifiers determine characteristics of the compiler output.

- Qualifiers must appear after the command, but before the file name to be properly applied.

- GT.M allows the UNIX * and ? wildcards in a file name.

- The MUMPS command returns a status of 1 after any error in compilation.

The * wildcard accepts any legal combination of numbers and characters including a null, in the position the wildcard holds.

The ? wildcard accepts exactly one legal character in its position.

For example, mumps *.m compiles all files in the current default directory with an .m extension. mumps *pay?.m compiles .m files with names that contain any characters followed by pay, followed by one character. Unlike when using ZLINK or ZCOMPILE, the filename must be fully specified when compiling from the shell.

> **⚠ Caution**
>
> When forming routine names, the compiler truncates object filenames to a maximum length of 31 characters. For example, for a source file called Adatabaseenginewithscalabilityproven.m the compiler generates an object file called Adatabaseenginewithscalabilityp.o. Ensure that the first 31 characters of source file names are unique.

Qualifiers for the mumps command

The mumps command allows qualifiers that customize the type and form of the compiler output to meet specific programming needs. MUMPS command qualifiers may also appear as a modifier to the argument to the GT.M ZLINK and ZCOMPILE commands. The following section describes the mumps command qualifiers. Make sure the arguments are specified ahead of file name and after the command itself.

-di[rect_mode]

Invokes a small GT.M image that immediately initiates Direct Mode.

-direct_mode does not invoke the M compiler.

The -direct_mode qualifier is incompatible with a file specification and with all other qualifiers.

-dy[namic_literals]

Compiles certain data structures associated with literals used in the source code in a way that they are dynamically loaded and unloaded from the object code. The dynamic loading and unloading of these data structures:

- Supersedes any specification of -NOINLINE_LITERALS.

- Reduces the amount of private memory required by each process which in turn allows more processes to execute with the same memory.

- In some circumstances, increases application performance by making more memory available for file system buffers.

- Increases the CPU and stack costs of local variable processing.

With no -DYNAMIC_LITERALS specified, these data structures continue to be generated when a routine is linked and stay in the private memory of each process. As the use of -DYNAMIC_LITERALS increases object code size, and as the dynamic loading and unloading only saves memory when the object code is in shared libraries, FIS recommends restricting the use of -DYNAMIC_LITERALS to only when compiling object code to be loaded into shared libraries or executed from an auto relink enabled directory.

-[no]embed_source

Instructs GT.M to embeds routine source code in the object code. The default is NOEMBED_SOURCE. Like other GT.M compilation qualifiers, this qualifier can be specified through the $ZCOMPILE intrinsic special variable and gtmcompile environment variable. EMBED_SOURCE provides $TEXT and ZPRINT access to the correct source code, even if the original M source file has been edited or removed. Where the source code is not embedded in the object code, GT.M attempts to locate the source code file. If it cannot find source code matching the object code, $TEXT() returns a null string. ZPRINT prints whatever source code found and also prints a TXTSRCMAT message in direct mode; if it cannot find a source file, ZPRINT issues a FILENOTFND error.

-[no]i[gnore]

Instructs the compiler to produce an object file even when the compiler detects errors in the source code (-ignore), or not to produce an object file when the compiler encounters an error (-noignore).

When used with the -noobject qualifier, the -ignore qualifier has no effect.

Execution of a routine that compiles with errors produces run-time errors when the execution path encounters any of the compile time errors.

This compile-as-written mode facilitates a flexible approach to debugging and expedites conversion to GT.M from an interpreted environment. Many M applications from an interpreted environment contain syntax abnormalities. This feature of compiling and later executing a routine provides the feel of developing in an interpreted environment.

By default, the compiler operates in -ignore mode and produces an object module even when the source routine contains errors.

-le[ngth]=lines

Controls the page length of the listing file.

The M compiler ignores the -length qualifier unless it appears with the -list qualifier.

By default, the listing has -length=66.

-[no]li[st][=filename]

Instructs the compiler to produce a source program listing file, and optionally specifies a name for the listing file. The listing file contains numbered source program text lines. When the routine has errors, the listing file also includes an error count, information about the location, and the cause of the errors.

When you do not specify a file name for the listing file, the compiler produces a listing file with the same name as the source file with a .lis file extension.

The -length and -space qualifiers modify the format and content of the listing file. The M compiler ignores these qualifiers unless the command includes the -list qualifier.

By default, the compiler operates -nolist and does not produce listings.

-noin[line_literals]

Compiles routines to use library code in order to load literals instead of generating in-line code thereby reducing the routine size. At the cost of a small increase in CPU, the use of -NOINLINE_LITERAL may help counteract growth in object size due to -DYNAMIC_LITERALS.

> ### ⚠️ Important
>
> Both -DYNAMIC_LITERALS and -NOINLINE_LITERNALS help optimize performance and virtual memory usage for applications whose source code includes literals. As the scalability and performance from reduced per-process memory usage may or may not compensate for the incremental cost of dynamically loading and unloading the data structures, and as the performance of routines vs. inline code can be affected by the availability of routines in cache, FIS suggests benchmarking to determine the combination of qualifiers best suited to each workload. Note that applications can freely mix routines compiled with different combinations of qualifiers.

-[no]o[bject][=filename]

Instructs the compiler to produce an output object file and optionally specifies a name for the object file using the optional filename argument.

When you do not specify a file name, the compiler produces an object file with the same file name as the source file and an .o file extension.

When forming routine names, the compiler truncates object filenames to a maximum length of 31 characters. For example, for a source file called Adatabaseenginewithscalabilityproven.m the compiler generates an object file called Adatabaseenginewithscalabilityp.o. Ensure that first 31 characters of source file names are unique.

The -noobject qualifier suppresses the production of an object file and is usually used with the -list qualifier to produce only a listing file.

Compilation with -OBJECT without -NAMEOFRTN implicitly names the first routine to match the name specified with -OBJECT.

By default, the compiler produces object modules.

-[n]ameofrtn=filename

Instructs the compiler to produce an output object file with the specified name. You can use -NAMEOFRTN and -OBJECT to create two object files with different names from the same .m source file.

-[no]w[arning]

Instructs the compiler to suppress error output; the default is -warning.

When used with the -list qualifier, the -nowarning qualifier does not affect errors in the listing.

> **Note**
>
> When used with the -noobject qualifier, the -nowarning qualifier instructs the compiler to produce no object with no indication of the fact or the cause of any errors.

-r[un]

Invokes GT.M in Autostart Mode.

The next argument is taken to be an M entryref. That routine is immediately executed, bypassing Direct Mode. Depending on the shell, you may need to put the entryref in quotation marks (""). This qualifier does not invoke the M compiler and is not compatible with any other qualifier.

-s[pace]=lines

Controls the spacing of the output in the listing file. -space=n specifies n-1 blank lines separating every source line in the listing file. If n<1, the M command uses single spacing in the listing.

If this qualifier appears without the -list qualifier, the M compiler ignores the -space qualifier.

By default, listings use single spaced output (-space=1).

MUMPS Command Qualifiers Summary

mumps Command Qualifiers	
QUALIFIER	Default
"-di[rect_mode]" (page 43)	N/A
"-dy[namic_literals]" (page 43)	N/A
"-[no]embed_source" (page 44)	-noembed_source
"-[no]i[gnore]" (page 44)	-ignore
"-le[ngth]=lines" (page 44)	-length=66
"-[no]li[st][=filename]" (page 44)	-nolist
"-noin[line_literals]" (page 45)	N/A
"-[n]ameofrtn=filename" (page 45)	N/A
"-[no]o[bject][=filename]" (page 45)	-object
"-r[un]" (page 46)	N/A
"-s[pace]=lines" (page 46)	-space=1

Executing a Source Program

M source programs can be executed either from the shell or from GT.M (Direct Mode).

Executing in Direct Mode

As discussed in the section on compiling source programs, the GT.M command ZLINK compiles the source code into an object module and adds the object module to the current image.

The run-time system also invokes auto-ZLINKing when an M command, in a program or in Direct Mode, refers to a routine that is not part of the current image.

M commands and functions that may initiate auto-ZLINKing are:

- DO

- GOTO

- ZBREAK

- ZGOTO

- ZPRINT

- $TEXT()

GT.M auto-ZLINKs the routine only under these conditions:

- The routine has the same name as the source file.

- ZLINK can locate the routine file using $ZROUTINES, or the current directory if $ZROUTINES is null.

$ZROUTINES is a read-write special variable that contains a directory search path used by ZLINK and auto-ZLINK to locate source and object files.

When the argument to a ZLINK command includes a pathname, $ZSOURCE maintains that pathname as a default for ZEDIT and ZLINK. $ZSOURCE is a read-write special variable.

Once you use the ZEDIT or ZLINK commands, $ZSOURCE can contain a partial file specification. The partial file specification can be a directory path (full or relative), a file name, and a file extension. You can set $ZSOURCE with an M SET command. A ZLINK without an argument is equivalent to ZLINK $ZSOURCE.

For additional information on $ZSOURCE and $ZROUTINES, refer to Chapter 8: "Intrinsic Special Variables" (page 291).

Example:

```
GTM>ZLINK "taxes"
```

If ZLINK finds taxes.m or taxes.o, the command adds the routine taxes to the current image. When ZLINK cannot locate taxes.o, or when it finds taxes.o is older than taxes.m, it compiles taxes.m, producing a new taxes.o. Then, it adds the contents of the new object file to the image.

Locating the Source File Directory

A ZLINK command that does not specify a directory uses $ZROUTINES to locate files. When $ZROUTINES is null, ZLINK uses the current directory. $ZROUTINES is initialized to the value of the gtmroutines environment variable.

When the file being linked includes an explicit directory, ZLINK and auto-ZLINK searches only that directory for both the object and the source files. If compilation is required, ZLINK places the new object file in the named directory.

A subsequent ZLINK searching for this object file will never find the object file in the specified directory unless the directory is added to the search path in $ZROUTINES, or the object file is moved to another directory already in the search path.

ZLINK cannot change a currently active routine, (e.g., a routine displayed in a ZSHOW "S" of the stack). ZLINK a currently active routine by first removing it from the M stack, using ZGOTO, or one or more QUITs. For additional information on the functionality of ZGOTO and ZSHOW, see their entries in Chapter 6: "Commands" (page 108).

To maintain compatibility with other editions of GT.M that do not permit the percent sign (%) in a file name, GT.M uses an underscore (_) in place of the percent in the file name.

Example:

```
GTM>zlink "_MGR"
```

This ZLINK links the M routine %MGR into the current image.

Executing from the Shell

You can run a program from the shell prompt using the following command:

```
$ mumps -run ^filename
```

The mumps command searches the directories specified by the environment variable gtmroutines to locate the specified file name.

Example:

```
$ mumps -run ^payroll
```

This executes a routine named payroll.

Processing Errors from Direct Mode and Shell

	Executing in Direct Mode	Executing from the Shell (mumps -run ^routine)
Usage	Suitable for development and debugging.	Suitable for production.
Error Handler	Not invoked for code entered at the direct mode prompt; Note that XECUTE code is treated as not entered at the direct mode prompt The default $ZTRAP="B" brings a process to the Direct Mode for debugging.	Errors are suppressed and cause a silent process exit. Set the environment variable gtm_etrap which overrides the default $ZTRAP="B". If needed, error handlers can include appropriate error notification to $PRINCIPAL. For example, the gtmprofile script sets a default $ETRAP value of **"Write:(0=$STACK) ""Error**

	Executing in Direct Mode	Executing from the Shell (mumps -run ^routine)
		occurred: "",$ZStatus,!" which you can customize to suit your needs.
stderr	GT.M processes send error messages to stderr only under the following conditions: • The error is fatal which means that the process is about to terminate • During compilation except of indirection or XECUTE • The process is about to enter direct mode due to a BREAK command • The erroneous code was entered at the direct mode prompt • The message is informational, and intended for the user, which is unusual	

For more information, see Chapter 13: "Error Processing" (page 562).

Chapter 4. Operating and Debugging in Direct Mode

Revision History		
Revision V7.1-004	27 June 2024	• In "Using ZSHOW to Examine Context Information" (page 61), Added WFR,BUS,BTS,STG,KTG,ZTG,DEXA,GLB,JNL,MLK,PI\nMLBA,TRGA,WRL,PRG,WFL,WHE,INC to the ZSHOW output.
Revision V7.1-002	19 September 2023	• In "Using ZSHOW to Examine Context Information" (page 61), Add the new $ZICUVER ISV
Revision V7.1-001	26 June 2023	• In "Summary of GT.M Debugging Tools" (page 66), add ZLINK and auto-ZLINK to the $ZCSTATUS table entry
Revision V6.3-007	04 February 2019	• In "Using ZSHOW to Examine Context Information" (page 61), add $ZAUDIT and #ZTIMEOUT
Revision V6.3-005	29 June 2018	• In "Correcting Errors in an M Routine" (page 64), fix formatting errors in the example • In "Creating and Displaying M Routines" (page 55), correct the dmex.m exmaple so that only the undefined variable name error and the infinite for loop issue remains. • In "Using ZSHOW to Examine Context Information" (page 61), update the ZSHOW "*" output for V6.3-005.
Revision V6.3-003	12 December 2017	• In "Using ZSHOW to Examine Context Information" (page 61), remove extraneous whitespace
Revision V6.2-001	27 February 2015	In "Line Editing" (page 53), added GT.M behavior for the Delete key and an example to modify terminfo capabilities.

Direct Mode is an important tool in GT.M because it allows you to interactively debug, modify, and execute M routines. Direct Mode is a shell that immediately compiles and executes GT.M commands providing an interpretive-like interface. M simplifies debugging by using the same commands for debugging that are used for programming.

The focus of this chapter is to describe the debugging process in Direct Mode, and to illustrate the GT.M language extensions that enhance the process. Command functionality is described only in enough detail to illustrate why a particular command is useful for a debugging activity being described. If you have specific functionality questions about a command or variable, see the "Commands" [108], "Functions" [212], or "Intrinsic Special Variables" [291] chapter.

It is also from Direct Mode that you activate GT.M tools used to create M source code. The interaction of M commands used for editing and compiling is described in greater detail within Chapter 3: "Development Cycle" (page 32).

Operating in Direct Mode

This section provides an overview of the following basic operational issues in Direct Mode:

- Entering Direct Mode

- Available functionality

- Exiting Direct Mode

Entering Direct Mode

To enter Direct Mode, type $gtm_dist/mumps -direct at the shell prompt.

```
$ $gtm_dist/mumps -direct
GTM>
```

This shows using $gtm_dist/mumps -direct at the prompt to enter Direct Mode.

To create a gtm alias in your shell startup file (in the example below the startup file is assumed to be a .profile file):

1. Open an edition session for your .profile file by typing:

    ```
    $vi .profile
    ```

2. Add a function to the file to define your gtm alias:

    ```
    gtm(){ $gtm_dist/mumps -direct}
    ```

3. Save the file.

Now, when you want to enter Direct Mode for an editing or debugging session, simply type gtm at the shell prompt.

Example:

```
$ gtm
GTM>
```

This shows that the gtm alias typed at the shell prompt also takes you to the Direct Mode.

Functionality Available in Direct Mode

This section provides an overview of basic functionality and concepts that enhance your use of Direct Mode.

Command Recall

Direct Mode includes a line command recall function to display previously entered command lines. Use <CTRL-B> or the Up Arrow key at the GTM> prompt to scroll back through command lines. Use the Down Arrow key to scroll forward through the command lines. GT.M displays one command line at a time.You may delete and reenter characters starting at the end of a recalled line.

The RECALL command is another way to access previously entered Direct Mode command lines. RECALL is only valid in Direct Mode and causes an error if it appears in other M code.

The format of the RECALL command is:

```
REC[ALL] [intlit|strlit]
```

- The optional integer literal specifies a previously entered command by the counting back from the present.

- The optional string literal specifies the most recently entered command line that starts with characters matching the (case-sensitive) literal.

- When the RECALL command has no argument, it displays up to a maximum of 99 available past Direct Mode entries.

If the Direct Mode session has just started, you may not have entered 99 lines for GT.M to save and therefore you will not have 99 lines to look at. The most recently entered GT.M command line has the number one (1), older lines have higher numbers. GT.M does not include the RECALL command in the listing. If the RECALL command is issued from a location other than the Direct Mode prompt, GT.M issues a run-time error.

Example:

```
GTM>write $zgbldir
/usr/lib/fis-gtm/V5.4-002B_x86/mumps.gld
GTM>set $zgbldir="test.gld"
GTM>set a=10
GTM>set b=a
GTM>recall
1 set b=a
2 set a=10
3 set $zgbldir="test.gld"
4 write $zgbldir
GTM>
```

This REC[ALL] command displays the previously entered commands.

You can also display a selected command by entering RECALL and the line number of the command you want to retrieve.

Example:

```
GTM>recall 2
GTM>set a=10
```

This RECALLs the line number two (2).

If the RE[CALL] command includes a text parameter, GT.M displays the most recent command matching the text after the RE[CALL] command.

Example:

```
GTM>recall write
GTM>write $zgbldir
```

This RECALLs "WRITE", the command most recently beginning with this text. Note that the RECALL command text is case sensitive. The RECALL command with a text argument treats WRITE and write differently, that is, it treats them case sensitively. If you first type the WRITE command in lower-case and then type WRITE in upper-case to recall it, the RECALL command does not find a match.

Line Editing

GT.M permits the use of the GT.M command line editor at the Direct Mode prompt and during M READs from a terminal. The GT.M line editor allows cursor positioning using the <CTRL> key, edit keypad and function keys.

The GT.M Direct Mode line editing keys are as follows:

Backspace: Deletes the character to the left of the cursor

Delete: Deletes the character under the cursor

Up-arrow: Moves to a less recent item in the RECALL list

Down-arrow: Moves to a more recent item in the RECALL list

Left-arrow: Moves the cursor one character to the left

Right-arrow: Moves the cursor one character to the right

<CTRL-A>: Moves the cursor to the beginning of the line

<CTRL-B>: Moves the cursor one character towards the beginning of the line

<CTRL-D>: On an empty line, terminates GT.M and returns control to the shell.

<CTRL-E>: Moves the cursor to the end of the line

<CTRL-F>: Moves the cursor one character towards the end of the line

<CTRL-K>: Deletes all characters from the cursor to the end of the line

<CTRL-U>: Deletes the entire line

Note

When entering commands at the direct mode prompt, the insert mode can be toggled for that line by using the insert key. When GT.M starts, insert mode is enabled unless the value of the gtm_principal_editing environment variable includes the string NOINSERT. If insert mode is disabled or enabled for the $PRINCIPAL device by an USE statement before returning to direct mode, it will remain disabled or enabled at direct mode. The insert mode can be toggled within a direct mode line using the terminal's INSERT key.

Important

GT.M deletes the character under the cursor when you press the key on the keyboard that sends the escape sequence which maps to the kdch1 capability in your current terminfo entry (by convention, the Delete key). If the current terminfo entry is missing the kdch1 capability, GT.M uses a default value derived from members of the DEC VT terminal family, as it does for selected other missing terminfo capabilities. If you wish the Backspace and Delete keys to behave the same, the simplest way is to configure your terminal emulator to send the same character sequences for the Delete key that it does for the Backspace key. You can alternatively modify your terminfo setting: for example, create an editable version of your terminfo entry in a temporary file with a command such as: **infocmp > /tmp/$$_$TERM** and edit the temporary file to replace the entry for the kbs capability with the one in the kdch1 capability. Save your changes, and compile the edited file into a usable terminfo entry, for example:

```
export TERMINFO=$HOME/.terminfo # You may need to add this to your login profile
```

```
profilemkdir -p $TERMINFO
tic /tmp/$$_$TERM # or whatever your temporary file name was
```

When modifying terminfo capabilities, always look for unintended changes in the behavior of other applications, for example, text editors, that also rely on those capabilities. In the worst case, you may need to toggle between alternate terminfo entries for GT.M and other applications while you evaluate different options. Also, for terminfo entries without the cud1 capability, GT.M uses a linefeed when moving to the next line in direct mode.

The M Invocation Stack

The ANSI M Standard describes certain M operations in terms of how a stack-based virtual machine would operate. A stack is a repository for tracking temporary information on a "last-in/first-out" (LIFO) basis. M program behavior can be understood using a stack-based model. However, the standard is not explicit in defining how an implementation must maintain a stack or even whether it must use one at all.

The stack model provides a trail of routines currently in progress that shows the location of all the M operations that performed the invocations leading to the current point.

The ZSHOW command makes this stack information available within GT.M. For more information, see "Using the Invocation Stack in Debugging" (page 59) in this chapter, and the command description at "ZSHow" (page 193).

Exiting Direct Mode

Five M commands can terminate a Direct Mode session:

• HALT

• ZHALT

• ZCONTINUE

• GOTO

• ZGOTO

The HALT command exits Direct Mode and terminates the M process.

The ZHALT command exits Direct Mode and returns the exit status to the calling environment.

The ZCONTINUE command instructs GT.M to exit Direct Mode and resume routine execution at the current point in the M invocation stack. This may be the point where GT.M interrupted execution and entered Direct Mode. However, when the Direct Mode interaction includes a QUIT command, it modifies the invocation stack and causes ZCONTINUE to resume execution at another point.

The GOTO and ZGOTO commands instruct GT.M to leave Direct Mode, and transfer control to a specified entry reference.

Debugging a Routine in Direct Mode

To begin a debugging session on a specific routine, type the following command at the GTM prompt:

```
GTM>DO ^routinename
```

You can also begin a debugging session by pressing <CTRL-C> after running an M application at the shell. To invoke Direct Mode by pressing <CTRL-C>, process must have the Principal Device in the CENABLE state and not have the device set to CTRAP=$C(3).

When GT.M receives a <CTRL-C> command from the principal device, it invokes Direct Mode at the next opportunity, (usually at a point corresponding to the beginning of the next source line). GT.M can also interrupt at a FOR loop iteration or during a command of indeterminate duration such as LOCK, OPEN or READ. The GT.M USE command enables/disables the <CTRL-C> interrupt with the [NO]CENABLE deviceparameter. By default, GT.M starts <CTRL-C> enabled. The default setting for <CTRL-C> is controlled by $gtm_nocenable which controls whether <CTRL-C> is enabled at process startup. If $gtm_nocenable has a value of 1, "TRUE" or "YES" (case-insensitive), and the process principal device is a terminal, $PRINCIPAL is initialized to a NOCENABLE state where the process does not recognize <CTRL-C> as a signal to enter direct mode. No value, or other values of $gtm_nocenable initialize $PRINCIPAL with the CENABLE state. The [NO]CENABLE deviceparameter on a USE command can still control this characteristic from within the process.

GT.M displays the GTM> prompt on the principal device. Direct Mode accepts commands from, and reports errors to, the principal device. GT.M uses the current device for all other I/O. If the current device does not match the principal device when GT.M enters Direct Mode, GT.M issues a warning message on the principal device. A USE command changes the current device. For more information on the USE command, see Chapter 9: "Input/Output Processing" (page 340).

The default "compile-as-written" mode of the GT.M compiler lets you run a program with errors as part of the debugging cycle. The object code produced includes all lines that are correct and all commands on a line with an error, up to the error. When GT.M encounters an error, it XECUTEs non empty values of $ETRAP or $ZTRAP. By default $ZTRAP contains a BREAK command, so GT.M enters Direct Mode.

The rest of the chapter illustrates the debugging capabilities of GT.M by taking a sample routine, dmex, through the debugging process. dmex is intended to read and edit a name, print the last and first name, and terminate if the name is an upper-case or lower-case "Q".

Each of the remaining sections of the chapter uses dmex to illustrate an aspect of the debugging process in GT.M.

Creating and Displaying M Routines

To create or edit a routine, use the ZEDIT command. ZEDIT invokes the editor specified by the EDITOR environment variable, and opens the specified file. dmex.m, for editing.

Example:

```
GTM>ZEDIT "dmex"
```

Once in the editor, use the standard editing commands to enter and edit text. When you finish editing, save the changes, which returns you to Direct Mode.

To display M source code for dmex, use the ZPRINT command.

Example:

```
GTM>ZPRINT ^dmex
dmex;dmex - Direct Mode example
;
beg
    for  read !,"Name: ",name do name
    quit
name
    set ln=$l(name)
```

```
   if ln,$extract("QUIT",1,ln)=$tr(name,"quit","QUIT") do
   . s name="Q"
   . quit
   if ln<30,bame?1.a.1"-".a1","1" "1a.ap do print quit
   write !,"Please use last-name, "
   write "first-name middle-initial or 'Q' to Quit."
   quit
print
   write !,$piece(name,", ",2)," ",$piece(name,", ")
   quit
GTM>
```

This uses the ZPRINT command to display the routine dmex.

> **Note**
>
> The example misspells the variable name as bame.

Executing M Routines Interactively

To execute an M routine interactively, it is not necessary to explicitly compile and link your program. When you refer to an M routine that is not part of the current image, GT.M automatically attempts to compile and ZLINK the program.

Example:

```
GTM>DO ^dmex
Name: Revere, Paul
%GTM-E-UNDEF, Undefined local variable: bame
At M source location name+3^dmex
GTM>
```

In this example GT.M places you in Direct Mode, but also cites an error found in the program with a run-time error message. In this example, it was a reference to bame, which is undefined.

To see additional information about the error message, examine the $ECODE or $ZSTATUS special variables.

$ECODE is read-write intrinsic special variable that maintains a list of comma delimited codes that describe a history of past errors - the most recent ones appear at the end of the list. In $ECODE, standard errors are prefixed with an "M", user defined errors with a "U", and GT.M errors with a "Z". A GT.M code always follows a standard code.

$ZSTATUS is a read-write intrinsic special variable that maintains a string containing the error condition code and location of the last exception condition occurring during routine execution. GT.M updates $ZSTATUS only for errors found in routines and not for errors entered at the Direct Mode prompt.

> **Note**
>
> For more information on $ECODE and $STATUS see Chapter 8: "Intrinsic Special Variables" (page 291).

Example:

```
GTM>WRITE $ECODE
,M6,Z150373850
```

This example uses a WRITE command to display $ECODE.

Example:

```
GTM>WRITE $ZS
150373850,name+3^dmex,%GTM-E-UNDEF, Undefined
local variable: bame
```

This example uses a WRITE command to display $ZSTATUS. Note that the $ZSTATUS code is the same as the "Z" code in $ECODE.

You can record the error message number, and use the $ZMESSAGE function later to re-display the error message text.

Example:

```
GTM>WRITE $ZM(150373850)
%GTM-E-UNDEF, Undefined local variable: !AD
```

This example uses a WRITE command and the $ZMESSAGE function to display the error message generated in the previous example. $ZMESSAGE() is useful when you have a routine that produces several error messages that you may want to examine later. The error message reprinted using $ZMESSAGE() is generic; therefore, the code !AD appears instead of the specific undefined local variable displayed with the original message.

Processing with Run-time and Syntax Errors

When GT.M encounters a run-time or syntax error, it stops executing and displays an error message. GT.M reports the error in the message. In this case, GT.M reports an undefined local variable and the line in error, name+3^dmex. Note that GT.M re-displays the GTM> prompt so that debugging may continue.

To re-display the line and identify the error, use the ZPRINT command.

Example:

```
GTM>ZPRINT, name+3
%GTM-E-SPOREOL, Either a space or an end-of-line was expected but not found
ZP, name+3
^_____
GTM>
```

This example shows the result of incorrectly entering a ZPRINT command in Direct Mode. GT.M reports the location of the syntax error in the command line with an arrow. $ECODE and $ZSTATUS do not maintain this error message because GT.M did not produce the message during routine execution. Enter the correct syntax, (i.e., remove the comma) to re-display the routine line in error.

Example:

```
GTM>WRITE $ZPOS
name+3^dmex
```

This example writes the current line position.

$ZPOSITION is a read-only GT.M special variable that provides another tool for locating and displaying the current line. It contains the current entry reference as a character string in the format label+offset^routine, where the label is the closest preceding label. The current entry reference appears at the top of the M invocation stack, which can also be displayed with a ZSHOW "S" command.

To display the current value of every local variable defined, use the ZWRITE command with no arguments.

Example:

```
GTM>ZWRITE
ln=12
name="Revere, Paul"
```

This ZWRITE displays a listing of all the local variables currently defined.

> **Note**
>
> ZWRITE displays the variable name. ZWRITE does not display a value for bame, confirming that it is not defined.

Correcting Errors

Use the ZBREAK command to establish a temporary breakpoint and specify an action. ZBREAK sets or clears routine-transparent breakpoints during debugging. This command simplifies debugging by interrupting execution at a specific point to examine variables, execute commands, or to start using ZSTEP to execute the routine line by line.

GT.M suspends execution during execution when the entry reference specified by ZBREAK is encountered. If the ZBREAK does not specify an expression "action", the process uses the default, BREAK, and puts GT.M into Direct Mode. If the ZBREAK does specify an expression "action", the process XECUTEs the value of "action", and does not enter Direct Mode unless the action includes a BREAK. The action serves as a "trace-point". The trace-point is silent unless the action specifies terminal output.

Example:

```
GTM>ZBREAK name+3^dmex:"set bame=name"
```

This uses a ZBREAK with an action that SETs the variable bame equal to name.

Stepping Through a Routine

The ZSTEP command provides a powerful tool to direct GT.M execution. When you issue a ZSTEP from Direct Mode, GT.M executes the program to the beginning of the next target line and performs the ZSTEP action.

The optional keyword portion of the argument specifies the class of lines where ZSTEP pauses its execution, and XECUTEs the ZSTEP action specified by the optional action portion of the ZSTEP argument. If the action is specified, it must be an expression that evaluates to valid GT.M code. If no action is specified, ZSTEP XECUTEs the code specified by the $ZSTEP intrinsic special variable; by default $ZSTEP has the value "B", which causes GT.M to enter Direct Mode.

ZSTEP actions, that include commands followed by a BREAK, perform the specified action, then enter Direct Mode. ZSTEP actions that do not include a BREAK perform the command action and continue execution. Use ZSTEP actions that issue conditional BREAKs and subsequent ZSTEPs to perform tasks such as test for changes in the value of a variable.

Use ZSTEP to incrementally execute a routine or a series of routines. Execute any GT.M command from Direct Mode at any ZSTEP pause. To resume normal execution, use ZCONTINUE. Note that ZSTEP arguments are keywords rather than expressions, and they do not allow indirection.

Example:

```
GTM>ZSTEP INTO
Break instruction encountered during ZSTEP action
At M source location print^dmex
```

```
GTM>ZSTEP OUTOF
Paul Revere
Name: Q
%GTM-I-BREAKZST, Break instruction encountered during ZSTEP action
At M source location name^dmex
GTM>ZSTEP OVER
Break instruction encountered during ZSTEP action
At M source location name+1^dmex
```

This example shows using the ZSTEP command to step through the routine dmex, starting where execution was interrupted by the undefined variable error. The ZSTEP INTO command executes line name+3 and interrupts execution at the beginning of line print.

The ZSTEP OUTOF continues execution until line name. The ZSTEP OVER, which is the default, executes until it encounters the next line at this level on the M invocation stack. In this case, the next line is name+1. The ZSTEP OVER could be replaced with a ZSTEP with no argument because they do the same thing.

Continuing Execution From a Breakpoint

Use the ZCONTINUE command to continue execution from the breakpoint.

Example:

```
GTM>ZCONTINUE
Paul Revere
Name: q
Name: QUIT
Name: ?
Please use last-name, first name middle-initial
or 'Q' to Quit.
Name:
```

This uses a ZCONTINUE command to resume execution from the point where it was interrupted. As a result of the ZBREAK action, bame is defined and the error does not occur again. Because the process does not terminate as intended when the name read has q as a value, we need to continue debugging.

Interrupting Execution

Press <CTRL-C> to interrupt execution, and return to the GTM prompt to continue debugging the program.

Example:

```
%GTM-I-CTRLC, CTRLC_C encountered.
GTM>
```

This invokes direct mode with a <CTRL-C>.

Using the Invocation Stack in Debugging

M DOs, XECUTEs, and extrinsics add a level to the invocation stack. Matching QUITs take a level off the stack. When GT.M executes either of these commands, an extrinsic function, or an extrinsic special variable, it "pushes" information about the new environment on the stack. When GT.M executes the QUIT, it "pops" the information about the discarded environment off the stack. It then reinstates the invoking routine information using the entries that have now arrived at the active end of the stack.

> **Note**
>
> In the M stack model, a FOR command does not add a stack frame, and a QUIT that terminates a FOR loop does not remove a stack frame.

Determining Levels of Nesting

$STACK contains an integer value indicating the "level of nesting" caused by DO commands, XECUTE commands, and extrinsic functions in the M virtual stack.

$STACK has an initial value of zero (0), and increments by one with each DO, XECUTE, or extrinsic function. Any QUIT that does not terminate a FOR loop or any ZGOTO command decrements $STACK. In accordance with the M standard, a FOR command does not increase $STACK. M routines cannot modify $STACK with the SET or KILL commands.

Example:

```
GTM>WRITE $STACK
2
GTM>WRITE $ZLEVEL
3
GTM>
```

This example shows the current values for $STACK and $ZLEVEL. $ZLEVEL is like $STACK except that uses one (1) as the starting level for the M stack, which $STACK uses zero (0), which means that $ZLEVEL is always one more than $STACK. Using $ZLEVEL with "Z" commands and functions, and $STACK with standard functions avoids the need to calculate the adjustment.

Looking at the Invocation Stack

The $STACK intrinsic special variable and the $STACK() function provide a mechanism to access M stack context information.

Example:

```
GTM>WRITE $STACK
2
GTM>WRITE $STACK(2,"ecode")
,M6,Z150373850,
GTM>WRITE $STACK(2,"place")
name+3^dmex
GTM>WRITE $STACK(2,"mcode")
if ln<30,bame?1.a.1"-".a1","1" "1a.ap do print q
GTM>
```

This example gets the value of $STACK and then uses that value to get various types of information about that stack level using the $STACK() function. The "ecode" value of the error information for level two, "place" is similar to $ZPOSITION, "mcode" is the code for the level.

In addition to the $STACK intrinsic special variable, which provides the current stack level, $STACK(-1) gives the highest level for which $STACK() can return valid information. Until there is an error $STACK and $STACK(-1) are the same, but once $ECODE shows that there is an "current" error, the information returned by $STACK() is frozen to capture the state at the time of the error; it unfreezes after a SET $ECODE="".

Example:

```
GTM>WRITE $STACK
2
GTM>WRITE $STACK(-1)
2
GTM>
```

This example shows that under the conditions created (in the above example), $STACK and $STACK(-1) have the same value.

The $STACK() can return information about lower levels.

Example:

```
+1^GTM$DMOD
GTM>WRITE $STACK(1,"ecode")
GTM>WRITE $STACK(1,"place")
beg^dmex
GTM>WRITE $STACK(1,"mcode")
beg for read !,"Name:",namde do name
GTM>
```

This example shows that there was no error at $STACK level one, as well as the "place" and "mcode" information for that level.

Using ZSHOW to Examine Context Information

The ZSHOW command displays information about the M environment.

Example:

```
GTM>zshow "*"
$DEVICE=""
$ECODE=",M6,Z150373850,"
$ESTACK=2
$ETRAP=""
$HOROLOG="64813,21971"
$IO="/dev/pts/0"
$JOB=14550
$KEY=$C(13)
$PRINCIPAL="/dev/pts/0"
$QUIT=0
$REFERENCE=""
$STACK=2
$STORAGE=2147483647
$SYSTEM="47,gtm_sysid"
$TEST=1
$TLEVEL=0
$TRESTART=0
$X=0
$Y=26
$ZA=0
$ZALLOCSTOR=680360
$ZAUDIT=0
$ZB=$C(13)
$ZCHSET="M"
$ZCLOSE=0
$ZCMDLINE=""
$ZCOMPILE=""
$ZCSTATUS=0
```

```
$ZDATEFORM=0
$ZDIRECTORY="/path/to/the/current/directory"
$ZEDITOR=0
$ZEOF=0
$ZERROR="Unprocessed $ZERROR, see $ZSTATUS"
$ZGBLDIR="/path/to/the/global/directory"
$ZHOROLOG="64813,21971,720675,14400"
$ZICUVER=""
$ZININTERRUPT=0
$ZINTERRUPT="IF $ZJOBEXAM()"
$ZIO="/dev/pts/0"
$ZJOB=0
$ZKEY=""
$ZLEVEL=3
$ZMAXTPTIME=0
$ZMODE="INTERACTIVE"
$ZONLNRLBK=0
$ZPATNUMERIC="M"
$ZPIN="/dev/pts/0"
$ZPOSITION="name+5^dmex"
$ZPOUT="/dev/pts/0"
$ZPROMPT="GTM>"
$ZQUIT=0
$ZREALSTOR=697936
$ZRELDATE="20180614 00:33"
$ZROUTINES=". /usr/lib/fis-gtm/V6.3-005_x86_64 /usr/lib/fis-gtm/V6.3-005_x86_64/plugin/o(/usr/lib/fis-gtm/V6.3-005_x86_64/
plugin/r)"
$ZSOURCE=""
$ZSTATUS="150373850,name+5^dmex,%GTM-E-UNDEF, Undefined local variable: bame"
$ZSTEP="B"
$ZSTRPLLIM=0
$ZSYSTEM=0
$ZTIMEOUT=-1
$ZTDATA=0
$ZTDELIM=""
$ZTEXIT=""
$ZTLEVEL=0
$ZTNAME=""
$ZTOLDVAL=""
$ZTRAP="B"
$ZTRIGGEROP=""
$ZTSLATE=""
$ZTUPDATE=""
$ZTVALUE=""
$ZTWORMHOLE=""
$ZUSEDSTOR=671689
$ZUT=1528970771720738
$ZVERSION="GT.M V6.3-005 Linux x86_64"
$ZYERROR=""
ln=8
name="John Doe"
/dev/pts/0 OPEN TERMINAL NOPAST NOESCA NOREADS TYPE WIDTH=165 LENG=48
MLG:0,MLT:0
GLD:*,REG:*,SET:0,KIL:0,GET:0,DTA:0,ORD:0,ZPR:0,QRY:0,LKS:0,LKF:0,CTN:0,DRD:0
DWT:0,NTW:0,NTR:0,NBW:0,NBR:0,NR0:0,NR1:0,NR2:0,NR3:0,TTW:0,TTR:0,TRB:0,TBW:0,
TBR:0,TR0:0,TR1:0,TR2:0,TR3:0,TR4:0,TC0:0,TC1:0,TC2:0,TC3:0,TC4:0,ZTR:0,DFL:0,
DFS:0,JFL:0,JFS:0,JBB:0,JFB:0,JFW:0,JRL:0,JRP:0,JRE:0,JRI:0,JRO:0,JEX:0,DEX:0,
CAT:0,CFE:0,CFS:0,CFT:0,CQS:0,CQT:0,CYS:0,CYT:0,BTD:0,WFR:0,BUS:0,BTS:0,STG:0,
KTG:0,ZTG:0,DEXA:0,GLB:0,JNL:0,MLK:0,PRC:0,TRX:0,ZAD:0,JOPA:0,AFRA:0,BREA:0,
MLBA:0,TRGA:0,WRL:0,PRG:0,WFL:0,WHE:0,INC:0
name+5^dmex      ($ZTRAP)
```

```
    (Direct mode)
beg+1^dmex:51a6a6c4739b004094c4545246ce4d68
+1^GTM$DMOD     (Direct mode)
GTM>
```

This example uses the asterisk (*) argument to show all information that ZSHOW offers in the context debugging the error in the ^dmex routine. First are the Intrinsic Special Variables ($DEVICE-$ZYERROR, also available with ZSHOW "I"), then the local variables (bame, ln and name, also available with ZSHOW "V"), then the ZBREAK locations (name+3^dmex, also available with ZSHOW "B"), then the device information (also available with ZSHOW "D"), then the M stack (also available with ZSHOW "S"). ZSHOW "S" is the default for ZSHOW with no arguments.

Context information that does not exist in this example includes M LOCKs of this process (ZSHOW "L").

In addition to directing its output to the current device, ZSHOW can place its output in a local or global variable array. For more information, see the command description "ZSHow" (page 193).

> **Note**
>
> ZSHOW "V" produces the same output as ZWRITE with no arguments, but ZSHOW "V" can be directed to a variable as well as a device.

Transferring Routine Control

The ZGOTO command transfers control from one part of the routine to another, or from one routine to another, using the specified entry reference. The ZGOTO command takes an optional integer expression that indicates the M stack level reached by performing the ZGOTO, and an optional entry reference specifying the location to where ZGOTO transfers control. A ZGOTO command, with an entry reference, performs a function similar to the GOTO command with the additional capability of reducing the M stack level. In a single operation, the process executes $ZLEVEL-intexpr, implicit QUITs from DO or extrinsic operations, and a GOTO operation transferring control to the named entry reference.

The ZGOTO command leaves the invocation stack at the level of the value of the integer expression. GT.M implicitly terminates any intervening FOR loops and unstacks variables stacked with NEW commands, as appropriate.

ZGOTO $ZLEVEL:LABEL^ROUTINE takes the same action as GO LABEL^ROUTINE.

ZGOTO $ZLEVEL-1 produces the same result as QUIT (followed by ZCONTINUE, if in Direct Mode).

If the integer expression evaluates to a value greater than the current value of $ZLEVEL, or less than zero (0), GT.M issues a run-time error.

If ZGOTO has no entry reference, it performs some number of implicit QUITs and transfers control to the next command at the specified level. When no argument is specified, ZGOTO 1 is the result, and operation resumes at the lowest level M routine as displayed by ZSHOW "S". In the image invokedby mumps -direct, or a similar image, a ZGOTO without arguments returns the process to Direct Mode.

Displaying Source Code

Use the ZPRINT command to display source code lines selected by its argument. ZPRINT allows you to display the source for the current routine and any other related routines. Use the ZPRINT command to display the last call level.

Example:

```
GTM>ZPRINT beg
```

```
beg for read !,"Name: ",name do name
```

This example uses a ZPRINT command to print the line indicated as the call at the top of the stack. Notice that the routine has an error in logic. The line starting with the label beg has a FOR loop with no control variable, no QUIT, and no GOTO. There is no way out of the FOR loop.

Correcting Errors in an M Routine

Now that the routine errors have been identified, correct them in the M source file. Use ZEDIT to invoke the editor and open the file for editing. Correct the errors previously identified and enter to exit the editor.

Example:

```
GTM>zedit "dmex.m"
dmex;dmex - Direct Mode example
;
beg
    for  read !,"Name: ",name do name quit:name="Q"
    quit
name
    set ln=$length(name)
    if ln,$extract("QUIT",1,ln)=$tr(name,"quit","QUIT") do
    . set name="Q"
    . quit
    if ln<30,name?1.a.1"-".a1","1" "1a.ap do print quit
    write !,"Please use last-name, "
    write "first-name middle-initial or 'Q' to Quit."
    quit
print
    write !,$piece(name,", ",2)," ",$piece(name,", ")
    quit
```

This example shows the final state of a ZEDIT session of dmex.m. Note that the infinite FOR loop at line beg is corrected.

Relinking the Edited Routine

Use the ZLINK command to add the edited routine to the current image. ZLINK automatically recompiles and relinks the routine. If the routine was the most recent one ZEDITed or ZLINKed, you do not have to specify the routine name with the ZLINK command.

⚠ Caution

When you issue a DO command, GT.M determines whether the routine is part of the current image, and whether compiling or linking is necessary. Because this routine is already part of the current image, GT.M does not recompile or relink the edited version of the routine if you run the routine again without ZLINKing it first. Therefore, GT.M executes the previous routine image and not the edited routine.

Note

You may have to issue a ZGOTO or a QUIT command to remove the unedited version of the routine from the M invocation stack before ZLINKing the edited version.

Example:

```
GTM>ZLINK
 Cannot ZLINK an active routine
```

This illustrates a GT.M error report caused by an attempt to ZLINK a routine that is part of the current invocation stack.

To ZLINK the routine, remove any invocation levels for the routine off of the call stack. You may use the ZSHOW "S" command to display the current state of the call stack. Use the QUIT command to remove one level at a time from the call stack. Use the ZGOTO command to remove multiple levels off of the call stack.

Example:

```
GTM>ZSHOW "S"
name+3^dmex ($ZTRAP) (Direct mode)
beg^dmex (Direct mode)
 ^GTM$DMOD (Direct mode)
GTM>ZGOTO
GTM>ZSHOW "S"
 ^GTM$DMOD (Direct mode)
GTM>ZLINK
```

This example uses a ZSHOW "S" command to display the current state of the call stack. A ZGOTO command without an argument removes all the calling levels above the first from the stack. The ZLINK automatically recompiles and relinks the routine, thereby adding the edited routine to the current image.

Re-executing the Routine

Re-display the DO command using the RECALL command.

Execute the routine using the DO command.

Example:

```
GTM>D ^dmex
Name: Revere, Paul
Paul Revere
Name: q
```

This example illustrates a successful execution of dmex.

Using Forked Processes

The ZSYSTEM command creates a new process called the child process, and passes its argument to the shell for execution. The new process executes in the same directory as the initiating process. The new process has the same operating system environment, such as environment variables and input/output devices, as the initiating process. The initiating process pauses until the new process completes before continuing execution.

Example:

```
GTM>ZSYSTEM
$ ls dmex.*
dmex.m dmex.o
$ ps
```

```
PID TTY TIME COMMAND
7946 ttyp0 0:01 sh
7953 ttyp0 0:00 gtm
7955 ttyp0 0:00 ps
$ exit
GTM>
```

This example uses ZSYSTEM to create a child process, perform some shell actions, and return to GT.M.

Summary of GT.M Debugging Tools

The following table summarizes GT.M commands, functions, and intrinsic special variables available for debugging. For more information on these commands, functions, and special variables, see the "Commands" [108], "Functions" [212], and "Intrinsic Special Variables" [291] chapters.

For more information on syntax and run-time errors during Direct Mode, see Chapter 13: "Error Processing" (page 562).

GT.M Debugging Tools	
EXTENSION	**EXPLANATION**
$ECode	Contains a list of errors since it was last cleared
$STack	Contains the current level of DO/XECUTE nesting from a base of zero (0).
$STack()	Returns information about the M virtual stack context, most of which freezes when an error changes $ECODE from the empty string to a list value.
ZBreak	Establishes a temporary breakpoint, with optional count and M action.
ZCOMpile	Invokes the GT.M compiler without a corresponding ZLINK.
ZContinue	Continues routine execution from a break.
ZEDit	Invokes the UNIX text editor specified by the EDITOR environment variable.
ZGoto	Removes zero or more levels from the M invocation stack and transfers control.
ZLink	Includes a new or modified M routine in the current M image; automatically recompiles if necessary.
ZMessage	Signals a specified condition.
ZPrint	Displays lines of source code.
ZSHow	Displays information about the M environment.
ZSTep	Incrementally executes a routine to the beginning of the next line of the specified type.
ZSYstem	Invokes the shell, creating a forked process.
ZWRite	Displays all or some local or global variables.
$ZCSTATUS	Contains the value of the status code for the last compile performed by a ZCOMPILE, ZLINK or auto-ZLINK.
$ZEDit	Contains the status code for the last ZEDit.
$ZLEVel	Contains the current level of DO/EXECUTE nesting.

GT.M Debugging Tools	
EXTENSION	**EXPLANATION**
$ZMessage()	Returns the text associated with an error condition code.
$ZPOSition	Contains a string indicating the current execution location.
$ZPROmpt	Controls the symbol displayed as the direct mode prompt.
$ZROutines	Contains a string specifying a directory list containing the object, and optionally the source, files.
$ZSOurce	Contains name of the M source program most recently ZLINKed or ZEDITed; default name for next ZEDIT or ZLINK.
$ZStatus	Contains error condition code and location of the last exception condition occurring during routine execution.
$ZSTep	Controls the default ZSTep action.
$ZSYstem	Contains the status code of the last ZSYSTEM.

Chapter 5. General Language Features of M

Revision History		
Revision V7.1-004	27 June 2024	• In "TP Performance" (page 104), add reference to the preceding section and more information on type
Revision V7.1-002	19 September 2023	• In "Key Considerations - Writing TP Code" (page 100), Enhance guide lines • In "TP Definitions" (page 99), add description of combining TRESTART and TROLLBACK • In "TP Performance" (page 104), wording adjustments
Revision V7.1-001	26 June 2023	• In "Summary of GT.M Debugging Tools" (page 66), add ZLINK and auto-ZLINK to the $ZCSTATUS table entry
Revision V7.0-005	02 December 2022	• In "Interrupt Handling" (page 88), remove the extraneous "the".
Revision V7.0-003	24 June 2022	• In "Local Variables" (page 71), changed formalist to formallist
Revision V7.0-002	23 March 2022	• In "Interrupt Handling" (page 88), Revise CTRAP description
Revision V7.0-001	24 November 2021	• In "Pattern Code Indirection" (page 92), add missing quotes in the example. • In "Interrupt Handling" (page 88), add text about the interrupt handling for HANG and commands with timeouts
Revision V6.3-007	04 February 2019	• In "Timeouts" (page 87), correct the max timeout value.
Revision V6.3-006	26 October 2018	• In "String Literals" (page 78), UTF-8 mode tweaks.
Revision V6.3-005	29 June 2018	• In "Key Considerations - Writing TP Code" (page 100), change title to "Key Considerations - Writing TP Code", add introduction material, and enhance the TPNOTACID discussion.
Revision V6.3-003	12 December 2017	• In "Entry References" (page 91), remove a redundant note and specify that GT.M accepts an offset without a label for an entryref argument for DO, GOTO, and ZGOTO. • In "Pattern Match Operator" (page 85), add sentence about XECUTE deferral

Revision V6.3-002	22 August 2017	• In "TP Performance" (page 104), add a note about the VIEW [NO]LOGN[ONTP] and VIEW [NO]LOGT[PRESTART] facilities. • In "Timeouts" (page 87), describe precision and cap.
Revision V6.3-001	20 March 2017	• In "Arithmetic Operators" (page 80), added a statement about divide by literal zero being a run-time only error. • In "Atomic Indirection" (page 92), fixed typo by changing double to single underbar for concatenation • In "Name Indirection" (page 93), Fixed a typo in the Name Indirection sample • In "Postconditionals" (page 87), added explanation of behavior of literal FALSE. • In "Key Considerations - Writing TP Code" (page 100), corrected typo: whose affects -> whose effects • In "TP Example" (page 106), added a link to the description of the gtm_tpnotacidtime environment variable in the Administration and Operations Guide. • In "TP Performance" (page 104), Removed VMS-only text. Added description of TPRESTART message fields.
Revision V6.2-001	27 February 2015	In "Labels" (page 91), improved the description of "local" labels.
Revision V6.0-001	21 March 2013	• In "Numeric Relational Operators" (page 83), added <= and >= as new operators. • In "Indirection Concerns" (page 93), added a note on the handling of run-time errors.

This chapter describes general features of the M language, as well as general information about the operation of GT.M. Commands, Functions, and Intrinsic Special Variables are each described in separate chapters. This chapter contains information about exceptions, as well as information about general M features.

MUMPS is a general purpose language with an embedded database system. This section describes the features of the language that are not covered as Commands, Functions, or Intrinsic Special Variables chapters.

Data Types

M operates with a single basic data type, string. However, M evaluates data using methods that vary according to context.

Numeric Expressions

When M syntax specifies a numexpr, M evaluates the data as a sequence of ASCII characters that specify a number. M stops the evaluation and provides the result generated from successfully evaluated characters when it encounters any character that is not the following:

- A digit 0-9

- A plus sign (+) or minus sign (-) and also the first character in the string

- The first decimal point (.) in the string

Numeric Accuracy

GT.M provides 18 digits of accuracy, independent of the decimal point (.) placement, and a numeric range from $10^{**}(-43)$ to $(10^{**}47)$. Numbers with three digits or fewer to the right of the decimal point are precise.

Integer Expressions

When M syntax specifies an intexpr, M evaluates the data as it would a numexpr except that it stops the evaluation at any decimal point including the first.

Truth-valued Expressions

When M syntax specifies a tvexpr, M evaluates the data as a numeric. However, it stops the evaluation and returns a true value (1) as soon as it encounters a non-zero digit, otherwise it returns a false value (0). In other words, M treats expressions that have a non-zero numeric value as true, and expressions that have a zero numeric value as false. The sign and/or decimal have no affect on the evaluation of a truth-valued expression.

M Names

M uses names for variables, LOCK command arguments, labels on lines, and routine names. M names are alphanumeric and must start with an alphabetic character or a percent sign (%).

The percent sign can only appear as the first character in a name. By convention, names starting with percent signs are generally application-independent or distinguished in some similar way.

M does not reserve any names. That is, M always distinguishes keywords by context. Therefore, M permits a variable or a label called SET even though the language has a command called SET.

M names are case sensitive. That is, M treats ABC, Abc, ABc, AbC ABC, and abc as six different names.

M does not restrict the length of names in the main body of the standard. However, the portability section of the standard recommends limiting names to a maximum of eight (8) characters. GT.M's limit of 31 characters applies to:

- Local variables names

- Global variables names

- Routine names

- Source and object file names (not including the extension)

- Label names

- Local lock resource names

- Global lock resource names

A trigger name is up to 28 characters and a replication instance name is up to 15 characters.

Variables

M does not require predefinition of variable type or size. M variables are either local or global. Any variable may be unsubscripted or subscripted.

Arrays and Subscripts

In M, subscripted variables identify elements in sparse arrays. Sparse arrays comprise existing subscripts and data nodes -; no space is reserved for potential data nodes. These arrays generally serve logical, rather than mathematical, purposes.

M array subscripts are expressions, and are not restricted to numeric values.

The format for an M global or local variable is:

```
[^]name[(expr1[,...])]
```

- The optional leading caret symbol (^) designates a global variable.

- The name specifies a particular array.

- The optional expressions specify the subscripts and must be enclosed in parentheses and separated by commas (,).

The body of the M standard places no restrictions on variable names. However, the portability section of the standard does suggest limits on the length of an individual subscript expression, and on the total length of a variable name. The measurement for the length of names includes the length of the global variable name itself, the sum of the lengths of all the evaluated subscripts, and an allowance for an overhead of two (2) times the number of subscripts. The total must not exceed 237. For globals, GT.M permits this total to be modified with GDE up to 255. For locals, GT.M limits the length of individual subscripts to the maximum string length of 32,767. GT.M restricts the number of subscripts for local or global variables to 31.

M Collation Sequences

M collates all canonic numeric subscripts ahead of all string subscripts, including strings such as those with leading zeros that represent non-canonic numbers. Numeric subscripts collate from negative to positive in value order. String subscripts collate in ASCII sequence. In addition, GT.M allows the empty string subscript in most contexts, (the null, or empty, string collates ahead of all canonic numeric subscripts).

GT.M allows definition of alternative collation sequences. For complete information on enabling this functionality, See Chapter 12: "Internationalization" (page 543).

Local Variables

A local variable in M refers to a variable used solely within the scope of a single process. Local variable names have no leading delimiter.

M makes a local variable available and subject to modification by all routines executed within a process from the time that variable is first SET until it is KILLed, or until the process stops executing M. However, M "protects" a local variable after that variable appears as an argument to a NEW command, or after it appears as an element in a formallist used in parameter passing. When M protects a local variable, it saves a copy of the variable's value and makes that variable undefined. M restores the variable to its saved value during execution of the QUIT that terminates the process stack level associated with the "protecting" NEW or formallist. For more information on NEW and QUIT, see Chapter 6: "Commands" (page 108).

M restricts the following uses of variables to local variables:

• FOR command control variables.

• Elements within the parentheses of an "exclusive" KILL.

• TSTART [with local variables list].

• A KILL with no arguments removes all current local variables.

• NEW command arguments.

• Actualnames used by pass-by-reference parameter passing.

Global Variables and Resource Name Environments

M recognizes an optional environment specification in global names or in the LOCK resource names (nrefs), which have analogous syntax. Global variable names have a leading caret symbol (^) as a delimiter.

M makes a global variable available, and subject to modification by all routines executed within all processes in an environment, from the time that variable is first SET until it is KILLed.

Naked References

M accepts an abbreviation of the global name under some circumstances. When the leading caret symbol (^) immediately precedes the left parenthesis delimiting subscripts, the global variable reference is called a naked reference. M evaluates a naked reference by prefixing the last used global variable name, except for its last subscript, to the list of subscripts specified by the naked reference. The prefixed portion is known as the naked indicator. An attempt to use a naked reference when the prior global reference does not exist, or did not contain a subscript, generates an error.

Because M has only one process-wide naked indicator which it maintains as a side affect of every evaluation of a global variable, using the naked reference requires an understanding of M execution sequence. M execution generally proceeds from left to right within a line, subject to commands that change the flow of control. However, M evaluates the portion of a SET command argument to the right side of the equal sign before the left side. Also, M does not evaluate any further $SELECT() arguments within the function after it encounters a true selection argument.

In general, using naked references only in very limited circumstances prevents problems associated with the naked indicator.

Global Variable Name Environments

M recognizes an optional environment specification in global names. The environment specification designates one of some set of alternative database files.

The syntax for global variable names that include an environment specification is:

```
^|expr|name[(subscript[,...])]
```

In GT.M, the expression identifies the Global Directory for mapping the global variable.

Environment specifications permit easy access to global variables in alternative databases, including other "copies" of active variables in the current database. Environment specifications are sometimes referred to as extended global syntax or extended value syntax.

GT.M also allows:

```
^|expr1,expr2|name[(subscript[,...])]
```

Where the first expression identifies the Global Directory and the second expression is accepted but ignored by GT.M.

To improve compatibility with some other M implementations, GT.M also accepts another non-standard syntax. In this syntax, the leading and trailing up-bar (|) are respectively replaced by a left square-bracket ([) and a right square-bracket (]). This syntax also requires expratoms, rather than expressions. For additional information on expratoms, see "Expressions" (page 80).

The formats for this non-standard syntax are:

```
^[expratom1]name[(subscript...)]
```

or

```
^[expratom1,expratom2]name[(subscript...)]
```

Where expratom1 identifies the Global Directory and expratom2 is a dummy variable. Note that the first set of brackets in each format is part of the syntax. The second set of square brackets is part of the meta-language identifying an optional element.

Example:

```
$ gtmgbldir=Test.GLD
$ export gtmgbldir
$ GTM

GTM>WRITE $ZGBLDIR
TEST.GLD
GTM>WRITE ^A
THIS IS ^A IN DATABASE RED
GTM>WRITE ^|"M1.GLD"|A
THIS IS ^A IN DATABASE WHITE
GTM>WRITE $ZGBLDIR
TEST.GLD
GTM>HALT

$ echo gtmgbldir
TEST.GLD
```

The statement WRITE ^|"M1.GLD"|A writes variable ^A using the Global Directory, M1.GLD, but does not change the current Global Directory.

Example:

```
GTM>WRITE $ZGBLDIR
M1.GLD
GTM>WRITE ^A
THIS IS ^A IN DATABASE WHITE
GTM>WRITE ^|"M1.GLD"|A
THIS IS ^A IN DATABASE WHITE
```

The statement WRITE ^|"M1.GLD"|A is equivalent to WRITE ^A.

Specifying separate Global Directories does not always translate to using separate databases.

Example:

```
GTM>WRITE ^|"M1.GLD"|A,!,^|"M2.GLD"|A,!,^|"M3.GLD"
|A,!
THIS IS ^A IN DATABASE WHITE
THIS IS ^A IN DATABASE BLUE
THIS IS ^A IN DATABASE WHITE
```

In this example, the WRITE does not display ^A from three GT.M database files. Mapping specified by the Global Directory Editor (GDE) determines the database file to which a Global Directory points.

This result could have occurred under the following mapping:

```
^|"M1.GLD"|A --> REGIONA --> SEGMENTA --> FILE1.DAT
^|"M2.GLD"|A --> REGIONA --> SEGMENT1 --> FILE2.DAT
^|"M3.GLD"|A --> REGION3 --> SEGMENT3 --> FILE1.DAT
```

For more information on Global Directories, refer to the "Global Directory Editor" chapter of the **GT.M Administration and Operations Guide**.

Optional GT.M Environment Translation Facility

For users who wish to dynamically (at run-time) determine a global directory from non-global directory information (typically UCI and VOL) in the environment specification, GT.M provides an interface to add an appropriate translation.

Using this facility impacts the performance of every global access that uses environment specification. Make sure you use it only when static determination of the global directory is not feasible. When used, make every effort to keep the translation routines very efficient.

The use of this facility is enabled by the definition of the environment variable gtm_env_translate, which contains the path of a shared library with the following entry point:

gtm_env_xlate

If the shared object is not accessible or the entry point is not accessible, GT.M reports an error.

The gtm_env_xlate() routine has the following C prototype:

```
int gtm_env_xlate(gtm_string_t *in1, gtm_st
   ring_t *in2, gtm_string *in3, gtm_string_t *out)
```

where gtm_string_t is a structure defined in gtmxc_types.h as follows:

```
typedef struct
{
  int length;
  char *address;
}gtm_string_t;
```

The purpose of the function is to use its three input arguments to derive and return an output argument that can be used as an environment specification by GT.M. Note that the input values passed (in1, in2 and in3) are the result of M evaluation and must

not be modified. The first two arguments are the expressions passed within the up-bars "||" or the square-brackets "[ ]", and the third argument is the current working directory as described by $ZDIRECTORY.

A return value other than zero (0) indicates an error in translation, and is reported by a GT.M error

If the length of the output argument is non-zero, GT.M appends a secondary TEXT message, containing the text found at the address of the output structure.

GT.M does not do any memory management related to the output argument - space for the output should be allocated by the external routine. The routine must place the returned environment specification at the address it has allocated and adjust the length accordingly. On a successful return, the return value should be zero. If the translation routine must communicate an error to GT.M, it must return a non-zero value, and if it is to communicate additional error information, place the error text at the address where the environment would normally go and adjust the length to match the length of the error text.

Length of the return value may range from 0-32767, otherwise GT.M reports an error.

A zero-length (empty) string specifies the current value of $ZGBLDIR. Non-zero lengths must represent the actual length of the file specification pointed to by address, excluding any <NUL> terminator. If the address field of the output argument is NULL, GT.M issues an error.

The file specification may be absolute or relative and may contain an environment variable. If the file specified is not accessible, or is not a valid global directory, GT.M reports errors in the same way it does for any invalid global directory.

It is possible to write this routine in M (as a call-in), however, global variables in such a routine would change the naked indicator, which environment references normally do not. Depending on the conventions of the application, there might be difficult name-space management issues such as protecting the local variables used by the M routine.

While it is possible for this routine to take any form that the application designer finds appropriate within the given interface definition, the following paragraphs make some recommendations based on the expectation that a routine invoked for any more than a handful of global references should be efficient.

It is expected that the routine loads one or more tables, either at compilation or the first time it is invoked. The logic of the routine performs a look up on the entry in the set of tables. The lookup might be based on the length of the strings and some unique set of characters in the names, or a hash, with collision provisions as appropriate.

The routine may have to deal with a case where one or both of the inputs have zero length. A subset of these cases may have the first string holding a comma limited string that needs to be re-interpreted as being equivalent to two input strings (note that the input strings must never be modified). The routine may also have to handle cases where a value (most likely the first) is accidentally or intentionally, already a global directory specification.

Example:

```
$ cat gtm_env_translate.c
#include <stdio.h>
#include <string.h>
#include "gtmxc_types.h"
static int init = 0;
typedef struct
{
  gtm_string_t field1, field2, ret;
} line_entry ;
static line_entry table[5], *line, linetmp;
/* Since these errors may occur before setup is complete, they are statics */
static char *errorstring1 ="Error in function initialization, environment variable GTM_CALLIN_START not defined.
Environment translation failed.";
```

```
static char *errorstring2 ="Error in function initialization, function pointers could not be determined.
 Envrironment
▶ translation failed.";
#define ENV_VAR"GTM_CALLIN_START"
typedef int(*int_fptr)();
int_fptr GTM_MALLOC;
int init_functable(gtm_string_t *ptr)
{
/* This function demonstrates the initialization of other function pointers as well (if the user-code needs them
 for
▶ any reason, they should be defined as globals) */
char *pcAddress;
long lAddress;
void **functable;
void (*setup_timer) ();
void (*cancel_timer) ();
pcAddress = getenv(ENV_VAR);
if (pcAddress == NULL)
{
ptr->length = strlen(errorstring1);
ptr->address = errorstring1;
return 1;
}
lAddress = -1;
lAddress = atol(pcAddress);
if (lAddress == -1)
{
ptr->length = strlen(errorstring2);
ptr->address = errorstring2;
return 1;
}
functable = (void *)lAddress;
setup_timer = (void(*)()) functable[2];
cancel_timer = (void(*)()) functable[3];
GTM_MALLOC = (int_fptr) functable[4];
return 0;
}
void copy_string(char **loc1, char *loc2, int length)
{
char *ptr;
ptr = (char *) gtm_malloc(length);
strncpy( ptr, loc2, length);
*loc1 = ptr;
}
int init_table(gtm_string_t *ptr)
{
int i = 0;
char buf[100];
char *buf1, *buf2;
FILE *tablefile;
char *space = " ";
char *errorstr1 = "Error opening table file table.dat";
char *errorstr2 = "UNDETERMINED ERROR FROM GTM_ENV_XLATE";
if ((tablefile = fopen("table.dat","r")) == (FILE *)NULL)
{
```

```
ptr->length = strlen(errorstr1);
copy_string(&(ptr->address), errorstr1, strlen(errorstr1));
return 1;
}
while (fgets(buf, (int)sizeof(buf), tablefile) != (char *)NULL)
{
line= &table[i++];
buf1 = buf;
buf2 =strstr(buf1, space);
line->field1.length = buf2 - buf1;
copy_string( &(line->field1.address), buf1, line->field1.length);
buf1 = buf2+1;
buf2 = strstr(buf1, space);
line->field2.length = buf2-buf1;
copy_string( &(line->field2.address), buf1, line->field2.length);
buf1 = buf2+1;
line->ret.length = strlen(buf1) - 1;
copy_string( &(line->ret.address), buf1, line->ret.length);
}
fclose(tablefile);
/* In this example, the last entry in the table is the error string */
line = &table[4];
copy_string( &(line->ret.address), errorstr2, strlen(errorstr2));
line->ret.length = strlen(errorstr2);
return 0;
}
int cmp_string(gtm_string_t str1, gtm_string_t str2)
{
if (str1.length == str2.length)
return strncmp(str1.address, str2.address, (int) str1.length);
else
return str1.length - str2.length;
}
int cmp_line(line_entry *line1, line_entry *line2)
{
return (((cmp_string(line1->field1, line2->field1))||(cmp_string(line1->field2, line2->field2))));
}
int look_up_table(line_entry *aline, gtm_string_t *ret_ptr)
{
int i;
int ret_v;
for(i=0;i<4;i++)
{
line = &table[i];
ret_v = cmp_line( aline, line);
if (!ret_v)
{
ret_ptr->length = line->ret.length;
ret_ptr->address = line->ret.address;
return 0;
}
}
/*ERROR OUT*/
line = &table[4];
ret_ptr->length= line->ret.length;
```

```
ret_ptr->address = line->ret.address;
return 1;
}
int gtm_env_xlate(gtm_string_t *ptr1, gtm_string_t *ptr2, gtm_string_t *ptr_zdir, gtm_string_t *ret_ptr)
{
int return_val, return_val_init;
if (!init)
{
return_val_init = init_functable(ret_ptr);
if (return_val_init) return return_val_init;
return_val_init = init_table(ret_ptr);
if (return_val_init) return return_val_init;
init = 1;
}
linetmp.field1.length= ptr1->length;
linetmp.field1.address= ptr1->address;
linetmp.field2.length= ptr2->length;
linetmp.field2.address= ptr2->address;
return_val = look_up_table(&linetmp, ret_ptr);
return return_val;
}
> cat table.dat
day1 week1 mumps
day2 week1 a
day3 week2 b
day4 week2 c.gld
```

▼

This example demonstrates the mechanism. A table is set up the first time for proper memory management, and for each reference, a table lookup is performed. Note that for the purpose of simplicity, no error checking is done, so table.dat is assumed to be in the correct format, and have exactly four entries. This routine should be built as a sharedlibrary, see Chapter 11: "Integrating External Routines" (page 521) for information on building as a shared library.The function init_functable is necessary to set up the GT.M memory management functions.

Literals

M has both string and numeric literals.

String Literals

A string literal (strlit) is enclosed in quotation marks (" ") and can contain a sequence of ASCII and Unicode® UTF-8 characters. While the standard indicates the characters must be graphic, GT.M accepts non-graphic characters and, at compile-time, gives a warning. Using $CHAR() and concatenate to represent non-graphic characters in strings not only avoids the warning but is less error prone and makes for easier understanding. M attempts to use character text that appears outside of quotation mark delimiters according to context, which generally means as a local variable name.

To include a quotation mark (") within a strlit, use a set of two quotation marks ("" "").

Example:

```
GTM>write """"
"
```

```
GTM>
```

The WRITE displays a single quotation mark because the first quotation mark delimits the beginning of the string literal, the next two quotation marks denote a single quote within the string, and the last quotation mark delimits the end of the string literal.

Use the $[Z]CHAR() function and the concatenation operator to include control characters within a string.

Example:

```
GTM>WRITE "A"_$CHAR(9)_"B"
A B
GTM>
```

The WRITE displays an "A," followed by a tab (<HT>), followed by a "B" using $CHAR(), to introduce the non-graphic character.

Numeric Literals

In M, numeric literals (numlit) are entered without surrounding delimiters.

Example:

```
GTM>WRITE 1
1
GTM> WRITE 1.1
1.1
```

These display numeric literals that are integer and decimal.

M also accepts numeric literals in the form of a mantissa and an exponent, separated by a delimiter of "E" in uppercase. The mantissa may be an integer or a decimal fraction. The integer exponent may have an optional leading minus sign (-).

Example:

```
GTM>WRITE 8E6
8000000
GTM> WRITE 8E-6
.000008
GTM>
```

Caution

The exponential numeric form may lead to ambiguities in the meaning of subscripts. Because numeric subscripts collate ahead of string subscripts, the string subscript "01E5" is not the same as the numeric subscript 01E5.

GT.M handles numeric strings which are not canonical within the implementation as strings unless the application specifically requests they be treated as numbers. Any use in a context defined as numeric elicits numeric treatment; this includes operands of numeric operators, numeric literals, and some intrinsic function arguments. When the code creates a large number out of range , GT.M gives a NUMOFLOW error. When the code creates a small fractional number out of range GT.M treats it as zero (0). The GT.M number range is (to the limit of accuracy) 1E-43 to 1E47. When the application creates an in-range number that exceeds the GT.M numeric accuracy of 18 significant digits, GT.M silently retains the most significant digits. With standard

collation, GT.M collates canonic numeric strings used as subscripts numerically, while it collates non-canonic numbers as strings.

Expressions

The following items are legal M expression atoms (expratom). An expression atom is a component of an M expression.

- Local variables

- Global variables

- Intrinsic special variables

- Intrinsic functions

- Extrinsic functions

- Extrinsic special variables

- Numeric literals

- String literals

- An expression enclosed in parentheses

- Any of the above preceded by a unary operator

In addition, any of these items may be combined with a binary operator and another expression atom.

Operators

M has both unary and binary operators.

Precedence

All unary operations have right to left precedence.

All M binary operations have strict left to right precedence. This includes all arithmetic, string, and logical operations. Hierarchies of operations require explicit establishment of precedence using parentheses (). Although this rule is counterintuitive, it is easy to remember and has no exceptions.

Arithmetic Operators

All arithmetic operators force M to evaluate the expressions to which they apply as numeric. The arithmetic operators are:

+ as a unary operator simply forces M to evaluate the expression following as numeric; as a binary operator it causes M to perform addition.

- as a unary operator causes M to negate the expression following; as a binary operator it causes M to perform subtraction.

* binary operator for multiplication.

** binary operator for exponentiation.

/ binary operator for fractional division.

\ binary operator for integer division.

binary operator for modulo, that is, causes M to produce the remainder from integer division of the first argument by the second.

Because of the practice of using it to intentionally induce an error, GT.M does not produce a DIVZERO error at compile time, only at run time, for divide or integer divide by a literal expression that evaluates to zero (0).

Remember that precedence is left to right for all arithmetic operators.

Example:

```
GTM>WRITE 1+1
2
GTM>WRITE 2-1
1
GTM>WRITE 2*2
4
GTM>WRITE 3**2
9
GTM>WRITE 4/2
2
GTM>WRITE 7
2
GTM>WRITE 7#3
1
GTM>
```

This simple example demonstrates how each arithmetic binary operation uses numeric literals.

Example:

```
GTM>WRITE +"12ABC"
12
GTM>WRITE --"-3-4"
-3
GTM>
```

The first WRITE shows the unary plus sign (+) operation forcing the numeric evaluation of a string literal. The second WRITE demonstrates the unary minus sign (-). Note the second minus sign within the string literal does not cause subtraction, but rather, terminates the numeric evaluation with the result of negative three (-3). Each of the leading minus signs causes one negation and therefore, the result is negative three (-3).

Logical Operators

M logical operators always produce a result that is TRUE (1) or FALSE (0). All logical operators force M to evaluate the expressions to which they apply as truth-valued. The logical operators are:

' unary NOT operator negates current truth-value; M accepts placement of the NOT operator next to a relational operator, for example, A'=B as meaning '(A=B).

&binary AND operator produces a true result only if both of the expressions are true.

! binary OR operator produces a true result if either of the expressions is true.

Remember that precedence is always left to right, and that logical operators have the same precedence as all other operators.

Example:

```
GTM>WRITE '0
1
GTM>WRITE '1
0
GTM>WRITE '5689
0
GTM>WRITE '-1
0
GTM>WRITE '"ABC"
1
GTM>
```

The above example demonstrates the unary NOT operation. Note that any non-zero numeric value is true and has a false negation.

Example:

```
GTM>WRITE 0&0
0
GTM>WRITE 1&0
0
GTM>WRITE 0&1
0
GTM>WRITE 1&1
1
GTM>WRITE 2&1
1
GTM>WRITE 0!0
0
GTM>WRITE 1!0
1
GTM>WRITE 0!1
1
GTM>WRITE 1!1
1
GTM>WRITE 2!1
1
GTM>
```

The above example demonstrates all cases covered by the binary logical operators.

String Operators

All string operators force M to evaluate the expressions to which they apply as strings. The string operator is:

_binary operator causes M to concatenate the second expression with the first expresion

Example:

```
GTM>WRITE "B"_"A"
BA
GTM>WRITE "A"_1
A1
GTM>
```

The above example demonstrates M concatenation.

Numeric Relational Operators

M relational operators always generate a result of TRUE (1) or FALSE (0). All numeric relational operators force M to evaluate the expressions to which they apply as numeric. The numeric relational operators are:

>binary arithmetic greater than

<binary arithmetic less than

The equal sign (=) does not force numeric evaluation, and should be viewed as a string operator. However, the equal sign between two numeric values tests for numeric equality.

Other numeric relations are formed using the logical NOT operator apostrophe (') as follows:

'> not greater than, that is, less than or equal to

'< not less than, that is, greater than or equal to

>= greater than or equal to, that is, not less than

<= less than or equal to, that is, not greater than

'= not equal, numeric or string operation

Example:

```
GTM>WRITE 1>2
0
GTM>WRITE 1<2
1
GTM>
```

The above example demonstrates the basic arithmetic relational operations.

Example:

```
GTM>WRITE 1'<2
0
GTM>WRITE 2'<1
1
GTM>
```

The above example demonstrates combinations of arithmetic, relational operators with the logical NOT operator.

String Relational Operators

M relational operators always generate a result of TRUE (1) or FALSE (0). All string relational operators force M to evaluate the expressions to which they apply as strings. The string relational operators are:

= binary operator causes M to produce a TRUE if the expressions are equal.

[binary operator causes M to produce a TRUE if the first expression contains the ordered sequence of characters in the second expression.

] binary operator causes M to produce a TRUE if the first expression lexically follows the second expression in the character encoding sequence, which by default is ASCII.

]] binary operator causes M to produce a TRUE if the first expression lexically sorts after the second expression in the subscript collation sequence.

Note that all non-empty strings lexically follow the empty string, and every string contains the empty string.

Other string relations are formed using the logical NOT operator apostrophe (') as follows:

'[does not contain.

'] does not follow, that is, lexically less than or equal to.

']] does not sort after, that is, lexically less than or equal to in the subscript collation sequence.

'= not equal, numeric or string operation.

Example:

```
GTM>WRITE "A"="B"
0
GTM>WRITE "C"="C"
1
GTM>WRITE "A"["B"
0
GTM>WRITE "ABC"["C"
1
GTM>WRITE "A"]"B"
0
GTM>WRITE "B"]"A"
1
GTM>WRITE "A"]]"B"
0
GTM>WRITE "B"]]"A"
1
```

These examples demonstrate the string relational operators using string literals.

Example:

```
GTM>WRITE 2]10
1
GTM>WRITE 2]]10
0
```

```
GTM>WRITE 0]"$"
1
GTM>WRITE 0]]"$"
0
```

These examples illustrate that when using the primary ASCII character set, the main difference in the "follows" (]) operator and the "sorts-after" (]]) operator is the way they treat numbers.

Example:

```
GTM>WRITE 1=1
1
GTM>WRITE 1=2
0
GTM>WRITE 1="1"
1
GTM>WRITE 1=01
1
GTM>WRITE 1="01"
0
GTM>WRITE 1=+"01"
1
GTM>
```

These examples illustrate the dual nature of the equal sign operator. If both expressions are string or numeric, the results are straight forward. However, when the expressions are mixed, the native string data type prevails.

Example:

```
GTM>WRITE "a"'="A"
1
GTM>WRITE "FRED"'["RED"
0
GTM>WRITE "ABC"']""
0
```

These examples demonstrate combinations of the string relational operators with the NOT operator.

Pattern Match Operator

The pattern match operator (?) causes M to return a TRUE if the expression ahead of the operator matches the characteristics described by the pattern following the operator. The pattern is not an expression.

Patterns are made up of two elements:

1. A repetition count

2. A pattern code, a string literal or an alternation list

The element following the pattern match operator may consist of an indirection operator, followed by an element that evaluates to a legitimate pattern.

The repetition count consists of either a single integer literal or a period (.) delimiter with optional leading and trailing integer literals. A single integer literal specifies an exact repetition count. The period syntax specifies a range of repetitions where the

leading number is a minimum and the trailing number is a maximum. When the repetition count is missing the leading number, M assumes there is no minimum, (i.e., a minimum of zero). When the repetition count is missing the trailing number, M does not place a maximum on the number of repetitions.

The pattern codes are:

A alphabetic characters upper or lower case

C control characters ASCII 0-31 and 127

E any character; used to pass all characters in portions of the string where the pattern is not restricted

L lower-case alphabetic characters, ASCII 97-122

N digits 0-9, ASCII 48-57

P punctuation, ASCII 32-47, 58-64, 91-96, 123-126

U upper-case alphabetic characters, ASCII 65-90

Pattern codes may be upper or lower case and may be replaced with a string literal. GT.M allows the M pattern match definition of patcodes A, C, N, U, L, and P to be extended or changed, (A can only be modified implicitly by modifying L or U) and new patcodes added. For detailed information on enabling this functionality, see Chapter 12: "Internationalization" (page 543).

Note

The GT.M compiler accepts pattern codes other than those explicitly defined above. If, at run-time, the pattern codes come into use and no pattern definitions are available, GT.M issues a run-time error (PATNOTFOUND). GT.M does not currently implement a mechanism for Y and Z patterns and continues to treat those as compile-time syntax errors. GT.M defers literal optimizations involving patterns within an XECUTE as well as evaluations that encounter issues with the pattern table.

Example:

```
GTM>WRITE "ABC"?3U
1
GTM>WRITE "123-45-6789"?3N1"-"2N1"-"4N
1
```

The first WRITE has a simple one-element pattern while the second has multiple elements including both codes and string literals. All the repetition counts are fixed.

Example:

```
I x?.E1C.E W !,"Must not contain a control character" Q
```

This example uses a pattern match to test for control characters.

Example:

```
I acn?1U.20A1","1U.10A D
.S acn=$G((^ACX($P(acn,","),$P(acn,",",2)))
```

This example uses a pattern match with implicit minimums to determine that an "account number" is actually a name, and to trigger a look-up of the corresponding account number in the ^ACX cross index.

The pattern match operator accepts the alteration syntax. Alteration consists of a repeat count followed by a comma-delimited list of patatoms enclosed in parentheses "()". The semantic is that the pattern matches if any of the listed patterns matches the operand string. For example, ?1(2N1"-"7N,3N1"-"2N1"-"4N).1U might be a way to match either a social security number or a taxpayer ID. Since alteration is defined as one of the ways of constructing a patatom, alteration can nest (be used recursively).

> **Note**
>
> Complex pattern matches may not be efficient to evaluate, so every effort should be made to simplify any commonly used pattern and to determine if more efficient alternative logic would be more appropriate.

Commands

M commands may be abbreviated to a defined prefix. Most commands have arguments. However, some commands have either optional arguments or no arguments. When a command has no argument and is followed by more commands on the same line, at least two spaces (<SP>) must follow the command without arguments. Commands that accept arguments generally accept multiple arguments on the same command. M treats multiple arguments the same as multiple occurrences of the same command, each with its own argument.

Postconditionals

M provides postconditionals as a tool for placing a condition on the execution of a single command and, in some cases, a single command argument. A postconditional consists of a colon (:) delimiter followed by a truth-valued expression. When the expression evaluates to true, M executes the command occurrence. When the expression evaluates to false, M does not execute the command occurrence.

Command Postconditionals

Command postconditionals appear immediately following a command and apply to all arguments for the command when it has multiple arguments. All commands except commands that themselves have a conditional aspect accept a command postconditional. Among the M standard commands, ELSE, FOR, and IF do not accept command postconditionals. All the GT.M command extensions accept command postconditionals. When a postconditional evaluates to a literal FALSE (0), GT.M discards the command and its arguments at compile time, which means it does not perform any validity checking on the arguments.

Argument Postconditionals

Commands that affect the flow of control may accept postconditionals on individual command arguments. Because multiple arguments act as multiple commands, this is a straight-forward application of the same principal as command postconditional. The only M standard commands that accept argument postconditionals are DO, GOTO, and XECUTE. The GT.M command extensions that accept argument postconditionals are BREAK, ZGOTO, and ZSYSTEM.

Timeouts

M provides timeouts as a tool to retain program control over commands of indefinite duration. A timeout consists of a colon (:) delimiter on an argument, followed by a numeric expression specifying the number of seconds to millisecond (three decimal

place) precision for M to attempt to execute the command. When the timeout is zero (0), M makes a single attempt to complete the command.

GT.M caps the maximum timeout to 2,147,483.647 seconds (about 24.8 days), and converts values greater than the maximum timeout to that cap. When a command has a timeout, M maintains the $TEST intrinsic special variable as the command completes. If the command completes successfully, M sets $TEST to TRUE (1). If the command times out before successful completion, M sets $TEST to FALSE (0). When a command argument does not specify a timeout, M does not maintain $TEST.

The following commands accept timeouts:

- LOCK

- JOB

- OPEN

- READ

- ZALLOCATE

When a READ times out, M returns any characters that have arrived between the start of the command and the timeout. M does not produce any partial results for any of the other timed commands.

Interrupt Handling

GT.M process execution is interruptible with the following events:

- Typing <CTRL-C> or getting SIGINT if CTRAP=$CHAR(3) for the terminal device or on a $PRINCIPAL terminal if its mode is CENABLE.

- Typing <CTRL-n> if CTRAP=$CHAR(n) on a terminal device performing a READ

- Getting a MUPIP INTRPT (SIGUSR1)

- Exceeding $ZMAXTPTIME in a transaction

- $ZTIMEOUT expires

- A terminal disconnect ("hangup")

- A terminal output error during an asynchronous flush

- GT.CM network error

- A MALLOCLIM error

- +$ZTEXit evaluates to a truth value at the outermost TCOMMIT or TROLLBACK

When GT.M detects any of these events, it transfers control to a vector that depends on the event. For most events, GT.M uses the $ETRAP or $ZTRAP vectors described in more detail in the Error Processing chapter. For INTRPT and $ZTEXit, it XECUTEs the interrupt handler code placed in $ZINTERRUPT. If $ZINTERRUPT is an empty string, the process ignores any MUPIP INTRPT directed at it. The default value of $ZINTERRUPT is "IF $ZJOBEXAM()" which redirects a dump of ZSHOW "*" to a file and reports each such occasion to the operator log. For $ZTIMEOUT, the value may specify a vector that takes precedence

over the current error handling vector. For <CTRL-C> without CENABLE, it transfers control as if there was an error; with CENABLE, GT.M enters Direct Mode to give the programmer control. Without CENABLE or CTRAP, GT.M ignores <CTRL-C> on a $PRINCIPAL terminal. The GT.M terminal handler only recognizes other <CTRL> characters if CTRAP enabled when the OS terminal handling delivers and they appear in the terminal input stream.

GT.M recognizes most of these events when they occur but transfers control to the interrupt vector at the start of each M line, at each iteration of a FOR loop, at certain points during the execution of commands which may take a "long" time. For example, ZWRITE, HANG, LOCK, MERGE, ZSHOW "V", OPEN of terminals, disk files, PIPEs, FIFOs, and SOCKETs (unless zero timeout,) WRITE /WAIT for SOCKETs, and READ for terminals, SOCKETs, FIFOs, PIPEs, and sequential files in FOLLOW mode.

The HANG command and timed restartable I/O commands such as timed READ for terminals, SOCKETs, FIFO, PIPE, and sequential files in FOLLOW mode as well as SOCKET OPEN CONNECT and WRITE /WAIT account for time spent in handling the interrupt. However, the LOCK command pauses the timeout countdown until the interrupt handling is complete.

If +$ZTEXIT evaluates to a truth value at the outermost TCOMMIT or TROLLBACK, GT.M XECUTEs $ZINTERRUPT after completing the commit or rollback. Except for <CTRL-C> GT.M recognizes CTRAP characters when READ. CTRAP characters other than <CTRL-C> tend to be limited by terminal configuration.

M Locks

The LOCK command reserves one or more resource names. Only one process at a time can reserve a resource name. Resource names follow exactly the same formation rules as M variables. They may be unsubscripted or subscripted and may or may not have a leading caret (^) prefix. M code commonly uses LOCKs as flags that control access to global data. Generally, a LOCK specifies the resource with the same name as the global variable that requires protected access. However, this is only a convention. LOCKing does not keep two or more processes from modifying the same global variable. It only keeps another process from LOCKing the same resource name at the same time.

M LOCKs are hierarchical. If one process holds a LOCK on a resource, no other process can LOCK either an ancestor or a descendant resource. For example, a LOCK on ^A(1,2) blocks LOCKs on either ^A(1), or ^A(1,2,3), but not on, for example, ^A(2) or its descendants.

A LOCK argument may contain any subscripted or unsubscripted M variable name including a name without a preceding caret symbol (^). As they have the appearance of local variable names, resource names with no preceding caret symbol (^) are commonly referred to as "local LOCKs" even though these LOCKs interact with other processes. For more information on the interaction between LOCKs and processes, refer to the LKE chapter in the **GT.M Administration and Operations Guide**.

The GT.M run-time system records LOCK information in memory associated with the region holding the global of the same name. However, GT.M does not place LOCKs in the database structures that hold the globals. Instead the GT.M LOCK manager sets up a "LOCK database" associated with each database region. Only the M commands LOCK, ZALLOCATE, and ZDEALLOCATE and the LKE utility access the information in the "LOCK database".

GT.M distributes the LOCK database within space associated with the database files identified by the Global Directory (GD). The Global Directory Editor (GDE) enables you to create and maintain global directories. GT.M associates LOCKs of resource names starting with a caret symbol (^) with the database region used to map variables with the same name. If the global directory maps the name ^A to file A.DAT, GT.M maps all LOCKs on resource name ^A to LOCK space implemented in shared memory control structures associated with A.DAT. GT.M maps LOCKs on names not starting with a caret symbol (^) to the region of the database specified with the GDE command LOCKS -REGION.

By default, GDE creates global directories mapping "local" LOCKs to the region DEFAULT.

^LOCKS automatically intersect for all users of the same data in any database file, because GT.M associates the ^LOCKs with the same region as the global variables with the same name.

"Local" LOCK intersections are dependent on the global directory, because users may access the database through different global directories. The "local" LOCKs of two processes interact with each other only when the same lock resource names map to the same database region.

See Also	• "Lock" (page 132)
	• "ZSHOW Information Codes" (page 193)
	• "ZAllocate" (page 170)
	• "ZDeallocate" (page 176)
	• GDE LOCKs (Administration and Operations Guide)
	• LKE Chapter (Administration and Operations Guide)

Intrinsic Functions

M Intrinsic Functions start with a single dollar sign ($) and have one or more arguments enclosed in parentheses () and separated by commas (,). These functions provide an expression result by performing actions that would be impossible or difficult to perform using M commands. It is now possible to invoke a C function in a package via the external call mechanism. For information on the functions, see Chapter 7: "Functions" (page 212).

Intrinsic Special Variables

M Intrinsic Special Variables start with a single dollar sign ($). GT.M provides such variables for program examination. In some cases, the Intrinsic Special Variables may be SET to modify the corresponding part of the environment. For information, see Chapter 8: "Intrinsic Special Variables" (page 291).

Routines

M routines have a name and consist of lines of code followed by a formfeed. M separates the name of a routine from the body of the routine with an end-of-line which is a line-feed. This form is mostly used for interchange with other M implementations and can be read and written by the %RI and %RO utility routines.

GT.M stores routine sources in UNIX text files.

In M, a routine has no particular impact on variable management and may include code that is invoked at different times and has no logical intersection.

Lines

A line of M code consists of the following elements in the following order:

• An optional label.

• A line-start delimiter. The standard defines the line-start delimiter as a space (<SP>) character. In order to enhance routine readability, GT.M extends M by accepting one or more tab (<HT>) characters as line-start delimiters.

• Zero or more level indicators, which are periods (.). The level indicators show the level of nesting for argumentless DO commands: the more periods, the deeper the nesting. M ignores lines that contain level indicators unless they directly follow an argumentless DO command with a matching level of nesting.

For more information on the DO command, see Chapter 6: "Commands" (page 108).

- Zero or more commands and their arguments. M accepts multiple commands on a line. The argument(s) of one command are separated from the next command by a command-start delimiter, consisting of one or more spaces (<SP>).

- A terminating end-of-line, which is a line feed.

Labels

In addition to labels that follow the rules for M names, M accepts labels consisting only of digits. In a label consisting only of digits, leading zeros are considered significant. For example, labels 1 and 01 are different. Formalists may immediately follow a label. A Formalists consists of one or more names enclosed in parentheses (). Formalists identify local variables that "receive" passed values in M parameter passing. For more information, see "Parameter Passing" (page 93).

In GT.M, a colon (:) delimiter may be appended to the label, which causes the label to be treated as "local." Within the routine in which they appear, they perform exactly as they would without the trailing colon but they are available only during compilation and inaccessible to other routines and to indirection or XECUTE. Because references to local labels preceding their position in a routine produce a LABELUNKNOWN error at run-time, FIS recommends omitting the routinename from labelrefs to a local label. Using local labels reduces object size and linking overhead for both all types for dynamic linking except indirection and XECUTE. Use of local labels may either improve or impair performance; typically any difference is modest. The more likely they are to all be used within the code block at run-time, the more likely an improvement. In other words, conditional code paths which prevent all references to local variables appearing in the block may actually impair performance.

Comments

In addition to commands, a line may also contain a comment that starts with a leading semi-colon (;) delimiter. The scope of a comment is the remainder of the line. In other words, M ignores anything to the right of the comment delimiter. The standard defines the comment delimiter (;) as it would a command, and therefore requires that it always appear after a linestart. GT.M extends the standard to permit comments to start at the first character of a line or in an argument position.

Entry References

M entryrefs provide a generalized target for referring to a line within a routine. An entryref may contain some combination of a label, an offset, and a routine name (in that order). The offset is delimited by a plus sign (+) and the routinename is delimited by a caret symbol(^). When an entryref does not contain a label, M assumes the offset is from the beginning of the routine. When an entryref does not contain an offset, M uses an offset of zero (0). When an entryref does not contain a routine name, M assumes the routine that is currently executing.

M permits every element in an entryref to have the form of an indirection operator, followed by an element that evaluates to a legitimate occurrence of that portion of the entryref.

> **Note**
>
> GT.M accepts an offset without a label (for example +3^RTN) for an entryref argument to DO, GOTO and ZGOTO but prohibits the same during paramter passing with the JOB command.

Offsets provide an extremely useful tool for debugging. However, avoid their use in production code because they generally produce maintenance problems.

Label References

M labelrefs are a subset of entryrefs that exclude offsets and separate indirection. Labelrefs are used with parameter passing.

Indirection

M provides indirection as a means to defer definition of elements of the code until run-time. Indirection names a variable that holds or "points" to the element. The indirection operator is the "at" symbol (@).

Argument Indirection

Most commands accept indirection of their entire argument.

Example:

```
GTM>set x="^INDER"
GTM>do @x
```

This example is equivalent to **do ^INDER**.

Atomic Indirection

Any expratom or any local or global variable name may be replaced by indirection.

Example:

```
GTM>set x="HOOP",b="x"
GTM>set a="HULA "_@b
GTM>write a
HULA HOOP
GTM>
```

This example uses indirection within a concatenation operation.

Entryref Indirection

Any element of an entryref may be replaced by indirection.

Example:

```
GTM>set lab="START",routine="PROG"
GTM>do @lab^@routine
```

This example is equivalent to **do START^PROG**.

Pattern Code Indirection

A pattern code may be replaced by indirection.

Example:

```
GTM>FOR p="1U.20A1"",""1U.20A","5N" IF x?@p QUIT
GTM>ELSE WRITE !,"Incorrect format" QUIT
```

This example uses pattern code indirection to test x for either a name (of the form Last,First) or a number.

Name Indirection

Indirection may replace the prefix of a subscripted global or local variable name. This "name" indirection requires two indirection operators, a leading operator similar to the other forms of indirection, and a trailing operator marking the transition to those subscripts that are not specified by indirection.

Example:

```
GTM>SET from="B",to="^A(15)",x=""
GTM>FOR SET x=$O(@from@(x)) Q:x="" S @to@(x)=@from@(x)
```

This example uses name indirection to copy the level contents of a local array to a part of a global array. The example assumes that all existing first level nodes of variable B have data.

Indirection Concerns

M indirection provides a very powerful tool for allowing program abstraction. However, because indirection is frequently unnecessary and has some disadvantages, use it carefully.

Because routines that use indirection in some ways do not contain adequate information for easy reading, such routines tend to be more difficult to debug and maintain.

To improve run-time performance, GT.M tends to move work from run-time to compile-time. Indirection forces compiler actions to occur at run-time, which minimizes the benefits of compilation.

M allows most forms of indirection to be recursive. However, in real applications, recursive indirection typically makes the code obscure and slow.

There are circumstances where indirection serves a worthwhile purpose. For instance, certain utility functions with a general nature may be clearly abstracted and coded using indirection. Because M has no "CASE" command, DO (or GOTO) with argument indirection provides a clear solution to the problem of providing complex branching.

Some M users prototype with indirection and then replace indirection with generated code that reduces run-time overhead. In any case, always consider whether indirection can be replaced with a clearer or more efficient approach.

Run-time errors from indirection or XECUTEs maintain $STATUS and $ZSTATUS related information and cause normal error handling but do not provide compiler supplied information on the location of any error within the code fragment.

Parameter Passing

Parameter passing provides a way of explicitly controlling some or all of the variable context transferred between M routines.

M uses parameter passing for:

- A DO command with parameters

- Extrinsic functions and special variables

Parameter passing is optional on DO commands.

Parameter passing uses two argument lists: the actuallist that specifies the parameters that M passes to an invoked routine, and the formalist that specifies the local variables to receive or associate with the parameters.

Actuallists

An actuallist specifies the parameters M passes to the invoked routine. The actuallist contains a list of zero or more parameters enclosed in parentheses, immediately following a DO or extrinsic function.

An actuallist:

- Is made up of items separated by commas

- Contains expressions and/or actualnames. Items may be missing, that is, two commas may appear next to each other, with nothing between them.

- Must be used in an invocation of a label with a formallist, except in the case of extrinsic special variables.

- Must not contain undefined variables.

- Must not have more items than a formallist with which it is used.

- May contain the same item in more than one position.

Example:

```
GTM>DO MULT(3,X,.RESULT)
```

This example illustrates a DO with parameters. The actuallist contains:

- 3 - a numeric literal

- X - a local variable

- .RESULT - an actualname

Actualnames

An actualname starts with a leading period (.) delimiter, followed by an unsubscripted local variable name. Actualnames identify variables that are passed by reference, as described in a subsequent section. While expressions in an actualname are evaluated when control is transferred to a formallabel, the variables identified by actualnames are not; therefore, they do not need to be defined at the time control is transferred.

Formallists

A formallist specifies the variables M uses to hold passed values. A formallist contains a list of zero or more parameters enclosed in parentheses, immediately following a label.

A formallist:

- Is made up of items separated by commas.

- Contains unsubscripted local variable names.

- Must be used and only used with a label invoked with an actuallist or an extrinsic.

- May contain undefined variables.

- May have more items than an actuallist with which it is used.

- Must not contain the same item in more than one position.

- Must contain at least as many items as the actuallist with which it is used.

Example:

```
MULT(MP,MC,RES)
SET RES=MP*MC
QUIT RES
```

In this example, illustrating a simple parameterized routine, the formallist contains the following items:

- MP

- MC

- RES

An example in the section describing "Actuallists" shows an invocation that matches this routine.

Formallabel

A label followed by a formallist is called a formallabel.

Parameter Passing Operation

M performs an implicit NEW on the formallist names and replaces the formallist items with the actuallist items.

M provides the actuallist values to the invoked procedure by giving each element in the formallist the value or reference provided by the corresponding element in the actuallist. M associates the first name in the formallist with the first item in the actuallist, the second name in the formallist with the second item in the actuallist and so on. If the actuallist is shorter than the formallist, M ensures that the formallist items with no corresponding value are in effect NEWed. If the formallist item has no corresponding item in the actuallist (indicated by two adjacent commas in the actuallist), that item in the formallist becomes undefined.

If the actuallist item is an expression and the corresponding formallist variable is an array, parameter passing does not affect the subscripted elements of the array. If an actualname corresponds to a formallist variable, M reflects array operations on the formallist variable, by reference, in the variable specified by the actualname.

M treats variables that are not part of the formallist as if parameter passing did not exist (i.e., M makes them available to the invoked routine).

M initiates execution at the first command following the formallabel.

A QUIT command terminates execution of the invoked routine. At the time of the QUIT, M restores the formallist items to the values they had at the invocation of the routine.

> ### Note
>
> In the case where a variable name appears as an actualname in the actuallist, and also as a variable in the formallist, the restored value reflects any change made by reference.

A QUIT from a DO does not take an argument, while a QUIT from an extrinsic must have an argument. This represents one of the two major differences between the DO command with parameters and the extrinsics. M returns the value of the QUIT command argument as the value of the extrinsic function or special variable. The other difference is that M stacks $TEST for extrinsics.

For more information, see "Extrinsic Functions" (page 98) and "Extrinsic Special Variables" (page 99).

Example:

```
SET X=30,Z="Hello"
DO WRTSQR(X)
ZWRITE
QUIT
WRTSQR(Z)
SET Z=Z*Z
WRITE Z,!
QUIT
```

Produces:

```
900
X=30
Z="Hello"
```

Parameter Passing Mechanisms

M passes the actuallist values to the invoked routine using two parameter-passing mechanisms:

- Call-by-Value - where expressions appear

- Call-by-Reference - where actualnames appear

A call-by-value passes a copy of the value of the actuallist expression to the invoked routine by assigning the copy to a formallist variable. If the parameter is a variable, the invoked routine may change that variable. However, because M constructs that variable to hold the copy, it deletes the variable holding the copy when the QUIT restores the prior formallist values. This also means that changes to the variable by the invoked routine do not affect the value of the variable in the invoking routine.

Example:

```
SET X=30
DO SQR(X)
ZWRITE
QUIT
SQR(Z)SET Z=Z*Z
QUIT
```

Produces:

```
X=30
```

A period followed by a name identifies an actualname and causes a call-by-reference.

A call-by-reference passes a pointer to the variable of the invoked routine so operations on the assigned formallist variable also act on the actualname variable. Changes, including KILLs to the formallist variable, immediately have the same affect on the

corresponding actualname variable. This means that M passes changes to formallist variables in the invoked routine back to the invoking routine as changes in actualname variables.

Example:

```
SET X=30
DO SQR(.X)
ZWRITE
QUIT
SQR(Z)SET Z=Z*Z
QUIT
```

Produces:

```
X=900
```

GT.M Parameter Passing Extensions

The standard does not provide for indirection of a labelref because the syntax has an ambiguity.

Example:

```
DO @X(1)
```

This example could be:

- An invocation of the label specified by X with a parameter of 1.

- An invocation of the label specified by X(1) with no parameter list.

GT.M processes the latter interpretation as illustrated in the following example.

Example:

The syntax:

```
SET A(1)="CUBE",X=5
DO @A(1)(.X)
WRITE X,!
QUIT
CUBE(C);cube a variable
SET C=C*C*C
QUIT
```

Produces the result:

```
125
```

GT.M follows analogous syntax for routine indirection:

DO ^ @X(A) invokes the routine specified by X(A).

DO ^ @(X)(A) invokes the routine specified by X and passes the parameter A.

DO ^ @X(A)(A) invokes the routine specified by X(A) and passes the parameter A.

External Calls

GT.M allows references to a GT.M database from programs written in other programming languages that run under UNIX.

In GT.M, calls to C language routines may be made with the following syntax:

```
DO &[packagename.]name[^name][parameter-list]
```

or as an expression element,

```
$&[packagename.]name[^name][parameter-list]
```

Where packagename, like the name elements is a valid M name. Because of the parsing conventions of M, the identifier between the ampersand (&) and the optional parameter-list has precisely constrained punctuation – a later section describes how to transform this into a more richly punctuated name should that be appropriate for the called function. While the intent of the syntax is to permit the name^name to match an M labelref, there is no semantic implication to any use of the caret (^).

> ### Note
>
> For more information on external calls, see Chapter 11: "Integrating External Routines" (page 521).

Extrinsic Functions

An extrinsic function is an M subroutine that another M routine can invoke to return a value.

The format for extrinsic functions is:

```
$$[label][^routinename]([expr|.lname[,...]])
```

- The optional label and optional routinename make up the formallabel that specifies the name of the subroutine performing the extrinsic function. The formallabel must contain at least one of its optional components.

- The optional expressions and actualnames make up the actuallist that specifies the list of actual parameters M passes to the invoked routine.

M stacks $TEST for extrinsic functions. This is one of the two major differences between the DO command with parameters and extrinsics. On return from an extrinsic function, M restores the value of $TEST to what it was before the extrinsic function, regardless of the actions executed by the invoked routine.

M requires a routine that implements an extrinsic function to terminate with an explicit QUIT command which has an argument. M returns the value of the QUIT command argument as the value of the extrinsic function. This is the other major difference between the DO command with parameters and extrinsics. It is now possible to invoke a C function in a package via the external call mechanism.

Example:

```
POWER(V,X,S,T);extrinsic to raise to a power
;ignores fractional powers
SET T=1,S=0
IF X<0 SET X=-X,S=1
FOR X=1:1:X S T=T*V
QUIT $S(S:1/T,1:T)
GTM> WRITE $$^POWER(3,4)
81
GTM>
```

> **Note**
>
> The POWER routine uses a formallist that is longer than the "expected" actuallist to protect local working variables. Such practice may be encouraged or discouraged by your institution's standards.

Extrinsic Special Variables

An extrinsic special variable is a user-written M subroutine that another M routine can invoke to return a value.

The format for extrinsic special variables is:

```
$$[label][^routinename]
```

- The optional label and optional routinename make up the formallabel, which specifies the name of the subroutine performing the extrinsic function. The formallabel must contain at least one of its optional component.

An extrinsic special variable can be thought of as an extrinsic function without input parameters. $$x is equivalent in operation to $$x(). Extrinsic special variables are the only case where invocation of a formallabel does not require an actuallist. M stacks $TEST for extrinsic special variables.

M requires that a routine that implements an extrinsic special variable terminate with an explicit QUIT command which has an argument. M returns the value of the QUIT command argument as the value of the extrinsic special variable.

Example:

```
GTM>ZPRINT ^DAYOWEEK
DAYOWEEK();extrinsic special variable to
;provide the day of the week
QUIT $ZD($H,"DAY")
GTM>WRITE $$DAYOWEEK^DAYOWEEK
MON
```

Transaction Processing

Transaction Processing (TP) provides a way for M programs to organize database updates into logical groups that occur as a single event (i.e., either all the database updates in a transaction occur, or none of them occur). With a properly constructed transaction, no other actor or process behaves as if it observed any intermediate state. Transaction processing has been designed to improve throughput and minimize the possibility and impact of "live lock" conditions.

TP Definitions

In M, a transaction is a sequence of commands that begins with a TSTART command, ends with a TCOMMIT command, and is not within the scope of another transaction. Applications can nest TSTART/TCOMMIT commands to create sub-transactions, but sub-transactions only commit at the outer-most TCOMMIT. $TLEVEL greater than 1 indicates sub-transaction nesting.

A successful transaction ends with a COMMIT that is triggered by the TCOMMIT command at the end of the transaction. A COMMIT causes all the database updates performed within the transaction to become available to other processes.

An unsuccessful transaction ends with a ROLLBACK. ROLLBACK is invoked explicitly by the TROLLBACK command, or implicitly at a process termination that occurs during a transaction in progress. An error within a transaction does not itself cause an implicit ROLLBACK, although $ETRAP error processing may cause an implicit TROLLBACK. A ROLLBACK removes

any database updates performed within the transaction before they are made available to other processes. ROLLBACK also releases all resources LOCKed since the start of the transaction, and makes the naked reference undefined. While it cause a significant process state change, unlike a RESTART, a TROLLBACK does not cause any transfer of control. Because of this, a useful technique is to set a flag in a local variable that is not a restart variable, issue a TRESTART and have a block conditioned on the flag variable which does a TROLLBACK.

A RESTART is a transfer of control to the TSTART at the beginning of the transaction. RESTART implicitly includes a ROLLBACK and may optionally restore local variables, known as restart variables, to the values they had when the initial TSTART originally executed. A RESTART always restores $TEST and the naked reference to the values they had when the initial TSTART executed. RESTART does not manage device state information. A RESTART is invoked by the TRESTART command or by M if it is determined that the transaction is in conflict with other database updates. RESTART can only successfully occur if the initial TSTART includes an argument that enables RESTART, which FIS strongly recommends in order to deal with implicit RESTARTs.

Key Considerations - Writing TP Code

Some key considerations for writing application code between TSTART and TCOMMIT are as follows:

- Do not use BREAK, CLOSE, JOB, OPEN, READ, USE, WRITE, LOCK, HANG, ZEDIT, ZSYSTEM and external calls as they violate the ACID principal of Isolation. Using these commands inside a transaction may lead to longer than usual response time, high CPU utilization, repeat execution due to transaction restart, and/or TPNOTACID messages in the operator log. If application logic requires their use, put them before TSTART or after TCOMMIT so that they do not interfere with the transaction processing mechanism. For example, placing a LOCK before TSTART and releasing it after TCOMMIT provides an additional application layer of serialization for the transaction code. If the user story requires one or more non-ACID operation within a transaction, condition them on 0=$TRESTART so they only processes once, and never while holding a database critical section. If the user story requires a one-to-one relationship between a non-ACID action and a transaction, use TROLLBACK, typically with TRESTART and/or error handling to align them, but be aware this risks a "live-lock" pathology where the action consumes a disproportionate amount of resources while attempting to complete over an extended period.

- Keep your transaction code "pure" . By "pure" we mean that you restrict code to only perform database updates (SET, MERGE, and so on). The primary purpose of a GT.M transaction is to perform database updates that commit in entirety or do not commit at all. Perform external interaction like performing a user interaction or invoking an external call before or after the transaction.

- Design transactions to minimize the number of regions they use, particularly update. Like keeping transactions small, this minimizes contention and improves performance.

- Keep transactions as short as possible.

- Code for handling errors during transactions must include a TROLLBACK. A TROLLBACK should appear as early as possible in the error handling code. You can run commands like WRITE, OPEN, etc. after TROLLBACK because the TROLLBACK releases resources held by the transaction.

- Remember that trigger code executes within an implicit transaction. So, trigger code is always subject to transaction considerations.

Most transaction processing systems try to have transactions that meet the "ACID" test – Atomic, Consistent, Isolated, and Durable.

To provide ACID transactions, GT.M uses a technique called optimistic concurrency control. Each block has a transaction number that GT.M sets to the current database transaction number when updating a block. Application logic, brackets

transactions with TSTART and TCOMMIT commands. Once inside a transaction, a GT.M process tracks each database block that it reads (any database block containing existing data that it intends to update has to be read first) and keeps a list of updates in process private memory that it intends to apply. Application logic within the process views the database as if the transaction updates have been applied; application logic in other processes does not see states internal to the transaction. At TCOMMIT time, the process checks whether any blocks have changed since it read them, and if none have changed, it commits the transaction, making its changes visible to other processes Atomically with Isolation and Consistency (Durability comes from the journal records written at COMMIT time). Optimistic concurrency attempts to exploit the odds that two processes need access to the same resource at the same time. If the chances are small, it permits many processes to work concurrently, particularly in a system with multiple CPUs. If the chances are not small the penalty is repeated execution of the same transaction logic.

If one or more blocks have changed, the process reverts its state to the TSTART and re-executes the application code for the transaction. If it fails to commit the second time, it tries yet again. If it fails to commit on the third attempt, it locks other processes out of the database and executes the transaction as the sole process (that is, on the fourth attempt, it switches to a from an optimistic approach to a pessimistic one).

This technique normally works very well and is one of the factors that allow GT.M to excel at transaction processing throughput.

> **Note**
>
> GT.M uses implicit transaction processing when it needs to ensure complex operations, including spanning block actions, spanning region actions and trigger actions preserve Atomicity. Of these, triggers involve application code and therefore are most subject to the following discussion.

Pathological cases occur when processes routinely modify blocks that other processes have read (called "collisions"), resulting in frequent transaction restarts. Collisions can be legitimate or accidental. Importantly, the longer that a transaction is "open" (the "collision window," when the application logic is between TSTART and TCOMMIT), the greater the probability that a collision, which requires a transaction restart.

Legitimate collisions can result from normal business activity, for example, if two joint account holders make simultaneous ATM withdrawals from a joint account. When the time an application takes to process each transaction is a minuscule fraction of a second, the probability of a collision is very low, and in the rare case where one occurs, the restart mechanism handles it well. An example with a higher probability of collision comes from commercial accounts, where a large enterprise may have tens to hundreds of accounts, individual transactions may hit multiple accounts, and during the business day many people may execute transactions against those accounts. Again, the small collision window means that collisions remain rare and the restart mechanism handles them well when they occur.

Legitimate (from a GT.M point of view) collisions can also occur as a consequence of application design. For example, if an application has an application level transaction journal that every process appends to then that design will likely result in high rates of collisions, creating a pathological case where every transaction fails three times and then commits on the fourth attempt with all other processes locked out. The way to avoid these is to adjust the application design, either to use M LOCKs to gate such "hot spots" or, better, to give each process its own update space which, at some event, a single process then consolidates.

Accidental collisions result when two processes access unrelated data that happens to reside on the same data block. For example, some global indexed by last name can result in an accidental collision if two account holders whose last names start with the same letter, the global data nodes may reside in the same block. Because the path to many data blocks typically pass though at least one index block, data additions cause changes in index blocks and can generate accidental collisions. While it is not possible to avoid accidental collisions (especially in blocks containing metadata such as index blocks), they are typically rare and the occasional collision is handled well by the restart mechanism. Because the application is rarely in a position to

efficiently prevent accidental collisions, FIS strongly recommends using TSTART forms that allow GT.M to use restarts and thus relieve the application logic of having to manage TRESTNOT errors. GT.M uses the database block as the granularity for concurrency control because it is generally an efficient and successful compromise between a more granular and expensive lock and a less granular but more likely to conflict lock. It also simplifies some things by aligning with the unit of transfer to non-volatile storage. When the application guarantees that every update to a global variable (node) comes from a single process, declaring this with the NOISOLATION characteristic can materially improve performance, by allowing GT.M to resolve some conflicts without a full restart.

Application design that keeps transactions open for long periods of time can cause pathological rates of accidental collision. When a process tries to run an entire report in a transaction, instead of the transaction taking a fraction of a second (remember that transactions are intended to be atomic), the report takes seconds or even minutes and effectively ensures collisions and restarts. Furthermore, since the probability of collisions is high, the probability of these long-running transactions executing the fourth retry (with other processes shut out) goes up, and when that happens, the system appears to respond erratically, or hang temporarily.

Non-Isolated actions are another consideration in the design of wholesome transactions. Because M permits all language features with a transaction, an application may use actions that interact with actors outside of the transaction; such actions violate the ACID principal of Isolation, which states to be wholesome a transaction must not interact with other agents or processes until it commits (see below for a more detailed discussion). While there may be reasons drawn from the larger application model that justify violations of Isolation, doing so carries risks. One problem is time, external interactions typically have a longer duration, and in the worst case may have an indefinite duration. The JOB, LOCK, OPEN, and READ commands have an optional timeout to place time limits on external interactions as do some WRITE format arguments. The HANG command induces a potentially arbitrary delay. In addition, BREAK, WRITE, ZEDIT, ZSYSTEM and external calls also involve external interaction. Except for WRITE commands without a timeout and external calls, in order to minimize potential the impact of non-ACID transactions, GT.M limits the duration of database locks for transactions that use these non-Isolated commands, and records that use of that limitation as a TPNOTACID message in the operator log. However, that time limit, managed with the gtm_tpnotacidtime environment variable, can be long enough, depending on its value, to permit noticeable processing disruptions. Further, processes denied a long lock may have trouble completing and consume system resources with repeated unsuccessful attempts. External calls are excluded from this protection because they are the domain of more sophisticated design and may actually remain isolated (see the tip below on Implementing Web Services). WRITE is currently excluded because most un-timed WRITE commands are non-blocking, but applications should avoid blocking WRITEs within a transaction. Beyond the issue of duration, because the application can repeat due to a restart, because of an error or explicit application logic, non-isolated actions require careful design to appropriately manage their external interactions; this is discussed in more detail below. In summary, put external interactions before or after transactions rather than within them. If the application requires a non-Isolated action within a transaction, be aware of the risks, design, implement and test very carefully.

GT.M provides a transaction timeout feature that interrupts long-running transactions in order to limit their impact on the system, and the consequent user perception of system erratic response times and temporary hangs. Calls to an external library, say to access a web service, can subvert the timeout mechanism when the external library uses an uninterruptable system call. If such a web service uses an adjacent server that responds immediately, the web service is wholesome. But if the web service accesses a remote server without a guaranteed short response time, then collisions may be frequent, and if a process in the fourth retry waits for a web service that never responds, it brings the entire application to a standstill.

Implementing Web Services Safely

To safely implement web services inside a transaction, an application must implement a guaranteed upper bound on the time taken by the service. The story or use case for each circumstance determines the appropriate timeout for the corresponding transaction. For example, if the web service is to authorize a transaction, there might be a 500 millisecond timeout with the authorization refused if the approval service does not respond within that time.

There are two approaches to implementing web services with a timeout.

1. For applications that call out to C code, the C code should guarantee a return within a time limit, using a wrapper if necessary. GT.M provides functions that external C code can use to implement timers. If the call is to an third party library, or one without a way to guarantee a timeout, the external C code may need to create an intermediate proxy that can provide a timeout to GT.M.

2. Because web services are usually implemented by a known protocol layered on TCP/IP and GT.M provides a SOCKET device for TCP/IP connections, implement the call out to the web service using a GT.M SOCKET device. GT.M can then enforce the TP timeout mechanism, which it cannot for an external call, especially one that calls via a library into an uninterruptible OS service.

To conform with the M approach of providing maximum flexibility and, when possible, backwards compatibility with older versions of the standard, M transaction processing requires the use of programming conventions that meet the ACID test.

For example, some effects of the BREAK, CLOSE, JOB, OPEN, READ, USE WRITE, ZEDIT, ZSYSTEM commands and external calls may be observed by parties to the system. Because the effects of these commands might cause an observing process or person to conclude that a transaction executing them was in progress and perhaps finished, they violate, at least in theory, if not in practice, the principle of Isolation.

The LOCK command is another example. A program may attempt to use a LOCK to determine if another process has a transaction in progress. The answer would depend on the management of LOCKs within transactions, which is implementation-specific. This would therefore clearly violate the principle of Isolation. The LOCK command is discussed later in this section.

The simplest way to construct a transaction that meets the ACID test is not to use any commands within a transaction whose effects may be immediately "visible" outside the transaction. Unfortunately, because some M applications are highly interactive, this is not entirely straightforward. When a user interaction relies on database information, one solution is for the program to save the initial values of any global values that could affect the outcome, in local variables. Then, once the interaction is over and the transaction has started, the program checks the saved values against the corresponding global variables. If they are the same, it proceeds. If they differ, some other update has changed the information, and the program must issue a TROLLBACK (perhaps after a TRESTART), and initiate another interaction as a replacement.

Even when the "visible" commands appear within a transaction, an M application may provide wholesome operation by relying on additional programming or operating conventions.

A program using LOCKs to achieve serializability relies on properly designed and universally followed LOCKing conventions to achieve Isolation with respect to database operations. LOCKs placed outside the transaction (usually a LOCK immediately before the TSTART and an unlock immediately after the TCOMMIT) achieve serializability by actually serializing any approximately concurrent transaction. LOCKs placed inside the transaction (frequently a LOCK immediately after the TSTART and an unlock immediately before the TCOMMIT) signal M to ensure that no operations using the same LOCK resource(s) overlap. M LOCKs are on resource names that have the same form as variable names, not database actions that lock actual variables, This allows considerable flexability in LOCK protocol design, but does require considerable care. LOCKing protocols typically require appropriate timeout logic to prevent deadlocks. Within a transaction, an M implementation may defer both LOCKing and unlocking to achieve its goal of serializability. A program using TSTARTs with the SERIAL keyword replaces the convention with a guarantee from M that all the database activity of the transaction meets the test of Isolation with respect to database activity.

In GT.M the Durability aspect of the ACID properties relies on the journaling feature. When journaling is on, every transaction is recorded in the journal file as well as in the database. The journal file constitutes a serial record of database actions and

states. It is always written before the database updates and is designed to permit recovery of the database if the database should be damaged. By default, when a process commits a transaction, it does not return control to the application code until the transaction has reached the journal file. The exception to this is that when the TSTART specifies TRANSACTIONID="BATCH" the process resumes application execution without waiting for the file system to confirm the successful write of the journal record. The idea of the TRANSACTIONID="BATCH" has nothing inherently to do with "batch" processing - it is to permit maximum throughput for transactions where the application has its own check-pointing mechanism, or method of recreating the transaction in case of a failure. The real durability of transactions is a function of the durability of the journal files. Putting journal files on reliable devices (RAID with UPS protection) and eliminating common points of failure with the path to the database (separate drives, controllers cabling) improve durability. The use of the replication feature can also improve durability by moving the data to a separate site in real time.

Attempting to QUIT (implicitly or explicitly) from code invoked by a DO, XECUTE, or extrinsic after that code issued a TSTART not yet matched by a TCOMMIT, produces an error. Although this is a consequence of the RESTART capability, it is true even when that capability is disabled. For example, this means that an XECUTE containing only a TSTART fails, while an XECUTE that performs a complete transaction succeeds.

TP Performance

To achieve the best GT.M performance, transactions should:

- be as short as possible

- consist, as much as possible, only of global updates

- be SERIAL with no embedded LOCKs and minimal surrounding LOCKs where justified

- have RESTART enabled with a minimum of local variables protected by a restart portion of the TSTART argument.

- Large concurrent transactions using TCOMMIT may result in repeated and inefficient attempts by competing processes to capture needed scarce resources, resulting in poor performance.

Example:

```
TSTART ():SERIAL
SET (ACCT,^M(0))=^M(0)+1
SET ^M(ACCT)=PREC,^PN(NAM)=ACCT
TCOMMIT
```

This transaction encapsulates these two SETs. The first increments the tally of patients registered, storing the number in local variable ACCT for faster access in the current program, and in global variable ^M(0). The second SET stores a patient record by account number and the third cross-references the account number with the patient name. Placing the SETs within a single transaction ensures that the database always receive either all of the SETs or none of them, thus protecting database integrity against process or system failure. Similarly, another concurrent process, whether using transactions or not, never finds one of the SETs in place without also finding the other one.

Example:

```
TSTART ():SERIAL
IF $TRESTART>3 DO QUIT
.TROLLBACK
.WRITE !,"Too many RESTARTs"
.QUIT
SET (NEXT,^ID(0))=^ID(0)+1
SET ^ID(NEXT)=RECORD,^XID(ZIP,NEXT)=""
```

```
TCOMMIT
```

This transaction automatically restarts if it cannot serialize the SETs to the database, and terminates with a TROLLBACK if more than 3 RESTARTs occur.

GT.M provides a way to monitor transaction restarts by reporting them to the operator logging facility. If the environment variable gtm_tprestart_log_delta is defined, GT.M reports every Nth restart where N is the numeric evaluation of the value of gtm_tprestart_log_delta. If the environment variable gtm_tprestart_log_first is defined, the restart reporting begins after the number of restarts specified by the value of gtm_tprestart_log_first. For example, defining both the environment variable to the value 1, causes all TP restarts to be logged. When gtm_tprestart_log_delta is defined, leaving gtm_tprestart_log_first undefined is equivalent to giving it the value 1.

Here is an example message:

```
%GTM-I-TPRESTART, Database /gbls/dtx/dtx.dat; code: L; blk: 0x00BA13DD in glbl: ^DTX; pvtmods: 0, blkmods: 1,
▶ blklvl: 1, type: 4, readset: 3, writeset: 1, local_tn: 0x00000000000002D0, zpos: LABEL+108^ROUTINENAME
```

▼

- pvtmods - Is always less than or equal to blkmods. This means it can be 1 only if "blkmods" is also 1. If it is 1, it means that process P1 was planning to UPDATE (not just READ) the block number (indicated as "blk: ..." in the TPRESTART message) as part of its TP transaction.

- blkmods - Is either 1 or 0. 1 implies the transaction restarted because this process (P1) is attempting to READ/UPDATE a block that has concurrently been updated by another process (P2) since P1 access the block as part of its TP transaction. This means the "code: ..." output in the TPRESTART message will have L as the last letter. 0 implies the restart occurred because of a different reason. The "code: ..." then has something other than "L" as the last letter. Note that each letter in "code: ..." corresponds to the failure code in each try/retry in the order of occurrence.

- blklvl - Is the level in the GDS structure of the block ("blk: ..." field in the TPRESTART message) that caused the TP restart.

- type - A value of 0,1,2,4 shows the restart occurred in the TP transaction BEFORE executing TCOMMIT; 1 means searching, 2 means reading, 3 means committing, 4 means validating history, and 0 means others.

- readset - The number of GDS blocks that accessed as part of this TP transaction in the region containing the global ("glbl: ..." in the TPRESTART message).

- writeset - Out of the readset number, the number of GDS blocks this process was attempted to UPDATE as part of this TP transaction in the region containing the global ("glbl: ..." in the TPRESTART message).

- local_tn - This is a never-decreasing counter (starting at 1 at process startup) incremented for every new TP transaction, TP restart, and TP rollback. Two TPRESTART messages by the same process should never have the same value of local_tn. The difference between the local_tn values of two messages from the same process indicates the number of TP transactions done by that process in the time interval between the two messages.

> **Note**
>
> - Use VIEW [NO]LOGT[PRESTART][=intexpr] to enable or disable the logging of TPRESTART messages. Note that you can use the gtm_tprestart_log_delta and gtm_tprestart_log_first environment variables to set the frequency of TPRESTART messages.
>
> - Use VIEW [NO]LOGN[ONTP][=intexpr] to enable or disable the logging of NONTPRESTART messages. This facility is the analog of TPRESTART tracking, but for non-TP mini-transacstions. Note that you can

use the gtm_nontprestart_log_delta and gtm_nontprestart_log_first environment variables to set the frequency of the NONTPRESTART messages.

- For more information, refer to "Key Words in VIEW Command" (page 152) and the Environment Variables section of **GT.M Administration and Operations Guide**.

TP Example

Here is a transaction processing example that lets you exercise the concept. If you use this example, be mindful that the functions "holdit" and "trestart" are included as tools to allow you access to information within a transaction which would normally be hidden from users. These types of functions would not normally appear in production code. Comments have been inserted into the code to explain the function of various segments.

```
trans
 ;This sets up the program constants
 ;for doit and trestart
 new
 set $piece(peekon,"V",51)=""
 set $piece(peekon,"V",25)="Peeking inside Job "_$job
 set $piece(peekoff,"^",51)=""
 set $piece(peekoff,"^",25)="Leaving peeking Job "_$job
 ;This establishes the main loop
 set CNFLTMSG="Conflict, please reenter"
 for read !,"Name: ",nam quit:'$length(nam) do
 .if "?"=nam do quit
 ..write !,"Current data in ^trans:",! do:$data(^trans) quit
 ...zwrite ^trans
 .for set ok=1 do quit:ok write !,$char(7),CNFLTMSG,$char(7),!
 ..set old=$get(^trans(nam),"?")
 ..if "?"=old write !,"Not on file" do quit
 ...;This is the code to add a new name
 ...for do quit:"?"'=data
 ....read !,"Enter any info using '#' delimiter: ",!,data
 ...if ""=data write !,"No entry made for ",nam quit
 ...TSTART ():SERIAL if $$trestart ;$$trestart for demo
 ...if $data(^trans(nam)) set ok=^trans(nam)=data TROLLBACK quit
 ...set ^trans(nam)=data
 ...TCOMMIT:$$doit ;$$doit for demo
 ..;This is the beginning of the change and delete loop
 ..for do quit:+fld=fld!'$length(fld) write " must be numeric"
 ...write !,"Current data: ",!,old
 ...read !,"Piece no. (negative to delete record) : ",fld
 ..if 'fld write !,"no change made" quit
 ..;This is the code to delete a new name
 ..if fld<0 do quit ; delete record
 ...for do quit:"YyNn"[x
 ....write !,"Ok to delete ",nam," Y(es) or N(o) <N>? "
 ....read x set x=$extract(x)
 ...if "Yy"'[x!'$length(x) write !,"No change made" quit
 ...TSTART ():SERIAL if $$trestart ;$$trestart for demo
 ...if $get(^trans(nam),"?")'=old TROLLBACK set ok=0 quit
 ...kill ^trans(nam)
 ...TCOMMIT:$$doit; $$doit for demo
```

```
..;This is the code to change a field
..for read !,"Data: ",data quit:("?"'=data)&(data'["#") do
...write " must not be a single '?' or contain any '#'"
..TSTART ():SERIAL if $$trestart ;$$trestart for demo
..if '$data(^trans(nam)) set ok=0 TROLLBACK q
..if $piece(^trans(nam),"#",fld)=$piece(old,"#",fld) do quit
...set ok=$piece(^trans(nam),"#",fld)=data TROLLBACK
..set $piece(^trans(nam),"#",fld)=data
..TCOMMIT:$$doit; $$doit for demo
 quit
doit()
;This inserts delay and an optional
;rollback only to show how it works
 write !!,peekon do disp
 for do quit:"CR"[act
 .read !,"C(ommit), R(ollback), or W(ait) <C>? ",act
 .set act=$translate($extract(act),"cr","CR")
 .if "?"=act do disp
 if "R"=act TROLLBACK write !,"User requested DISCARD"
 write !,peekoff,!
 quit $TLEVEL
trestart()
;This is only to show what is happening
 if $TRESTART do
 .write !!,peekon,!,">>>RESTART<<<",! do disp write !,peekoff,!
 quit 1
disp
 write !,"Name: ",nam
 write !,"Original data: ",!,old,!,"Current data: "
 write !,$get(^trans(nam),"KILLED!")
 quit
```

Generally, this type of program would be receiving data from multiple sessions into the same global.

See Also	
	• gtm_tpnotacidtime
	• "$ZMAXTPTIme" (page 314)
	• "TROllback" (page 149)
	• "TStart" (page 149)
	• "TCommit" (page 147)
	• "$TLevel" (page 300)
	• "$ZTLevel" (page 334)
	• "$ZTExit" (page 328)

Chapter 6. Commands

Revision History		
Revision V7.1-004	27 June 2024	• In "Examples of ZSHow" (page 198), Added STG,KTG,ZTG,DEXA,GLB,JNL,MLK,PRC,TRX,ZAD,JC to the ZSHOW output. • In ""POOLLIMIT":<region>:expr" (page 158), Add clarification • In "ZSHOW Information Codes" (page 193), Provide a database index block statistics • In "ZWRite" (page 209), add details for ZWRITE gvn
Revision V7.1-003	23 November 2023	• In "CMD[LINE]="strlit" " (page 126), Use CMDLINE to identify processes in PS listings
Revision V7.1-002	19 September 2023	• In "Examples of ZSHow" (page 198), Add the new $ZICUVER ISV
Revision V7.1-001	26 June 2023	• In "Open" (page 142), fixed a typo • In "ZCOMpile" (page 175), add reference to $ZCOMPILE • In "ZHALT" (page 180), fix a typo
Revision V7.0-005	02 December 2022	• In "ZSHOW Information Codes" (page 193), GTM-F135406 \nAdding STG,KTG and ZTG counters.
Revision V7.0-003	24 June 2022	• In "For" (page 118), In "[NO]UNDEF", changed a note to reflect that NOUNDEF no longer applies to an undefined FOR control variable. • In "[NO]UNDEF" (page 159), In "[NO]UNDEF", changed a note to reflect that NOUNDEF no longer applies to an undefined FOR control variable.
Revision V7.0-002	23 March 2022	• In "Halt" (page 122), fix typo • In "Hang" (page 122), specify that the actual HANG time may fractionally differ from the specified time • In "Read" (page 143), formatting cleanup • In "ZMessage" (page 189), Remove CTRLY from the list of internal errors • In "ZSHOW Information Codes" (page 193), add statistics(PRG,WFL,WHE,and WRL) for the update process and its helper processes; Clean up code description list for clarity

		• In "ZSTEP Interactions" (page 204), clarify CTRAP and ZSTEP interaction; remove caution about GT.CM neterror
Revision V7.0-001	24 November 2021	• In "ZSHOW Information Codes" (page 193), add DEXA, GLB, JNL, MLK, PRC, TRX, ZAD, JOPA, AFRA, BREA, MLBA and TRGA in ZSHOW Information Codes table; add actions to the ZSHOW "B" output description
Revision V6.3-014	06 October 2020	• In "ZMessage" (page 189), specify that Information and Success messages appear on STDERR. • In "ZSHOW Information Codes" (page 193), corrected CFS and DRD. Added DEXA, GLB, JNL, MLK, PRC, TRX, ZAD, JOPA, AFRA, BREA, MLBA and TRGA.
Revision V6.3-013	30 June 2020	• In "Examples of ZSHow" (page 198), add WFR, BUS, and BTS. • In "ZSHOW Information Codes" (page 193), change wording which implies that statistics are displayed in the same order as the alphabetical table; add WFR, BUS, and BTS stats.
Revision V6.3-011	20 December 2019	• In "Key Words in VIEW Command" (page 152), Add [NO]LOGTPRESTART and tweak [NO]LOGNONTPRESTART
Revision V6.3-010	31 October 2019	• In "ZLink" (page 181), clen up table format
Revision V6.3-008	24 April 2019	• In "Examples of JOB" (page 129), Add an example with OUTPUT and ERROR usage • In "If" (page 123), make a spelling correction • In "Key Words in VIEW Command" (page 152), correct delimiter for VIEW LOCTPRESTART and add section 3 ID • In "[NO]LOGN[ONTP][:intexpr]" (page 155), correct delimiter
Revision V6.3-007	04 February 2019	• In "Examples of ZSHow" (page 198), add $ZTIMEOUT and $ZAUDIT to $zjobexam() output • In "ZSHOW Information Codes" (page 193), mark CQS, CQT, CYS, CYT as * and add a legend; modify CFS definition
Revision V6.3-006	26 October 2018	• In "[NO]BADCHAR" (page 152), UTF-8 mode tweaks. • In "Read" (page 143), minor correction. • In "ZCOMpile" (page 175), specify that ZCOMPILE assumes a default extension of .m if it is not specified with expr

		• In "ZSYstem" (page 205), remove a spurious space in the first sentence.
Revision V6.3-005	29 June 2018	• In "Examples of ZSHow" (page 198), add an example for ZSHOW "I"
		• In "[NO]FULL_BOOL[EAN|WARN]" (page 153), show that "EAN" and "WARN" are alternatives in the syntax description.
		• In "[NO]STATSHARE"[:<region-list>]" (page 158), made minor corrections
		• In "ZGoto" (page 178), add information about ZGOTO 0.
		• In "ZSHOW Information Codes" (page 193), add BTD and place in alpha order with highlighting of characters contribution to the mnemonic
Revision V6.3-004	23 March 2018	• In "Close" (page 115), Add statement about CLOSE $IO.
		• In "View" (page 151), specify that a VIEW sub-argument can be a list of regions.
		• In "[NO]STATSHARE"[:<region-list>]" (page 158), add changes for VIEW "[NO]STATSHARE[:<region-list>]"
		• In "ZSHOW Information Codes" (page 193), Remove duplicate descriptions of ZSHOW "G" output items; specify that lines associated with G end with ? when the process does not have access to the current shared statistics
Revision V6.3-003	12 December 2017	• In "ZSHOW Information Codes" (page 193), add description of T code
Revision V6.3-002	22 August 2017	• In "Lock" (page 132), add BADLOCKNEST description
		• In "Examples of ZSYSTEM" (page 205), add an example that uses quoting.
		• In "Xecute" (page 169), add explanation for compile-time evaluation of literal arguments of XECUTE
Revision V6.3-001	20 March 2017	• In "Else" (page 117), change some wording in search of added clarity
		• In "If" (page 123), added an explanation on behavior of a literal FALSE argument and some words about short-circuiting of Boolean expressions.
		• In "Lock" (page 132), fixed misidentified ISV ($TRRETY) with correct name ($TRESTART).
		• In "PASS[CURLVN]" (page 128), described what happens if a parameter name collides with one of the JOB caller's local variables

		• In "TP Example" (page 106), added a link to the description of the gtm_tpnotacidtime environment variable in the Administration and Operations Guide. • In "View" (page 151), added the LOGNONTP keyword. • In "Examples of ZSHow" (page 198), updated output • In ""JOBPID":"value"" (page 155), alerted the user to the fact that VIEW "JOBPID":1 affects used defined OUTPUT and ERROR parameters to the JOB command. • In "Key Words in VIEW Command" (page 152), added information about the ""FLUSH"[:<region_list>]" [153] keyword. • In "[NO]STATSHARE"[:<region-list>]" (page 158), added the description of the STATSHARE keyword. • In "ZRUPDATE" (page 192), fixed typo and removed the reference to the field test release. • In "ZSHow" (page 193), added information about the ZSHOW output format change for long values. • In "ZSHOW Information Codes" (page 193), corrected the description of A code; added description of additional codes
Revision V6.2-001	27 February 2015	• In "JNLWAIT" (page 154), specified that GT.M ignores JNLWAIT when inside a TP transaction. • Updated "ZSHOW Information Codes" (page 193) for ZSHOW "A" and mnemonics introduced in V6.2-001. • Added a new section called "Auto-ZLINK setup" (page 184). • Added "[NO]DMTERM" (page 153) and ""POOLLIMIT":<region>:expr" (page 158) keywords for the VIEW command. • In "JOB Processparameters" (page 126), added the "PASS[CURLVN]" (page 128) processparameter. • In "Examples of ZPRINT" (page 191), added some examples for runtime disambiguators.
Revision V6.1-000	28 August 2014	• In "Key Words in VIEW Command" (page 152), added the LINK keyword and an example for TRACE. • In "ZLink" (page 181), added the description of recursive relink.

		• In "ZSHOW Information Codes" (page 193), added the description of ZSHOW "R" and removed the description of ZSHOW "C".
		• In "Close" (page 440), added information for closing a listening LOCAL socket.
		• In "Job" (page 125), added information about using DETACHed sockets in INPUT, OUTPUT, and ERROR Processsmarameters.
		• Moved the contents of SET * under SET and KILL * under KILL.
Revision V6.0-003	24 February 2014	• In "Key Words in VIEW Command" (page 152), added DBFLUSH, DBSYNC, and EPOCH keywords.
		• In "Job" (page 125), corrected the description of the STARTUP jobprocessparameter.
		• In "$ZLength()" (page 270), corrected the example of the two argument forms of $ZLENGTH().
Revision V6.0-001	21 March 2013	• In "Lock" (page 132), added a point about implementing fairer access when multiple processes need the same lock resource.
		• In "Argument Keywords of $VIEW()" (page 244), added more information about the GVSTAT argument.
		• Improved the description of "ZMessage" (page 189).
		• In "Job" (page 125), added the description of the CMDLINE Job Processparameter.
		• In "Key Words in VIEW Command" (page 152), added the description of the [NO]LOGTPRESTART keyword.
		• In "[NO]UNDEF" (page 159), added a note about NOUNDEF not applying to an undefined FOR control variable.
		• In "ZTRigger" (page 207), corrected the description of the ZTRIGGER command.
		• In "Xecute" (page 169), added a note about handling run-time errors.
		• In "TREstart" (page 148), added information about the handling of TPRESTARTs in an interrupted error.
Revision V6.0-000	19 November 2012	• In "CLOSE Deviceparameters" (page 441), added the description of the DESTROY deviceparameter.

		• In "ZSHOW Information Codes" (page 193), added information about the new ZSHOW "G" mnemonics.
		• In "Lock" (page 132) and "ZAllocate" (page 170), added updates for V6.0-000.

This chapter describes M language commands implemented in GT.M. All commands starting with the letter Z are GT.M additions to the ANSI standard command set. The M standard specifies standard abbreviations for commands and rejects any non-standard abbreviation. Behavior of I/O commands including OPEN, USE, READ, WRITE, and CLOSE is described in Chapter 9: "Input/Output Processing" (page 340).

Break

The BREAK command pauses execution of the code and initiates Direct Mode.

The format of the BREAK command is:

```
B[REAK][:tvexpr] [expr[:tvexpr][,...]]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The optional expression contains a fragment of GT.M code to XECUTE before the process enters Direct Mode.

- The BREAK command without an argument causes a pause in execution of the routine code and immediately initiates Direct Mode. In this case, at least two (2) spaces must follow the BREAK to separate it from the next command on the line.

- The optional truth-valued expression immediately following the expression is the argument postconditional that controls whether GT.M XECUTEs the argument. If present and true, the process executes the code before entering Direct Mode. If present and false, the process does not execute the code before entering Direct Mode.

- If an argument postconditional is present and true, the process pauses code execution and initiates Direct Mode before and after XECUTing the argument.

- An indirection operator and an expression atom evaluating to a list of one or more BREAK arguments form a legal argument for a BREAK.

Issuing a BREAK command inside an M transaction destroys the Isolation of that transaction. Because of the way that GT.M implements transaction processing, a BREAK within a transaction may cause the transaction to suffer an indefinite number of restarts ("live lock").

Generally, programs in production must not include BREAK commands. Therefore, GT.M provides the ZBREAK and ZSTEP commands, which insert temporary breakpoints into the process rather than the source code. BREAKs inserted with ZBREAK only exist until the image terminates or until explicitly removed by another ZBREAK command. ZSTEP also inserts temporary BREAKs in the image that only exist for the execution of the ZSTEP command. In the GT.M debugging environment, ZBREAKs and ZSTEPs that insert BREAKs provide a more flexible and less error-prone means of setting breakpoints than coding BREAKs directly into a routine. For more information on ZBREAK and ZSTEP, refer to the sections that describe those commands. Any BREAK commands in code intended for production should be conditionalized on something that is FALSE in production, as, unlike ZBREAK commands, GT.M currently has no means to "turn off" BREAK commands.

ZCONTINUE resumes execution of the interrupted program.

GT.M displays messages identifying the source of a BREAK as:

- The body of a program

- A ZBREAK action

- A device EXCEPTION

- A ZSTEP action

The VIEW "BREAKMSG" mask selectively enables or disables these messages. For an explanation of the mask, refer to "View" (page 151). By default, a process executing a GT.M image displays all BREAK messages.

When a process encounters a BREAK, it displays a prompt indicating readiness to process commands in Direct Mode. By default, Direct Mode displays the GTM> prompt. SETting the $ZPROMPT intrinsic special variable alters the prompt.

Examples of BREAK

Example:

```
LOOP0    F  S act=$O(^act(act)) Q:act=""  B:debug  D LOOP1
```

This FOR loop contains a BREAK with a command postconditional.

Example:

```
GTM>ZPRINT ^br
br;
      kill
      for i=1:1:3 do break;
      quit
break;
      write "Iteration ",i,?15,"x=",$get(x,"<UNDEF>"),!
      break:$data(x) "write ""OK"",!":x,"write ""Wrong again"",!":'x
      set x=$increment(x,$data(x))
      quit
GTM>DO ^br
Iteration 1    x=<UNDEF>
Iteration 2    x=0
%GTM-I-BREAK, Break instruction encountered
         At M source location break+2^br
GTM>ZCONTINUE
Wrong again
%GTM-I-BREAK, Break instruction encountered
         At M source location break+2^br
GTM>ZCONTINUE
Iteration 3    x=1
OK
%GTM-I-BREAK, Break instruction encountered
         At M source location break+2^br
```

```
GTM>ZCONTINUE
%GTM-I-BREAK, Break instruction encountered
          At M source location break+2^br
GTM>ZCONTINUE
GTM>
```

This uses a BREAK with both command and argument postconditionals. The actions display debugging messages.

See Also	• "Key Words in VIEW Command" (page 152) • "ZContinue" (page 176)

Close

The CLOSE command breaks the connection between a process and a device.

The format of the CLOSE command is:

```
C[LOSE][:tvexpr] expr[:(keyword[=expr][:...])][,...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The required expression specifies the device to CLOSE.

- The optional keywords specify device parameters that control device behavior; some device parameters take arguments delimited by an equal sign (=). If there is only one keyword, the surrounding parentheses are optional.

- An indirection operator and an expression atom evaluating to a list of one or more CLOSE arguments form a legal argument for a CLOSE.

- CLOSE of the current device ($IO), implicitly uses $PRINCIPAL.

See Also	• "Close" (page 440) • "Deviceparameter Summary Table" (page 444)

Do

The DO command makes an entry in the GT.M invocation stack and transfers execution to the location specified by the entryref.

The format of the DO command is:

```
D[O][:tvexpr] [entryref[(expr|.lvn[,...])][:tvexpr][,...]]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The optional entryref specifies a location (with some combination of label, offset, and routinename) at which execution continues immediately following the DO.

- A DO command without an argument (that is, a DO followed by two (2) spaces) transfers execution to the next line in the routine if that line contains an appropriate number of periods (.) after the optional label and before the required linestart. These periods indicate the current level of "immediate" nesting caused by argumentless DOs. If the line following the DO contains too many periods, GT.M reports an error; if the line following the DO contains too few periods, GT.M ignores the DO command.

- A DO command without an argument stacks the current value of $TEST, in contrast to a DO with an argument, which does not protect the current value of $TEST.

- The optional parameter list enclosed in parentheses ( ) contains parameters to pass to the routine entry point.

- Label invocations using DO do not require parentheses for calls with no actuallist. If DO or a $$ that does not specify an actuallist invokes a label with a formallist, the missing parameters are undefined in the called routine.

> ⚠️ **Warning**
>
> If DO or $$ specifies a routine but no label using an actuallist, then whether that routine's top label has a formallist or not, the actuallist applies to it directly, whereas before the actuallist would "fall through" to the first label with executable code.

- If the DO specifies a parameter list, the entryref location must start with a label and an argument list (M prohibits entryrefs with offsets during parameter passing).

- If an element in the parameter list starts with a period, it specifies an unsubscripted local variable name and the DO passes that variable by reference. Otherwise, the element specifies an expression that the DO evaluates and passes as a value.

- The optional truth-valued expression following the parameter list, or the entryref if the argument contains no parameter list, specifies the argument postconditional and controls whether GT.M performs a DO using that argument.

- An indirection operator and an expression atom evaluating to a list of one or more DO arguments form a legal argument for a DO.

An explicit or implicit QUIT within the scope of the DO, but not within the scope of any other DO, FOR, XECUTE, or extrinsic, returns execution to the instruction following the calling point. This point may be the next DO argument or another command. At the end of a routine, or the end of a nesting level created by an argumentless DO, GT.M performs an implicit QUIT. Any line that reduces the current level of nesting by changing the number of leading periods (.) causes an implicit QUIT, even if that line only contains a comment. Terminating the image and execution of ZGOTO commands are the only ways to avoid eventually returning execution to the calling point.

A DO command may optionally pass parameters to the invoked subroutine. For more information about entryrefs and parameter passing, refer to Chapter 5: "General Language Features of M" (page 68).

Examples of DO

Example:

```
GTM>DO ^%RD
```

This example invokes the routine directory utility program (%RD) from Direct Mode. The caret symbol (^) specifies that the DO command invokes %RD as an external routine.

Example:

```
GTM>DO A(3)
```

This example invokes the subroutine at label A and passes the value 3 as a parameter. The DO argument does not have a caret symbol (^), therefore, it identifies A as a label in the current routine.

Example:

```
ReportA ; Label for ReportA
        SET di="" OPEN outfile USE outfile
        FOR  SET di=$ORDER(^div(di)) QUIT:di=""  DO PREP DO  DO POST
        .SET de="",(nr,gr)=0
        .WRITE "Division ",di,! F   S de=$ORDER(^de(di,de)) QUIT:de=""   DO
        ..WRITE "Department ",de," Gross Rev: ",^grev(di,de),!
        ..WRITE "Department ",de," Net Rev: ",^nrev(di,de),!
        ..SET gr=gr+^grev(di,de),nr=nr+^nrev(di,de)
        .W "Division Gross Rev: ",gr,!,"Division Net Rev: ",nr,!
         DO PRINT^OUTPUT(outfile)
         QUIT
```

This routine first uses a DO with a label argument (PREP) to do some pre-processing. Then, it uses an argumentless DO to loop through each division of a company to format a report. Within the first argumentless DO, a second argumentless DO (line 4) loops through and formats each department within a division. After the processing of all departments, control returns to the first argumentless DO, which prints a summary of the division. Following processing of all divisions, a DO with a label argument (POST) does some post-processing. Finally, at the next-to-last line, the routine uses a DO that invokes a subroutine at a label (PRINT) in an external routine (^OUTPUT), passing the name of the output file (outfile) as a parameter.

Example:

```
GTM>zprint ^SQR
SQR(z);
  set revert=0
  if $view("undef") set revert=1 view "noundef"
  if z="" write "Missing parameter.",!     view:revert "undef" quit
  else  write z*z,! view:revert "undef" quit
GTM>do ^SQR(10)
100
GTM>do ^SQR
Missing parameter.
```

This examples demonstrates label invocations using DO with and without parentheses.

Else

ELSE executes the remainder of the line after the ELSE if $TEST is FALSE (0). GT.M does not execute the rest of the line if $TEST is TRUE (1).

The format of the ELSE command is:

```
E[LSE]
```

- Because ELSE is a conditional command, it does not support a command postconditional.

- The scope of the ELSE is the remainder of the line. The scope of an ELSE can be extended with DO (or XECUTE) commands.

- Because the ELSE has no argument, at least two (2) spaces must follow the command to separate it from the next command on the line.

Because the scopes of both the IF and the ELSE commands extend to the rest of the GT.M line, placing an ELSE on the same line as the corresponding IF cannot achieve the desired result (unless the intent of the ELSE is to test the result of a command using a timeout). If an ELSE were placed on the same line as its corresponding IF, then the expression tested by the IF would be either TRUE or FALSE. If that condition is TRUE, the code following the ELSE would not execute; if that condition is FALSE, the ELSE would not be in the execution path.

ELSE is analogous to IF '$TEST, except the latter statement switches $TEST to its complement and ELSE never alters $TEST.

> ## ⚠️ Caution
>
> Use ELSE with care. Because GT.M stacks $TEST only at the execution of an extrinsic or an argumentless DO command, any XECUTE or DO with an argument has the potential side effect of altering $TEST. For information about $TEST, refer to "$Test" (page 300).

Examples of ELSE

Example:

```
If x=+x Set x=x+y
Else  Write !,x
```

The IF command evaluates the conditional expression x=+x and sets $TEST. If $TEST=1 (TRUE), GT.M executes the commands following the IF. The ELSE on the following line specifies an alternative action to take if the expression is false.

Example:

```
If x=+x Do ^GOFISH
Else  Set x=x_"^"_y
```

The DO with an argument after the IF raises the possibility that the routine ^GOFISH changes the value of $TEST, thus making it possible to execute both the commands following the IF and the commands following the ELSE.

Example:

```
Open dev::0 Else  Write !,"Device unavailable" QUIT
```

This ELSE depends on the result of the timeout on the OPEN command. If the OPEN succeeds, it sets $TEST to one (1) and GT.M skips the rest of the line after the ELSE. If the OPEN fails, it sets $TEST to zero (0), and GT.M executes the remainder of the line after the ELSE.

For

The FOR command provides a looping mechanism in GT.M. FOR does not generate an additional level in the M standard stack model.

The format of the FOR command is:

```
F[OR][lvn=expr[:numexpr1[:numexpr2]][,...]]]
```

- Because FOR is a conditional command, it does not support a command postconditional.

- The scope of the FOR is the remainder of the line. The scope of a FOR can be extended with DO (or XECUTE) commands.

- When the FOR has no argument, at least two (2) spaces must follow the command to separate it from the next command on the line. This specifies a loop that must be terminated by a QUIT, HALT, GOTO, or ZGOTO.

- The optional local variable name specifies a loop control variable delimited by an equal sign (=). A FOR command has only one control variable, even when it has multiple arguments.

- When initiating the FOR, GT.M assigns the loop control variable the value of the expression. When only an initial value appears, GT.M executes the remainder of the line once for that argument without forcing the control variable to be numeric.

- If the argument includes an increment and, optionally, a terminator, GT.M treats the initial expression as a number.

- The optional numeric expression after the first colon (:) delimiter specifies the increment for each iteration. The FOR command does not increment the control variable on the first iteration.

- The optional numeric expression after the second colon (:) delimiter specifies the limiting value for the control variable. This terminating expression is evaluated only when the control variable is initialized to the corresponding initial value, then used for all subsequent iterations.

- GT.M does not execute the commands on the same line following the FOR if:

  The increment is non-negative and the initial value of the control variable is greater than the limiting value.

  The increment is negative and the initial value of the control variable is less than the limiting value.

- After the first iteration, GT.M does not alter the control variable and ceases execution under the control of the FOR if:

  The increment is non-negative, and altering the control variable by the increment would cause the control variable to be greater than the limiting value.

  The increment is negative, and altering the control variable by the increment would cause the control variable to be less than the limiting value.

- When the FOR has multiple arguments, each one affects the loop control variable in sequence. For an argument to gain control, no prior argument to the FOR can have an increment without a limit.

Increments and limits may be positive, negative, an integer, or a fraction. GT.M never increments a FOR control variable "beyond" a limit. Other commands may alter a control variable within the extended scope of a FOR that it controls. When the argument includes a limit, such modification can cause the FOR argument to yield control at the start of the next iteration, or, less desirably loop indefinitely.

NOUNDEF applies even in the case of an undefined FOR control variable, such as when a KILL or NEW command is used on the control variable, which may cause an unintended indefinite loop. For example, FOR A=1:1:10 KILL A results in an indefinite loop with VIEW "NOUNDEF".

GT.M terminates the execution of a FOR when it executes an explicit QUIT or a GOTO (or ZGOTO in GT.M) that appears on the line after the FOR. FOR commands with arguments that have increments without limits and argumentless FORs can be indefinite loops. Such FORs must terminate with a (possibly postconditional) QUIT or a GOTO within the immediate scope of the FOR. FORs terminated by such commands act as "while" or "until" control mechanisms. Also, such FORs can, but seldom, terminate by a HALT within the scope of the FOR as extended by DOs, XECUTEs, and extrinsics.

Examples of FOR

Example:

```
GTM>Kill i For i=1:1:5 Write !,i
1
2
3
4
5
GTM>Write i
5
GTM>
```

This FOR loop has a control variable, i, which has the value one (1) on the first iteration, then the value two (2), and so on, until in the last iteration i has the value five (5). The FOR terminates because incrementing i would cause it to exceed the limit. Notice that i is not incremented beyond the limit.

Example:

```
GTM>FOR x="hello",2,"goodbye" WRITE !,x
hello
2
goodbye
GTM>
```

This FOR loop uses the control variable x and a series of arguments that have no increments or limits. Notice that the control variable may have a string value.

Example:

```
GTM>For x="hello":1:-1 Write !,x
GTM>ZWRite x
x=0
GTM>
```

Because the argument has an increment, the FOR initializes the control variable x to the numeric evaluation of "hello" (0). Then, GT.M never executes the remainder of the line because the increment is positive, and the value of the control variable (0) initializes to greater than the limiting value (-1).

Example:

```
GTM>For y=-1:-3:-6,y:4:y+10,"end" Write !,y
-1
-4
-4
0
4
end
GTM>
```

This FOR uses two limited loop arguments and one value argument. The first argument initializes y to negative one (-1), then increments y to negative four (-4). Because another increment would cause y to be less than the limit (-6), the first argument terminates with y equal to negative four (-4). The second argument initializes the loop control variable to its current value and establishes a limit of six (6=-4+10). After two iterations, incrementing y again would cause it to be greater than the limit (6), so

the second argument terminates with y equal to four (4). Because the final argument has no increment, the FOR sets y to the value of the third argument, and GT.M executes the commands following the FOR one more time.

Example:

```
GTM>Set x="" For  Set x=$Order(ar(x)) Quit:x=""  Write !,x
```

This example shows an argumentless FOR used to examine all first level subscripts of the local array ar. When $ORDER() indicates that this level contains no more subscripts, the QUIT with the postconditional terminates the loop.

Goto

The GOTO command transfers execution to a location specified by its argument.

The format of the GOTO command is:

```
G[OTO][:tvexpr] entryref[:tvexpr][,...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The required entryref specifies the target location for the control transfer.

- The optional truth-valued expression immediately following the entryref specifies the argument postconditional, and controls whether GT.M performs a GOTO with that argument.

- Additional commands on a line following a GOTO do not serve any purpose unless the GOTO has a postconditional.

- An indirection operator and an expression atom evaluating to a list of one or more GOTO arguments form a legal argument to a GOTO.

A GOTO command within a line following a FOR command terminates that FOR command.

For more information on entryrefs, refer to Chapter 5: "General Language Features of M" (page 68).

Examples of GOTO

Example:

```
GTM>GOTO TIME+4
```

This GOTO command transfers control from Direct Mode to the line that is four (4) lines after the line labeled TIME (in the currently active routine). Using an offset is typically a debugging technique and rarely used in production code.

Example:

```
GOTO A:x<0,^A:x=0,A^B
```

This GOTO command transfers control to label A in the current routine, if x is less than zero (0), to routine ^A if x is equal to zero (0), and otherwise to label A in routine ^B. Once any of the transfers occurs, the rest of the arguments have no effect.

See Also

- "Entry References" (page 91)

Halt

The HALT command stops the program execution and cause GT.M to return control to the operating system environment that invoked the GT.M image.

The format of the HALT command is:

```
H[ALT][:tvexpr]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether GT.M executes the command.

- Because the HALT command has no argument, at least two (2) spaces must follow the command to separate it from the next command on the line. Note that additional commands do not serve any purpose unless the HALT has a postconditional.

A HALT releases all shared resources held by the process, such as devices OPENed in GT.M, databases, and GT.M LOCKs. If the process has an active M transaction (the value of $TLEVEL is greater than zero (0)), GT.M performs a ROLLBACK prior to terminating.

Because HALT and HANG share the same abbreviation (H), GT.M differentiates them based on whether an argument follows the command.

Example:

```
$ gtm
GTM>HALT
$
```

Because we invoke this GT.M image interactively, the HALT in Direct Mode leaves the process at the shell prompt.

Hang

The HANG command suspends GT.M program execution for a period of time specified by the command argument.

The format of the HANG command is:

```
H[ANG][:tvexpr] numexpr[,...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The numeric expression specifies the time in seconds to elapse before resuming execution; depending on the CPU scheduling algorithm of the operating system, the actual elapsed time may sometimes fractionally differ from the specified time. If the numeric expression is negative, HANG has no effect.

- An indirection operator and an expression atom evaluating to a list of one or more HANG arguments form a legal argument to a HANG.

A process that repeatedly tests for some event, such as a device becoming available or another process modifying a global variable, may use a HANG to limit its consumption of computing resources.

Because HALT and HANG share the same abbreviation (H), GT.M differentiates them based on whether an argument follows the command.

Examples of HANG

Example:

```
For  Quit:$Data(^CTRL(1))  Hang 30
```

This FOR loop repeatedly tests for the existence of ^CTRL(1), and terminates when that global variable exists. Otherwise the routine HANGs for 30 seconds and tests again.

Example:

```
SET t=1 For  Quit:$Data(^CTRL(1))  Hang t If t<30 Set t=t+1
```

This is similar to the previous example, except that it uses an adaptive time that lengthens from 1 second to a limit of 30 seconds if the routine stays in the loop.

If

The IF command provides conditional execution of the remaining commands on the line. When IF has an argument, it updates $TEST with the truth value of its evaluated argument. GT.M executes the remainder of a line after an IF statement when $TEST is 1 (TRUE). When $TEST is 0 (FALSE), GT.M does not execute the rest of the line. When the IF argument evaluates to a literal FALSE (0), GT.M discards the command and its arguments at compile time, which means it does not perform any validity checking on the remainder of the line.

The format of the IF command is:

```
I[F] [tvexpr[,...]]
```

- Because IF is a conditional command, it does not support a command postconditional.

- The scope of the IF is the remainder of the line. The scope of an IF can be extended with DO (or XECUTE) commands.

- The action of IF is controlled by the value of the expression and by $TEST, if there is no expression.

- IF with no argument acts on the existing value of $TEST (which it does not change); in this case, at least two (2) spaces must follow the IF to separate it from the next command on the line.

- An indirection operator, and an expression atom evaluating to a list of one or more IF arguments form a legal argument to IF.

> **Note**
>
> Commands with timeouts also maintain $TEST. For information about $TEST, refer to Chapter 8: "Intrinsic Special Variables" (page 291). Because GT.M stacks $TEST only at the execution of an extrinsic or an argumentless DO command, any XECUTE or DO with an argument has the potential side effect of altering $TEST.

> Use the argumentless IF with caution.

Example:

```
IF A,B ...
is equivalent to
IF A IF B
```

An IF with more than one argument behaves as if those arguments were logically "ANDed." However, execution of the line ceases with the evaluation of the first false argument. For IF argument expressions containing the "AND" operator (&), by default, execution still ceases with the evaluation of the first false argument, however any global references within the expression act in sequence to maintain the naked reference. The "FULL_BOOLEAN" and "SIDE_EFFECTS" compiler settings modify this behavior if you desire GT.M to provide side effects it would otherwise bypass due to short-circuiting of Boolean expressions.

Postconditionals perform a function similar to IF; however, their scope is limited to a single command or argument, and they do not modify $TEST. For more information on postconditionals, see Chapter 5: "General Language Features of M" (page 68).

Examples of If

Example:

```
IF x=+x!(x="") Do BAL
```

In this example, the DO executes if x contains a number or a null string.

Example:

```
Write !,?50,BAL If 'BAL Write "****"
IF  Set EMPTY(acct)=""
```

The IF in the first line changes the value of $TEST, determining the execution of the code following the argumentless IF in the second line. Such argumentless IFs may serve as a form of line continuation.

Example:

```
GTM>Set X=1,Y=1,Z=2 Kill UNDEF
GTM>If X=1,Y=1,Z=3,UNDEF=0 Write "HI"
GTM>
```

The IF command causes GT.M to cease executing the line after it determines Z is not equal to three (3). Therefore, GT.M never evaluates the reference to the undefined variable and never generates an error.

Example:

```
GTM>Set X=1 Kill UNDEF
GTM>If X=1!(UNDEF=3) Write "HI"
HI
GTM>
```

Because GT.M recognizes that the X=1 fulfills the IF, it skips evaluation of the UNDEF variable and executes this IF command without generating an error. Because GT.M does not require such optimizations and in fact discourages them by requiring that all global references maintain the naked indicator, other implementations may generate an error.

Job

The JOB command initiates another GT.M process that executes the named routine.

$ZJOB is set to the pid of the process created by the JOB command. For more details, refer to "$ZJob" (page 312).

The format of the JOB command is:

```
J[OB][:tvexpr] entryref[(expr[,...])]
[:[(keyword[=value][:...])][:numexpr]][,...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The required entryref specifies a location at which the new process starts.

- The optional parameter list enclosed in parentheses () contains parameters to pass to the routine entry point.

- If the JOB specifies a parameter list, the entryref location must start with a label and a formallist. M prohibits entryrefs with offsets during parameter passing.

- The optional elements in the parameter list specify expressions that the JOB evaluates and passes as values; because the JOB command creates a new process, its arguments cannot specify pass-by-reference.

- The keywords specify optional processparameters that control aspects of the environment for the new process.

- If the JOB command has only one processparameter, the surrounding parentheses are optional.

- Some keywords take numeric or string literals delimited by an equal sign (=) as arguments. Because the values are constants, strings must be enclosed in quotation marks (" "), and variable arguments require that the entire argument be constructed and referenced using indirection.

- The optional numeric expression specifies a time in seconds after which the command should timeout if unsuccessful; 0 results in a single attempt.

- When a JOB command contains no processparameters, double colons (::) separate the time-out numeric expression from the entryref.

- An indirection operator and an expression atom, evaluating to a list of one or more JOB command arguments, form a legal argument for a JOB command.

- The maximum command-line length for a JOB command is 8192 bytes.

- If the parent process is operating in UTF-8 mode, the JOB'd process also operates in UTF-8 mode.

- If your background process must have a different mode from its parent, then create a shell script to alter the environment as needed, and spawn it with a ZSYstem command using ZSYstem "/path/to/shell/script &".

The operating system deletes the resultant process when execution of its GT.M process is complete. The resultant process executes asynchrounously with the current process. Once GT.M starts the resultant process, the current process continues.

If a JOB command specifies a timeout, and GT.M creates the resultant process before the timeout elapses, JOB sets $TEST to true (1). If GT.M cannot create the process within the specified timeout, JOB sets $TEST to false (0). If a JOB command does not specify a timeout, the execution of the command does not affect $TEST.

If GT.M cannot create the process because of something that is unlikely to change during the timeout interval, such as invalid DEFAULT directory specification, or the parameter list is too long, the JOB command generates a run-time error. If the command does not specify a timeout and the environment does not provide adequate resources, the process waits until resources become available to create the resultant process.

The JOB Environment

When the JOB is forked, UNIX creates the environment for the new process by copying the environment of the process issuing the JOB command and making a few minor modifications. By default, the standard input is assigned to the null device, the standard output is assigned to routinename.mjo, and the standard error is assigned to routinename.mje.

JOB Implications for Directories

By default, GT.M uses the current working directory of the parent process for the working directory of the initiated process.

If the files specified by processsparameters, do not exist, and GT.M does not have permission to create them, the JOBed process terminates. When the corresponding files are in the current working directory, the OUTPUT, INPUT, and ERROR processsparameters do not require a full pathname.

JOB Processsparameters

The following sections describe the processsparameters available for the JOB command in GT.M.

CMD[LINE]="strlit"

The string literal specifies the $ZCMDLINE of the JOB'd process.

Note that the contents of CMD[LINE] appear in process listings. This can be useful for application administrators to identify process purpose. Be aware of the risks of exposing protected information by including that in CMD[LINE].

DEF[AULT]=strlit

The string literal specifies the default directory.

The maximum directory length is 255 characters.

If the JOB command does not specify a DEFAULT directory, GT.M uses the current default directory of the parent process.

ERR[OR]=strlit

strlit specifies the stderr of the JOBbed process. strlit can either be a file or a DETACHed socket (that is, a socket from the socket pool). To pass a DETACHed socket as the stderr of the JOBbed process, specify strlit in the form of "SOCKET:<handle>"

where <handle> is the socket handle. On successful completion of the JOBbed process, the passed socket is closed and is no longer available to the parent process.

The maximum string length is 255 characters.

By default, JOB constructs the error file from the routinename using a file extension of .mje: the default directory of the process created by the JOB command.

GBL[DIR]=strlit

The string literal specifies a value for the environment variable gtmgbldir.

The maximum string length is 255 characters.

By default, the job uses the same specification for gtmgbldir as that defined in $ZGBLDIR for the process using the JOB command.

IN[PUT]=strlit

strlit specifies the stdin of the JOBbed process. strlit can either be a file or a DETACHed socket (that is, a socket from the socket pool). To pass a DETACHed socket as the stdin of the JOBbed process, specify strlit in the form of "SOCKET:<handle>" where <handle> is the socket handle. On successful completion of the JOB command, the passed socket is closed and is no longer available to the parent process.

> **Note**
>
> Specify a DETACHed socket in both INPUT and OUTPUT parameters to pass it as the $PRINCIPAL of the JOBbed process.

The maximum string length is 255 characters.

GT.M does not supply a default file extension.

By default, the job takes its input from the null device.

OUT[PUT]=strlit

strlit specifies the stdout of the JOBbed process. strlit can either be a file or a DETACHed socket (that is, a socket from the socket pool). To pass a DETACHed socket as the stdout of the job, specify strlit in the form of "SOCKET:<handle>" where <handle> is the socket handle. On successful completion of the JOB command, the passed socket is closed and is no longer available to the parent process.

> **Note**
>
> Specify a DETACHed socket in both INPUT and OUTPUT parameters to pass it as the $PRINCIPAL of the JOBbed process.

The maximum string length is 255 characters.

By default, JOB constructs the output file pathname from the routinename using a file extension of .mjo and the current default directory of the process created by the JOB command.

PASS[CURLVN]

With the PASSCURLVN jobparameter, the JOB'd process inherits the current collation, local variables, aliases, and alias containers from the current stack level of the parent process. Therefore, a ZWRITE in the JOB'd process has the same output, except for any out of scope aliases, as a ZWRITE in the context of the JOB command. If the JOB command finds a ZWRITE representation of any lvn, consisting of its full name, its subscripts, corresponding value, quotes and the equal-sign (=), exceeding 1MiB, it produces a JOBLVN2LONG error in the parent process, and a JOBLVNDETAIL error in the error output stream of the JOB'd process. If a JOB command does not specify PASSCURLVN, the JOB'd process(es) inherits no local variables from the parent, although it can receive values passed as parameters to an actuallist entryref. While not an inexpensive command, you can use the "exclusive" NEW command to control the context passed to the JOB'd process; for example, adding "NEW (LOCALA,LOCALB)" before the JOB command would pass only LOCALA and LOCALB.

If a parameter in the formal list of JOB'ed entryref shares the same name with a local in the parent process, the parameter passing facility applies the actuallist in the JOB command argument to the formallist at the invoked label superseding any local variable passed from the parent process by the PASSCURLVN option.

STA[RTUP]="/path/to/shell/script"

Specifies the location of the shell script that executes before running the named routine.

The JOBbed process spawns a shell session to execute the shell script. If the shell script fails, the JOB'd process terminates without running the named routine. Because STARTUP executes in a separate shell, it has no impact on the environment of the JOB'd process, which is inherited from the parent. STARTUP is useful for actions such as creating directories. Use PIPE devices instead of the JOB command to control the environment of a spawned process.

JOB Processparameter Summary Table

The processparameters are summarized in the following table.

JOB Processparameters			
PARAMETER	DEFAULT	MINIMUM	MAXIMUM
DEF[AULT]=strlit	Same directory as the process issuing the JOB command	none	255 characters
ERR[OR]=strlit	./routinename.mje	none	255 characters
GBL[DIR]	Same as gtmgbldir for the process issuing the JOB command	none	255 characters
IN[PUT]=strlit	Null device	none	255 characters
OUT[PUT]=strlit	./routinename.mjo	none	255 characters
PASS[CURLVN]	Only pass any formallist values	N/A	ZWRITE key/value representations of any lvn must not exceed 1MiB
STA[RTUP]=strlit	none	none	Determined by the maximum length a file pathname can have on the operating system,

JOB Processparameters			
PARAMETER	**DEFAULT**	**MINIMUM**	**MAXIMUM**
			which is at least 255 bytes on all systems which GT.M currently supports.

Examples of JOB

Example:

```
GTM>JOB ^TEST("V54001","")
```

This creates a job that starts doing the routine ^TEST (with 2 parameters) in the current working directory.

Example:

```
JOB PRINTLABELS(TYPE,PRNTR,WAITIM)
```

This passes three values (TYPE, PRNTR, and WAITIM) to the new job, which starts at the label PRINTLABELS of the current routine.

Example:

```
set jout="serverjob.mjo"
set jerr="serverjob.mje"
job @("check(a,b):(OUTPUT="""_jout_""":ERROR="""_jerr_""")")
```

This passes two values (a and b) to the new job, which starts at the label check of the current routine. It also specifies that the stdout of the jobbed process to be the file name stored in the local variable jout and stderr to be the file name stored in the local variable jerr.

Example:

Refer to the sockexamplemulti31.m program in Using Socket Devices section for more examples on the JOB command.

See Also	• "$ZJOBEXAM()" (page 268) • "$ZJob" (page 312)

Kill

The KILL command deletes local or global variables and their descendant nodes.

The format of the KILL command is:

```
K[ILL][:tvexpr] [glvn | (glvn[,...]) | *lname | *lvn ]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The optional global or local variable name specifies the variable to delete; KILL deletes not only the variable specified in the argument, but also all variables descended from that variable, that is, those starting with the identical key-prefix.

- KILLing a variable that does not currently exist has no effect.

- The KILL command without an argument deletes all currently existing local variables; in this case, at least two (2) spaces must follow the KILL to separate it from the next command on the line.

- When a KILL argument consists of local variable names enclosed in parentheses, that "exclusive" KILL deletes all local variables except those listed in the argument.

- KILL does not affect copies of local variables that have been "stacked" by NEW or parameter passing with the possible exception of the following:

  For KILL arguments enclosed in parentheses, the environment variable gtm_stdxkill enables the standard-compliant behavior to kill local variables in the exclusion list if they had an explicit or implicit (pass-by-reference) alias not in the exclusion list. By default, this behavior is disabled. If gtm_stdxkill is set to 1,"TRUE", or "YES", KILL deletes a local variable unless all its names are in the parenthesized list. If gtm_stdxkill is not defined or set to 0 KILL operations exclude the data associated with an item if any one of its names appears in the parenthesized list. While non-standard, the default behavior decouples call-by-reference functions or functions using aliases from needing knowledge of the caller's parameters.

- In conformance with the M standard, KILL of a variable joined by pass-by-reference to a formallist variable always KILLs the formalist variable when the actuallist variable is KILL'd even if the formallist variable is specified as protected by an exclusive KILL.

- KILL * removes the association between its argument and any associated arrays. The arguments are left undefined, just as with a standard KILL. If the array has no remaining associations after the KILL *, GT.M can reuse the memory it occupied. If there are no array(s) or association(s) the KILL * happily and silently does nothing.

- KILL * of an alias container variable is just like a KILL of an alias variable, and deletes the association between the lvn and the array.

- KILL * treats an alias formed though pass-by-reference the same as any alias variable by removing the alias association.

- KILL * with no arguments removes all aliases and alias containers connections.

- You can intermix KILL and KILL * in an argument list. For example, KILL *A,B

- Kill * is not permitted inside a parenthesized list of exclusions, e.g.: KILL (*A) is an error.

- An exclusive KILL where one associated name is inside the parenthetic list of exclusions and another associated name is not with that list kills the array through the name that is not inside the list. The association, however, is preserved.

- For more information and KILL * examples, refer to "Alias Variables Extensions" (page 11).

- An indirection operator and an expression atom evaluating to a list of one or more KILL arguments form a legal argument for a KILL.

⚠ Caution

> Use KILL with caution because it can have a major impact on the process environment (local variables) or shared data (global variables).

Examples of KILL

Example:

```
GTM>Kill  Set a=0,a(1)=1,a(1,1)="under" KILL a(1) ZWR
a=0
GTM>
```

This uses an argumentless KILL to get a "fresh start" by deleting all existing local variables. After SETting a, a(1), and a(1,1), the KILL deletes a(1) and its descendants. The ZWRITE shows only a remaining.

Example:

```
GTM>Kill (a,b),^AB(a,b)
```

The first argument (an exclusive KILL) specifies to KILL all local variables except a and b. The second argument deletes ^AB(a,b) and any descendants of that global variable node.

Example:

```
    kill *
    write !,"gtm_stdxkill=",+$ztrnlnm("gtm_stdxkill"),!
    set (A,B,C,E)="input"
    do X(.A,.B)
    zwrite
    write !,"_____",!
    set (A,B,C,E)="input"
    do Y(.A,.B)
    zwrite
    write !,"_____",!
    set (A,B,C,E)="base"
    set *C=A,*D=B
    kill (C,D)
    zwrite
    quit
X(C,D)    set (C,D)="output"
    kill (C,D)
    quit
Y(C,D)    set (C,D)="output"
    kill (A,C,D)
    quit
```

Produces the following output:

```
gtm_stdxkill=0
A="output"
B="output"
C="input"
_____
A="output"
B="output"
C="input"
_____
A="base" ;*
B="base" ;*
*C=A
*D=B
```

See Also

Lock

The LOCK command reserves and releases resource names, and provides a semaphore capability for GT.M processes. This capability can be used for interprocess synchronization and signaling.

Assigning a LOCK does not specify any explicit control over variables and does not directly effect either read or write access to global (or local) data. However, an application that adheres to clearly defined conventions of LOCKing before any access can indirectly achieve such an effect.

FIS recommends implementing database Consistency using transaction processing rather than LOCKs. If you wish to avoid GT.M's use of optimistic concurrency for TP, place the LOCK just before the original TSTART and release it after the final TCOMMIT.

The format of the LOCK command is:

```
L[OCK][:tvexpr] [[-|+]nref|(nref[,...])[:numexpr] [,...]]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The nref argument specifies a resource name in the format of the GT.M name, with or without subscripts and with or without a preceding caret (^). An nref can optionally have an environment specification, including one without a preceding caret (^).

- Outside of transactions, only one process in an environment can own a particular LOCK at any given time.

- Because the data storage in GT.M uses hierarchical sparse arrays, and LOCK frequently serves to protect that data from inappropriate "simultaneous" access by multiple processes, LOCK treats resource names in a hierarchical fashion; a LOCK protects not only the named resource, but also its ancestors and descendants.

- When one or more nrefs are enclosed in parentheses (), LOCK reserves all the enclosed names "simultaneously," that is, it reserves none of them until all become available.

- A LOCK with no argument or an argument with no leading sign releases all names currently reserved with previous LOCK commands by the process; when a LOCK has no argument, at least two (2) spaces must follow the LOCK to separate it from the next command on the line.

- A LOCK argument with a leading plus sign (+) acquires the named resources without releasing currently held resources; if the named resource is already LOCKed, such a LOCK "counts up" the process interest in the resource.

- A LOCK argument with a leading minus sign (-) "counts down" the process interest in a named resource; if the count on a particular lock reaches zero (0), GT.M releases the lock without releasing any other currently held locks; a LOCK that releases a named resource not currently owned by the process has no effect.

- GT.M allows the "process interest" lock counter on a named resource to increment up to 511.

- The optional numeric expression specifies a time in seconds after which the command should timeout if unsuccessful; 0 provides a single attempt; timed LOCK commands maintain $TEST: 1 for a successful LOCK action, 0 for an unsuccessful (within the specified time) LOCK action. Note that untimed LOCK commands do not change $TEST.

- A LOCK operation that finds no room in LOCK_SPACE to queue a waiting LOCK so another process releasing a blocking LOCK can wake it, does a slow poll waiting for LOCK_SPACE to become available. If LOCK does not acquire the ownership of the named resource with the specified timeout, it returns control to the application with $TEST=0. If timeout is not specified, LOCK continues slow poll till space becomes available.

- If a LOCK command in a TP transaction specifies no timeout or a timeout that exceeds the limit specified by $gtm_tpnotacidtime when 2 is less than $TRESTART, the process releases the database critical sections and generates TPNOACID messages, which may live-lock the process, possibly until the transaction terminates because it reaches $ZMAXTPTIME. While such a process may have an impact on system performance this behavior moderates the impact of potential deadlocks on other database operations.

- An indirection operator and an expression atom evaluating to a list of one or more LOCK arguments form a legal argument for a LOCK.

GT.M records LOCK and ZALLOCATE information in the "lock database." GT.M distributes the lock database in space associated with the database identified by the current Global Directory. However, the lock database does not overlap or coincide with the body of the database files holding the global data. Only the LOCK, ZALLOCATE and ZDEALLOCATE commands, and the LKE utility program access the lock database.

GT.M maps reservations of names starting with ^ to the database file used to map global variables of the same name. If the Global Directory maps the name A to file A.DAT, GT.M maps all reservations on ^A to file space associated with A.DAT.

GT.M maps reservations on names not starting with ^ to the region of the database specified with the GDE command LOCK -REGION=. By default, when GDE creates a Global Directory any reservations of local names are mapped to the region DEFAULT.

These two factors effect the following result in the programming environment:

- ^ reservations automatically intersect for all users of the same data in any database file independent of the Global Directory mapping that file.

- reservations without a leading ^ intersect in an arbitrary pattern dependent on the Global Directory and therefore controlled by a design decision potentially made independently of application code design.

Since GT.M uses resource names as semaphores for signaling among multiple processes in a database environment, they interlock in a tree structured fashion. When LOCK or ZALLOCATE reserves a subscripted resource name such as ^D(1), other users of the database mapped by the LOCKing (or ZALLOCATEing) process cannot reserve ancestors of that name, such as ^D, or descendants, such as ^D(1,2), until LOCK or ZDEALLOCATE releases that name.

Execution of the LOCK command does not affect the value or the state of a variable. LOCK tests each argument to determine whether the process can claim the name space. If another GT.M process has a LOCK on that name space, GT.M suspends the current process until the other process releases the name space. To prevent the potential "indefinite" suspension of a routine execution, specify a timeout for the LOCK command.

LOCK with a leading plus (+) or minus (-) sign (incremental LOCKing) allows the acquisition and release of locks without releasing all currently held locks. This can lead to deadlocks. For example, a deadlock occurs if two users LOCK resources named A and B in the following sequence.

Deadlock Situation	
USER X	USER Y
L +A	L +B

Deadlock Situation	
USER X	**USER Y**
L +B	L +A

To avoid deadlocks, use LOCK without a leading + or - sign on its arguments because such a command releases all previously LOCKed resources, or uniformly implement well designed LOCK accumulation orders and/or use a timeout with the LOCK command.

If a LOCK command specifies a timeout, and GT.M acquires ownership of the named resource before the timeout elapses, LOCK sets $TEST to TRUE (1). If GT.M cannot acquire ownership of the named resource within the specified timeout, LOCK sets $TEST to FALSE (0). If a LOCK command does not specify a timeout, the execution of the command does not affect $TEST. If a LOCK with an argument having a leading minus sign (-) specifies a timeout, the command always sets $TEST to TRUE (1).

If a process issues a LOCK command for a named resource already ZALLOCATEd by that process, the resource is both ZALLOCATEd and LOCKed. LOCK does not release ZALLOCATEd resources. To release such a named resource, the process must both ZDEALLOCATE and unLOCK the resource. For more information, refer to "ZAllocate" (page 170).

Currently, LOCK of an argument within a parenthetical list where the argument includes an extrinsic function that performs LOCK, ZALLOCATE or ZDEALLOCATE actions produces a BADLOCKNEST error except where there is only one such argument, it is the first argument in the list and the LOCK'ng as a consequence of the extrinsic function(s) is simple. Note that this pattern may still produce some unintended outcomes, so FIS recommends against its use.

For more information on troubleshooting locks with the GT.M Lock Utility (LKE), refer to the chapter on that utility in the **GT.M Administration and Operations Guide**.

Using Locks within Transactions

Within transactions, LOCKs are used by GT.M to ensure the ability to serialize. There is no guarantee, however, that attempts by other processes to examine LOCKs held with a transaction will produce the same results as when LOCKs are outside of a transaction. In other words, LOCKs within transactions should never be used as simple semaphores.

The LOCK command locks a specified resource name that controls a tree structured name space. Outside of transactions when one process in an environment acquires a LOCK or a ZALLOCATE on a named resource, no other GT.M process in that environment can LOCK a resource with an "overlapping" name until the first process releases the LOCK that it holds.

For information on the use of LOCKs within transactions, refer to Chapter 5: "General Language Features of M" (page 68).

LOCK Command Operation Summary		
COMMANDS ISSUED	RESULTING LOCKS	COMMENTS
L	none	Remove all prior locks.
L A	A	Remove prior locks then lock A.
L L +A	A	This sequence is equivalent to L A
L A		Remove prior locks before locking A, then remove lock on A. This is equivalent to L A L

LOCK Command Operation Summary		
COMMANDS ISSUED	**RESULTING LOCKS**	**COMMENTS**
L -A	none	
L A L +A L -A	A	Remove prior locks before locking A, increment lock on A without releasing prior lock on A, decrement lock on A without releasing prior lock on A.
L A L +B	A,B	Remove prior locks before locking A, then lock B without releasing A.
L A,B	B	Remove prior locks before locking A, unlock A, then lock B.
L (A,B)	A,B	Remove prior locks before locking A and B simultaneously.
L A L +B L +C	A,B,C	Remove prior locks before locking A, lock B without releasing A, lock C without releasing A and B
L A L +(B,C)	A,B,C	Remove prior locks before locking A, lock B and C simultaneously without releasing A.
L (A,B,C) L -B L -C	A	Remove prior locks before locking A, B, and C simultaneously, remove lock on B without releasing A and C, remove lock on C without releasing A.
L (A,B,C) L -(B,C)	A	Remove prior locks before locking A, B, and C simultaneously, remove lock on B and C without releasing A.
L (A,B) L -B	A	Remove prior locks before locking A and B simultaneously, remove lock on B without releasing A.

Example of LOCK

Example:

```
Lock A,^B,@C
Lock (A,B,@C)
```

The first LOCK command LOCKs A and unLOCKs A before LOCKing ^B, then unLOCKs ^B before locking the name specified by the variable C. The second LOCK command acquires all three resources at once. GT.M waits until all the named resources in the argument list become available before LOCKing all the resources. For example, if the resource specified by the variable C is not available for LOCKing, GT.M waits until that resource becomes available before LOCKing A and ^B.

Example:

```
LOCK (A,B)
```

```
LOCK +C
LOCK -B
```

This LOCKs A and B, then incrementally LOCKs C. Finally it releases the LOCK on B, while retaining the LOCKs on A and C.

Example:

```
LOCK (A,B,C)
LOCK +(B,C)
LOCK -(B)
```

This LOCKs A, B and C together. It then increments the lock "counts" of B and C. The last LOCK command removes one "count" of B, leaving one count of A and B and two counts of C.

Example:

```
LOCK ^D:5
```

This command attempts to LOCK ^D with a timeout of five seconds. If LOCK acquires the named resource before the timeout elapses, GT.M sets $TEST to 1 (TRUE). If LOCK fails to acquire the named resource before the timeout elapses, GT.M sets $TEST to 0 (FALSE).

See Also	• "M Locks" (page 89) • "ZSHOW Information Codes" (page 193) • "ZAllocate" (page 170) • "ZDeallocate" (page 176) • GDE LOCKs (Administration and Operations Guide) • LKE Chapter (Administration and Operations Guide)

Merge

The MERGE command copies a variable and all its descendants into another variable. MERGE does not delete the destination variable, nor any of its descendants.

The format of MERGE command is:

```
M[ERGE][:tvexpr] glvn1=glvn2[,...]
```

- The optional truth-valued expression immediately following the command is a command post conditional that controls whether or not GT.M executes the command.

- When both glvn1 and glvn2 are local variables, the naked indicator does not change.

- If glvn2 is a global variable and glvn1 is a local variable, the naked indicator references glvn2.

- When both are global variables, the state of the naked indicator is unchanged if glvn2 is undefined ($DATA(glvn2)=0).

- In all other cases including $DATA(glvn2)=10, the naked indicator takes the same value that it would have if the SET command replaced the MERGE command and glvn2 had a value.

- If glvn1 is a descendant of glvn2, or if glvn2 is a descendant of glvn1; GT.M generates an error.

- If $data(glvn2) is 0 then the command is a NOOP and GT.M issues no errors.

- An indirection operator and an expression atom evaluating to a list of one or more MERGE arguments form a legal argument for a MERGE.

> **Note**
>
> GT.M may permit certain syntax or actions that are described by the standard as in error. For example, a MERGE command that specifies an operation where the source and destination overlap but $DATA(source)=0 does not produce an error (which is equivalent to a no-operation).

MERGE simplifies the copying of a sub-tree of a local or global variable to another local or global variable. A sub-tree is all global or local variables that are descendants of a specified variable. MERGE offers a one-command alternative to the technique of using a series of SET commands with $ORDER() or $QUERY() references for doing sub-tree copy.

Examples of MERGE

Example:

```
GTM>Set ^gbl1="one"

GTM>Set ^gbl1(1,1)="oneone"
GTM>Set ^gbl1(1,1,3)="oneonethree"
GTM>Set ^gbl1(1,2,4)="onetwofour"
GTM>Set ^gbl2(2)="gbl2_2"
GTM>Set ^gbl2(2,1,3)="gbl2_2_1_3"
GTM>Set ^gbl2(2,1,4,5)="gbl2_2_1_4_5"
GTM>Merge ^gbl1(1)=^gbl2(2)
GTM>WRITE $Reference
^gbl1(1)
GTM>ZWRite ^gbl1
^gbl1="one"
^gbl1(1)="gbl2_2"
^gbl1(1,1)="oneone"
^gbl1(1,1,3)="gbl2_2_1_3"
^gbl1(1,1,4,5)="gbl2_2_1_4_5"
^gbl1(1,2,4)="onetwofour"
GTM>ZWRITE ^gbl2
^gbl2(2)="gbl2_2"
^gbl2(2,1,3)="gbl2_2_1_3"
^gbl2(2,1,4,5)="gbl2_2_1_4_5"
GTM>
```

This example illustrates how MERGE copies a sub-tree of one global into another. The nodes in the sub-tree of ^gbl(2), for which $DATA() value is 1 or 11, are copied to sub-tree of ^gbl1(1) as follows:

```
^gbl1(1) is updated from the value of ^gbl2(2)
^gbl1(1,1,3) is updated from the value of ^gbl2(2,1,3)
^gbl1(1,1,4,5) is updated from the value of ^gbl2(2,1,4,5)
```

Since ^gbl1(2,1) and ^gbl2(2,2,4) do not have values ($DATA()=0), the corresponding nodes ^gbl1(1,1) and ^gbl(1,2,4) respectively are left unchanged. The naked indicator takes the value ^gbl(1) as if SET replaced MERGE. Notice that the MERGE command does not change ^gbl2(2) or its descendants. Ancestor nodes of ^gbl(1) are also left unchanged.

Example:

```
GTM>Kill

GTM>Set ^gbl(1,2)="1,2"
GTM>Merge lcl(3,4)=^gbl(1)
GTM>Set ^("naked")=2
GTM>ZWRite ^gbl
^gbl(1,2)="1,2"
^gbl("naked")=2
GTM>ZWRite lcl
lcl(3,4,2)="1,2"
GTM>
```

This example illustrates how MERGE creates a sub-tree of a variable when the variable does not exist. Also, notice how the naked indicator is set when the source of the MERGE is a global and the destination a local.

New

The NEW command "stacks" copies of local variables and reinitializes those variables. An explicit or implicit QUIT from a DO, XECUTE or extrinsic function "unstacks" the NEWed variables, that is, restores the variable to the stacked value. A NEW lasts only while the current scope of execution is active.

The format of the NEW command is:

```
N[EW][:tvexpr] [[(]lvn[,...][)][,...]]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- NEW arguments are unsubscripted local variable names; NEW affects not only the variable specified in the argument, but also all variables descended from that variable.

- When an undefined variable is NEWed, the fact that it is undefined is "stacked", and when leaving the current scope, it returns to being undefined, that is, the variable is implicitly KILLed during transfer of control.

- Without an argument GT.M NEWs all currently existing local variables; in this case, at least two (2) spaces must follow the NEW to separate it from the next command on the line.

- For the scope of the NEW, a NEW of a name suspends its alias association. The association is restored when the scope of the New ends. The array remains in existence - it can be modified through other alias variables with which it is associated and which remain in scope. If none of its alias variables is in scope, the array remains intact and again becomes visible when the scope is restored.

  When a NEW argument is enclosed in parentheses, that NEW is considered "exclusive". An exclusive NEW creates a fresh data environment and effectively aliases the excluded variables with their original copies. This technique tends to improve performance and meets the M standard. However, it has two implications: The alias operation KILL *, with no arguments, or naming an exclusively NEW'd variable, acts as a KILL in the current scope (has the same effect as a non-alias KILL), and ZWRITE, ZSHOW "V", $ZDATA() report any exclusively NEW'd variable as an alias. Refer to the section on the KILL command for a description of alternative behaviors for the interaction of KILL and exclusive NEW. For a comprehensive discussion on alias variables, refer to "Alias Variables Extensions" (page 11).

- When the flow of execution terminates the scope of an argumentless or an exclusive NEW, GT.M restores all stacked variables to their previous values, and deletes all other local variables.

- The intrinsic special variables $ESTACK, $ETRAP, $ZGBLDIR, and $ZYERROR can be an explicit argument of a NEW.For more information, refer to Chapter 8: "Intrinsic Special Variables" (page 291).

- The intrinsic special variable $ZTRAP can also be an explicit argument of a NEW; this stacks the current value of $ZTRAP and assigns $ZTRAP a null value ($ZTRAP="").

- An indirection operator and an expression atom evaluating to a list of one or more NEW arguments form a legal argument for a NEW.

The NEW command provides a means of confining the scope of local variables. NEW operates only on unsubscripted local names and acts on the entire named array.

Examples of NEW

Example:

```
NEW1;
  Set A(1)=1,B=4,C=5
  Write !,"VARIABLES BEFORE NEW:",!
  ZWRite
  Do LABEL
  Write !,"VARIABLES AFTER RETURN:",!
  ZWRite
  Quit
LABEL
  New A Set C=7
  Write !,"VARIABLES AFTER NEW:",!
  ZWRite
  Quit
```

Produces the results:

```
VARIABLES BEFORE NEW:
A(1)=1
B=4
C=5
VARIABLES AFTER NEW:
B=4
C=7
VARIABLES AFTER RETURN:
A(1)=1
B=4
C=7
```

Example:

```
NEW2;
  Set (A,B,C,D)="TEST"
  Do LABEL
  Write !,"VARIABLES AFTER RETURN:",!
  ZWRite
  Quit
LABEL
  New (B,C) SET (A,B,Z)="NEW"
```

```
  Write !,"VARIABLES AFTER EXCLUSIVE NEW:",!
  ZWRite
  Quit
```

Produces the results:

```
VARIABLES AFTER EXCLUSIVE NEW:
A="NEW"
B="NEW"
C="TEST"
Z="NEW"
VARIABLES AFTER RETURN:
A="TEST"
B="NEW"
C="TEST"
D="TEST"
```

Example:

```
/usr/lib/fis-gtm/V5.4-002B_x86/gtm -run ^stackalias
stackalias ; Demonstrate New with alias
  ZPrint ; Print this program
  Set A=1,*B=A,*C(2)=A ; Create some aliases
  Write "------------",!
  Write "ZWRite in the caller before subprogram",!
  ZWRite
  Do S1 ; Call a subprogram
  Write "------------",!
  Write "ZWRite in the caller after subprogram - A association is restored",!
  ZWRite
  Quit
 ;
S1  ; Subprogram
  New A
  Set A="I am not an alias",B="I am an alias"
  Write "------------",!
  Write "ZWRite in the subprogram with new A and modified B",!
  ZWRite
  Quit
------------
ZWRite in the caller before subprogram
A=1 ;*
*B=A
C=3
*C(2)=A
D=4
------------
ZWRite in the subprogram with new A and modified B
A="I am not an alias"
B="I am an alias" ;*
C=3
*C(2)=B
D=4
------------
ZWRite in the caller after subprogram - A association is restored
A="I am an alias" ;*
```

```
*B=A
C=3
*C(2)=A
D=4
```

The following is essentially the same as the prior example but using an exclusive NEW:

```
$ /usr/lib/fis-gtm/V5.4-002B_x86/gtm -run ^stackalias1
stackalias1 ; Demonstrate New with alias
  ZPrint ; Print this program
  Set A=1,*B=A,*C(2)=A ; Create some aliases
  Write "------------",!
  Write "ZWRite in the caller before subprogram",!
  ZWRite
  Do S1 ; Call a subprogram
  Write "------------",!
  Write "ZWRite in the caller after subprogram - A association is restored",!
  ZWRite
  Quit
 ;
S1  ; Subprogram
  New (B)
  Set A="I am not an alias",B="I am an alias"
  Write "------------",!
  Write "ZWRite in the subprogram - Notice B is flagged as an alias",!
  ZWRite
  Quit
------------
ZWRite in the caller before subprogram
A=1 ;*
*B=A
C=3
*C(2)=A
D=4
------------
ZWRite in the subprogram - Notice B is flagged as an alias
A="I am not an alias"
B="I am an alias" ;*
------------
ZWRite in the caller after subprogram - A association is restored
A="I am an alias" ;*
*B=A
C=3
*C(2)=A
D=4
```

An exclusive New can create a scope in which only one association between a name or an lvn and an array may be visible. In this case, ZWRITE nevertheless shows the existence of an alias, even when that array is accessible from only one name or lvn.

See Also	"Alias Variables Extensions" (page 11) "$EStack" (page 295) "$ETrap" (page 295) "$ZGbldir" (page 307) "$ZYERror" (page 333)

Open

The OPEN command creates a connection between a GT.M process and a device.

The format of the OPEN command is:

```
O[PEN][:tvexpr] expr[:[(keyword[=expr][:...])] [:numexpr]][,...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The required expression specifies the device to OPEN.

- The optional keywords specify deviceparameters that control device behavior; some deviceparameters take arguments delimited by an equal sign (=); if the argument only contains one deviceparameter, the surrounding parentheses are optional.

- The optional numeric expression specifies a time in seconds after which the command should timeout if unsuccessful; choosing 0 results in a single attempt to open the device.

- When an OPEN command specifying a timeout contains no deviceparameters, double colons (::) separate the timeout numeric expression from the device expression.

- An indirection operator and an expression atom evaluating to a list of one or more OPEN arguments form a legal argument for an OPEN.

- In UTF-8 mode, the OPEN command recognizes the ICHSET, OCHSET, and CHSET deviceparameters to determine the encoding of the input / output devices.

- OPEN on a directory produces a GTMEISDIR error in both READONLY and NOREADONLY modes along with the directory name which failed to open. UNIX directories contain metadata that is only available to the file system. Note that you can use the ZSEARCH() function to identify files in a directory, and you can call the POSIX stat() function to access metadata. The optional GT.M POSIX plug-in packages the stat() function for easy access from M application code.

See Also

- "Open" (page 391)
- "Deviceparameter Summary Table" (page 444)

Quit

Except when a QUIT appears on a line after a FOR, the QUIT command terminates execution of the current GT.M invocation stack level initiated by a DO, XECUTE, extrinsic function or special variable, and return control to the next "lower" level. In this case, QUIT restores any values stacked at the current level by NEWs or by parameter passing. A QUIT command terminates any closest FOR command on the same line. Note that M overloads the QUIT command to terminate DO, FOR, XECUTE and extrinsics ($$) of which FOR is the most different.

The format of the QUIT command is:

```
Q[UIT][:tvexpr] [expr | *lname | *lvn]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- When a QUIT terminates an extrinsic function, it must have an argument that supplies the value returned by the function; in all other cases, QUIT must not have an argument and must be followed by at least two (2) spaces to separate it from the next command on the line.

- An indirection operator and an expression atom evaluating to a QUIT argument form a legal argument for a QUIT.

- An unsubscripted lvn (lname) specifies the root of an array, while a subscripted lvn must specify an alias container.

- When QUIT * terminates an extrinsic function or an extrinsic special variable, it always returns an alias container. If lvn is an lname that is not an alias, QUIT * creates an alias container. For more information and examples of alias variables, refer to "Alias Variables Extensions" (page 11).

- The QUIT performs two similar, but different, functions depending on its context. Because FORs do not add levels to the GT.M invocation stack, QUITs inside FOR loops simply terminate the loop. QUITs that terminate DOs, XECUTEs and extrinsics remove a GT.M invocation stack level and therefore may adjust the local variable environment resulting from previous NEWs or parameter passing. A QUIT from an extrinsic or a frame created by an argumentless DO restores $TEST to its stacked value.

- An indirection operator and an expression atom evaluating QUIT arguments forms a legal argument for a QUIT other than from a FOR.

- Attempting to QUIT (implicitly or explicitly) from code invoked by a DO, XECUTE or extrinsic after that code issued a TSTART not yet matched by a TCOMMIT, produces an error.

Examples of QUIT

Example:

```
     Do A
     Quit
A    Write !,"This is label A"
```

The explicit QUIT at the line preceding the label A prevents line A from executing twice. The sub-routine at line A terminates with the implicit QUIT at the end of the routine.

Example:

```
       Write $$ESV
        Quit
ESV()
       QUIT "value of this Extrinsic Special Variable"
```

Because the label ESV has an argument list (which is empty), GT.M can only legally reach that label with an extrinsic invocation. The QUIT on the second line prevents execution from erroneously "falling through" to the line labeled ESV. Because ESV identifies a subroutine that implements an extrinsic special variable, the QUIT on the line after ESV has an argument to provide the value of the extrinsic.

Example:

```
Set x="" For  Set x=$Order(^BAL(x)) Quit:x]]"AR5999"!'$Length(x)  DO STF
```

The postconditional QUIT terminates the FOR loop. Note the two spaces after the QUIT because it has no argument.

Read

The READ command transfers the input from the current device to a global or local variable specified as a READ argument. For convenience, READ also accepts arguments that perform limited output to the current device.

The format of the READ command is:

```
R[EAD][:tvexpr] (glvn|*glvn|glvn#intexpr)[:numexpr]|strlit|fcc[,...]
```

* The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

* A subscripted or unsubscripted global or local variable name specifies a variable into which to store the input; the variable does not have to exist prior to the READ; if the variable does exist prior to the READ, the READ replaces its old value.

* When an asterisk (*) immediately precedes the variable name, READ accepts one character of input and places the ASCII code for that character into the variable.

* When a number-sign (#) and a positive non-zero integer expression immediately follow the variable name, the integer expression determines the maximum number of characters accepted as input to the read; such reads terminate when GT.M reads the number of characters specified by the integer expression or a terminator character in the input stream or the optional timeout expires, whichever occurs first.

* The optional numeric expression specifies a time in seconds at most, for which the command waits for input to be terminated. When a timeout is specified, if the input has been terminated before the timeout expires, $TEST is set to 1 (true), otherwise, $TEST is set to 0 (false). When a READ times out, the target variable takes the value of the string received before the timeout.

* To provide a concise means of issuing prompts, GT.M sends string literal and format control character (!,?intexpr,#) arguments of a READ to the current device as if they were arguments of a WRITE.

* An indirection operator and an expression atom evaluating to a list of one or more READ arguments form a legal argument for a READ.

* In UTF-8 mode, the READ command uses the character set value specified on the device OPEN as the character encoding of the input device. If character set "M" or "UTF-8" is specified, the data is read with no transformation. If character set is "UTF-16", "UTF-16LE", or "UTF-16BE", the data is read with the specified encoding and transformed to UTF-8. If the READ command encounters an illegal character or a character outside the selected representation, it generates a run-time error. The READ command recognizes all Unicode® line terminators for non-FIXED devices.

For more information on READ, devices, input, output and format control characters, refer to Chapter 9: "Input/Output Processing" (page 340).

See Also	• "READ" (page 437) • "READ" (page 437) • "READ * Command for Terminals" (page 355) • "READ X#maxlen Command for Terminals" (page 355)

Set

SET assigns values to variables or to a selected portion of a variable.

The format of the SET command is:

```
S[ET][:tvexpr] setleft=expr | (setleft[,...])=expr | *lvn=lname | aliascontainer[,...]
```

where

```
setleft == glvn | $EXTRACT(glvn,[,intexpr1[,intexpr2]]) | $PIECE(glvn,expr1[,intexpr1[,intexpr2]]) | isv
```

and

```
aliascontainer == lvn | exfunc | exvar
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- A subscripted or unsubscripted local or global variable name on the left of the equal-sign (=) specifies a variable in which to store the expression found on the right side of the equal-sign; the variable need not exist prior to the SET; if the variable exists prior to the SET, the SET replaces its old value.

- During a SET, GT.M evaluates the right side of the equal sign before the left; this is an exception to the left-to-right order of evaluation in GT.M and means that GT.M maintains the naked indicator using the expression on the right-hand side of the equal sign (=) before setting the variable.

- When the portion of the argument to the left of the equal sign is in the form of a $PIECE function, SET replaces the specified piece or pieces of the variable (specified as the first argument to the $PIECE() form) with the value of the expression on the right side of the equal-sign; if the variable did not exist prior to the SET or does not currently contain the pieces identified by the optional third and fourth arguments to the $PIECE() form, SET adds sufficient leading delimiters, as specified by the second argument to the $PIECE form, to make the assignment fit the $PIECE() form. Note that if the fourth argument exceeds the third argument, SET does not modify the target glvn or change the naked indicator.

- When the portion of the argument to the left of the equal sign is in the form of a $EXTRACT function, SET replaces the specified character or characters of the variable (specified as the first argument to the $EXTRACT() form) with the value of the expression on the right side of the equal-sign; if the variable did not exist prior to the SET or does not contain the characters identified by the optional second and third arguments to the $EXTRACT() form, SET adds sufficient leading spaces to make the assignment fit the $EXTRACT() form. Note that if the third argument exceeds the second argument, SET does not modify the target glvn or change the naked indicator .

- **isv** on the left-hand side of the equal-sign specifies an Intrinsic Special Variable. Not all ISVs permit SET updates by the application - see the description of the individual ISV.

- When the portion of the argument to the left of the equal-sign is in the form of a list of **setlefts** enclosed in parentheses, SET assigns the value of the expression on the right of the equal sign to all the destinations.

- If a SET updates a global node matching a trigger definition, GT.M executes the trigger code after the node has been updated in the process address space, but before it is applied to the database. When the trigger execution completes, the trigger logic commits the value of a node from the process address space only if $ZTVALUE is not set. if $ZTVALUE is set during trigger execution, the trigger logic commits the value of a node from the value of $ZTVALUE. For more information on using SET in Triggers, refer to "Set" (page 601) section in the Triggers chapter.

- A SET * command explicitly makes the lvn on the left-hand side of the equal-sign an alias if it is an unsubscripted lvn (the root of an array) or an alias container if it is a subscripted lvn. If the portion of the argument on the right-hand side of the equal-sign is other than an lname (the root of an array), it must evaluate to an alias or alias container. Extrinsic functions and extrinsic special variables return an alias container if they terminate with a QUIT *. For more information on Alias Variables, refer to "Alias Variables Extensions" (page 11).

- In a SET * command, any previous array associated with the lvn on the left-hand side of the equal-sign ceases to be associated with it, and if lvn was the only lvn associated with that (old) array in any scope, GT.M may reclaim the space it

occupied. Alias assignment does not require that any data set exist for a name on the right-hand side of the equal-sign - the assignment simply creates an association.

- SET * left-hand side arguments cannot be parenthetically enclosed lists such as SET (a,*b)=c or SET (*a,*b)=c.

- SET and SET * assignments can be combined into one command in a comma separated list, for example, SET *a=b,^c(3)=d(4).

- SET * only accepts argument indirection, that is, while SET accepts x="*a=b",@x, SET does not permit x="*a",@x=b or SET x="b",*a=@x.

- An indirection operator and an expression atom evaluating to a list of one or more SET arguments form a legal argument for a SET.

- A SET with proper syntax always succeeds regardless of the prior state or value of the variable, as long as GT.M can evaluate the expression to the right of the equal sign (=).

For the syntax of $PIECE() or $EXTRACT(), refer to Chapter 7: "Functions" (page 212).

Examples of SET

Example:

```
GTM>Kill  Set a="x",(b,c)=1,@a="hello" ZWRite
a=x
b=1
c=1
x="hello"
GTM>
```

The KILL command deletes any previously defined local variables. The SET command has three arguments. The first shows a simple direct assignment. The second shows the form that assigns the same value to multiple variables. The third shows atomic indirection on the left of the equal sign. The ZWRITE command displays the results of the assignments.

Example:

```
GTM>Set ^(3,4)=^X(1,2)
```

As GT.M evaluates the right-hand side of the equal sign before the left-hand side within a SET argument, the right-hand expression determines the naked reference indicator prior to evaluation of the left-hand side. Therefore, this example assigns ^X(1,3,4) the value of ^X(1,2).

Example:

```
GTM>Kill x Set $Piece(x,"^",2)="piece 3" ZWRite x
x="^^piece 3"
GTM>
```

This SET demonstrates a "set piece" and shows how SET generates missing delimiters when required. For more information on $PIECE(), refer to Chapter 7: "Functions" (page 212).

Example:

```
GTM>Set x="I love hotdogs"
```

```
GTM>Set $Extract(x,3,6)="want"
GTM>Write x
I want hotdogs
GTM>Set $Extract(x,7)=" many "
GTM>Write x
I want many hotdogs
GTM>
```

The SET $EXTRACT command replaces and extracts the specified characters with the value of the expression on the right hand side of the equal-sign (=). For more information on $EXTRACT(), refer to Chapter 7: "Functions" (page 212).

Example:

```
GTM>kill A,B

GTM>set A=1,A(1)=1,A(2)=2
GTM>set *B=A ; A & B are aliases.
GTM>zwrite B
B=1 ;*
B(1)=1
B(2)=2
GTM>
```

This SET * command creates an alias associated between A and B. It associates the entire tree of nodes of A including its root and all descendants with B.

Example:

```
GTM>kill A,B,C

GTM>set A=1,*C(2)=A ; C(2) is a container
GTM>zwrite
A=1 ;*
*C(2)=A
GTM>set *B=C(2) ; B is now an alias
GTM>write B,":",$length(C(2)),":" ; An alias variable provides access but a container doesn't
1:0:
GTM>
```

This SET * command creates an alias by dereferencing an alias container.

See Also

- "Set" (page 601)
- "$Extract()" (page 218)

TCommit

The TCOMMIT command marks the end of a transaction or sub-transaction and decrements $TLEVEL. If TCOMMIT marks the end of a transaction (decrements $TLEVEL to zero), it invokes a COMMIT, which makes the database updates performed by the transaction generally available. A TCOMMIT issued when no transaction is in progress ($TLEVEL=0) produces an error.

The format of the TCOMMIT command is:

```
TC[OMMIT][:tvexpr]
```

- The optional truth-valued expression immediately following the command is a command postcondition that controls whether or not GT.M executes the command.

- Because TCOMMIT has no argument, at least two (2) spaces must follow the command to separate it from the next command on the line.

For an example of the use of the TCOMMIT command, see Chapter 5: "General Language Features of M" (page 68).

TREstart

The TRESTART command attempts to RESTART the current transaction. A RESTART transfers control back to the initial TSTART and restores much of the process state to what it was when that TSTART was originally executed. A TRESTART issued when no transaction is in progress ($TLEVEL=0) or when the transaction does not have RESTART enabled produces an error.

A TRESTART command causes the TP transaction to RESTART in the same way that GT.M uses to implicitly restart the transaction in case of resource conflicts. All restarts increment the internal transaction retry count to a maximum of three (3), at which point, GT.M performs the entire TP transaction within a critical section on all databases referenced in the transaction.

GT.M issues a TRESTMAX runtime error when application code attempts a TRESTART more than once during a transaction while $TRESTART=4 (note: in order to be wholesome, TRESTART usage in application code should always be conditional). In the final retry, GT.M holds the critical section lock on all databases involved in the transaction. Since a TRESTART cancels all the work done in the current transaction and transfers control back to the TSTART, limiting the number of times this can be done in the final retry limits the time a process can (by virtue of holding a critical section lock on the databases) prevent other processes from updating the database.

GT.M limits TP restarts in the final retry due to non-availability of M-locks in a similar fashion. GT.M allows a maximum of 16 such restarts after which it issues a TPLOCKRESTMAX runtime error.

The format for the TRESTART command is:

```
TRE[START][:tvexpr]
```

- The optional truth-valued expression immediately following the command is a command postcondition that controls whether or not GT.M executes the command.

- Because TRESTART has no argument, at least two (2) spaces must follow the command to separate it from the next command on the line.

    TRESTARTs (and implicit RESTARTs) do not restore any device states; they do restore the following to the state they had when GT.M executed the initial TSTART:

    - $TEST

    - All global variables modified by the current base transaction and any of its sub-transactions

    - The naked indicator

    - LOCKs held by the process

- A TP RESTART, either implicit or explicit, while executing $ZINTERRUPT in response to an interrupt (that is, $ZININTERRUPT is 1), and while error processing is in effect (that is, $ECODE'=""), raises a TPRESTNESTERR error and engages nested error handling, which unstacks M virtual machine frames back to where the incompletely handled error occurred, unstacks that frame and rethrows the error.

They also restore any local variables named by one or more active TSTARTs to the values they had when they were first named.

For an example of the use of the TRESTART command, see Chapter 5: "General Language Features of M" (page 68).

TROllback

The TROLLBACK command terminates a transaction by causing a ROLLBACK, which removes all database updates performed within a transaction. A TROLLBACK without an argument also sets $TLEVEL and $TRESTART to zero (0). Issuing a TROLLBACK when no transaction is in progress ($TLEVEL=0) produces an error.

The format of the TROLLBACK command is:

```
TRO[LLBACK][:tvexpr] [intexpr]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The optional integer expression indicates an argument specifying incremental rollback. If the value of the argument expression is greater than zero, it specifies the value of $TLEVEL to be achieved by the rollback. If the value of the expression is less than zero, the result is the number of levels to rollback. For example; -1 means rollback one level. If the argument expression is zero, the effect is same as not specifying the argument, that is, the entire GT.M transaction is rolled back.

- Attempting to rollback more than $TLEVEL levels (the outermost transaction) generates an error.

- When the TROLLBACK has no argument, at least two (2) spaces must follow the command to separate it from the next command on the line.

  In order to allow for error recovery and/or access to the global context of the error, errors do not initiate implicit ROLLBACKs. Therefore, the code for handling errors during transactions should generally include a TROLLBACK. Because the TROLLBACK releases resources held by the transaction, it should appear as early as possible in the error handling code.

- A TROLLBACK does not cause a transfer of control but is typically associated with one such as a QUIT (or GOTO).

- TROLLBACK to a $TLEVEL other than zero (0) leaves $REFERENCE empty. This behavior is same as a full TROLLBACK to $TEVEL=0.

For an example of the use of the TROLLBACK command, see Chapter 5: "General Language Features of M" (page 68).

TStart

The TSTART command marks the beginning of a transaction or sub-transaction and increments $TLEVEL. When TSTART marks the beginning of a transaction ($TLEVEL=1), its arguments determine whether the transaction may RESTART and whether serializability is enforced. If a transaction may RESTART, the TSTART arguments determine which local variables are restored during a RESTART. Serializability is enforced by LOCK commands or, if the SERIAL keyword is specified, by GT.M.

The format of the TSTART command is:

```
TS[TART][:tvexpr] [([lvn...])|lvn|*|][:keyword|(keyword...)]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- If $TLEVEL is 0 before the TSTART, the TSTART starts a transaction; otherwise it starts a sub-transaction.

- If the TSTART initiates a transaction and the portion of the argument before the colon (:) delimiter is empty, the transaction is not eligible for RESTART. If the TSTART starts a transaction ($TLEVEL=0) and the portion of the argument before the colon is not empty, the transaction is eligible for RESTART. If the TSTART is nested (starts a sub-transaction), its arguments have no effect on whether the transaction is eligible for RESTART.

- If the portion of the argument before the colon is an asterisk (*), any subsequent RESTART restores all local variables to the value they had when the TSTART was executed.

- If the portion of the argument before the colon is an unsubscripted local variable name or a list of such names enclosed in parentheses, a RESTART restores the named variables to the value they had when the TSTART was executed.

- If the portion of the argument before the colon is a set of empty parentheses (), a RESTART does not restore any local variables.

- The optional portion of the argument after the colon is a keyword or a colon-separated list of keywords enclosed in parentheses, where the keywords specify transaction characteristics.

- An indirection operator and an expression atom evaluating to a TSTART argument form a legal argument for a TSTART.

- Using TSTART in direct mode may not behave as expected because there is no code repository to support an appropriate transaction restart.

A TSTART within a transaction starts a sub-transaction. The argument to such a TSTART has no effect on whether the existing transaction may RESTART or whether serializability of the transaction is enforced. This type of TSTART may add local variables to be restored in a transaction that has RESTART enabled.

It is good coding practice to synchronize enabling of RESTART on TSTARTs at all levels of a transaction. A nested TSTART that does not permit RESTART where the transaction does, may indicate that the sub-transaction has not been coded to properly handle RESTART.

Sub-transactions cannot COMMIT independently from the transaction, nor can they RESTART independently. Sub-transactions exist largely as a programming convenience to allow flexibility in organizing code in a modular fashion, and in addition to allow incremental ROLLBACKs.

When journaling, a transaction with an initial TSTART that has an argument specifying TRANSACTIONID=expr, where expr is an expression that evaluates to the keyword (case insensitive) BA[TCH], does not wait for the journal update to be written before returning control to the application after a successful TCOMMIT. The goal of this feature is to permit application control over any performance impact of journaling on any subset of transactions that can be recreated or recovered by means other than journaling.

For an example of the TSTART command, refer to Chapter 5: "General Language Features of M" (page 68).

The following keywords may appear in a TSTART argument:

S[ERIAL]

The SERIAL keyword indicates that GT.M must ensure the serializability of the transaction. Note that GT.M always serializes transactions regardless of the SERIAL keyword. On a nested TSTART, this portion of the argument is irrelevant.

T[RANSACTIONID]=expr

The TRANSACTIONID keyword declares an arbitrary transaction identification.

If TRANSACTIONID="BATCH" or "BA" at transaction completion, the process immediately continues execution. When a process issues a [final] TCOMMIT for a transaction and journaling is active, by default the process waits until the entire transaction is written to the journal file(s) before executing the next command. This ensures that every transaction is durable before the process moves on to the next step. Transactions flagged as "BATCH" have lower latency and higher throughput, but a lower guarantee of durability. Normally this flag is used when operational procedures (such as a backup) or application code (such as a checkpoint algorithm) provides an acceptable alternative means of ensuring durability.

Use

The USE command selects the current device for READs (input) and WRITEs (output).

The format of the USE command is:

```
U[SE][:tvexpr] expr[:(keyword[=expr][:...])][,...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The required expression specifies the device to make the current device.

- A USE that selects a device not currently OPENed by the process causes a run-time error.

- The optional keywords specify deviceparameters that control device behavior; some deviceparameters take arguments delimited by an equal sign (=); if the argument only contains one deviceparameter, the surrounding parentheses are optional.

- An indirection operator and an expression atom evaluating to a list of one or more USE arguments form a legal argument for a USE.

See Also

- "Use" (page 417)
- "Deviceparameter Summary Table" (page 444)

View

The VIEW command adjusts an environmental factor selected by a keyword argument. For example, VIEW controls journal buffer flushing, determines whether GT.M reports undefined variables as errors or treats them as null, and determines which BREAK commands should display messages.

The format of the VIEW command is:

```
V[IEW][:tvexpr] keyword[:expr2[:...]][,...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The keyword specifies the environmental factor to change.

- The optional expression following the keyword specifies the nature of the change to the environmental factor. When this expression is a region list (a comma delimited list of regions), GT.M sorts the regions in an internal order eliminating any duplicates from the list for deadlock prevention. When region list is not specified, VIEW operates on all regions under the current global directory.

- An indirection operator and an expression atom evaluating to a list of one or more VIEW arguments form a legal argument for a VIEW

Key Words in VIEW Command

The following sections describe the keywords available for the VIEW command in GT.M.

"BREAKMSG":value

Sets the value of the BREAK message mask. When GT.M processes a BREAK command, the BREAK message mask controls whether to display a message describing the source of the BREAK.

The mask uses the following four values that are added together to provide the BREAKMSG value.

1 - BREAKs within the body of a program

2 - BREAKs within a ZBREAK action

4 - BREAKs within a device EXCEPTION

8 - BREAKs within a ZSTEP action

16 - ZBREAKs within a trigger removed due to updated trigger (TRIGZBREAKREM)

The default BREAKMSG mask is 31 (1+2+4+8+16) which means that GT.M displays all BREAK messages.

Example:

```
GTM>VIEW "BREAKMSG":5
```

In this example the BREAKMSG value is 5, representing the sum of 1 and 4. This enables BREAKS within the body of a program (value 1) and for a device EXCEPTION (value 4).

[NO]BADCHAR

Enables or disable the gneration of an error when character-oriented functions encounter malformed byte sequences (illegal characters).

At process startup, GT.M initializes BADCHAR from the environment variable gtm_badchar. Set the environment variable $gtm_badchar to a non-zero number or "YES" (or "Y") to enable VIEW "BADCHAR". Set the environment variable $gtm_badchar to 0 or "NO" or "FALSE" (or "N" or "F") to enable VIEW "NOBADCHAR". By default, GT.M enables VIEW "BADCHAR".

With VIEW "BADCHAR", GT.M functions generate the BADCHAR error when they encounter malformed byte sequences. With this setting, GT.M detects and clearly reports potential application program logic errors as soon as they appear. As an illegal UTF-8 character in the argument of a character-oriented function likely indicates a logic issue, FIS recommends using VIEW "BADCHAR" in production environments.

> **Note**
>
> When all strings consist of well-formed characters, the value of VIEW [NO]BADCHAR has no effect whatsoever. With VIEW "NOBADCHAR", the same functions treat malformed byte sequences as valid

characters. During the migration of an application to add support for the UTF-8 mode, illegal character errors are likely to be frequent and indicative of application code that is yet to be modified. VIEW "NOBADCHAR" suppresses these errors at times when their presence impedes development.

"DBFLUSH"[:<region_list>[:N]]

When using the BG access method, writes modified blocks in the global buffers to the database file. By default, this command option operates on all regions under the current global directory. N specifies the number of blocks to write; by default, DBFLUSH writes all modified blocks. Normally GT.M schedules block flushing at appropriate times, but this option exists for an application to explore the impact of flushing on their work load. See also the DBSYNC and EPOCH VIEW Options.

"DBSYNC"[:<region_list>]

Performs a file system hardening sync - fsync() - operation on the database file. By default, this command option operates on all regions under the current global directory. Normally GT.M schedules block flushing at appropriate times, but this option exists for an application to explore the impact of file hardening on their work load. See also the DBFLUSH and EPOCH VIEW Options.

[NO]DMTERM

Provides a mechanism to retain default line terminators for direct mode user interaction (including the BREAK command) independent of any TERMINATOR deviceparameter changes for $PRINCIPAL. With VIEW "NODMTERM", TERMINATOR deviceparameter apply to both READs from $PRINCIPAL and direct mode interactions. A case-insensitive value of the environment variable gtm_dmterm is "1", "yes", or "true" establishes a DMTERM state at process initiation; all other values, including no value, result in the default VIEW "NODMTERM" behavior. $VIEW("DMTERM") returns 1 for DMTERM mode or 0 for NODMTERM mode.

"EPOCH"[:<region_list>]

Flushes the database buffers and, if journaling is enabled, writes an EPOCH record. By default, this command option operates on all regions under the current global directory. Normally GT.M schedules epochs as a user controlled journaling characteristic, but this option exists for an application to explore the impact of epochs on their work load. See also the DBFLUSH and DBSYNC VIEW Options. Epochs include DBFLUSH and DBSYNC actions, but performing them before the epoch may reduce the duration of these actions within the epoch.

"FLUSH"[:<region_list>]

Flushes dirty global buffers from the global buffer pool. If journaling is turned on, "FLUSH" writes an EPOCH record and flushes dirty journal buffers prior to flushing dirty global buffers. If no region is specified, VIEW "FLUSH" flushes all regions in the current global directory that the GT.M process has opened.

[NO]FULL_BOOL[EAN|WARN]

Controls the evaluation of Boolean expressions (expressions evaluated as a logical TRUE or FALSE).

By default, GT.M enables VIEW "NOFULL_BOOLEAN" which means that GT.M stops evaluating a Boolean expression as soon as it establishes a definitive result. For example, neither 0&$$abc^def() nor 1!$$abc^def() executes $$abc^def(). However, in the case of global references, such as 0&^a or 1!^a, GT.M sets $reference and the naked indicator without actually accessing the global variable.

With VIEW "FULL_BOOLEAN", GT.M ensures that all side effect expression atoms, extrinsic functions ($$), external functions ($&), and $INCREMENT() execute in left-to-right order.

With VIEW "FULL_BOOLWARN", GT.M not only evaluates Boolean expressions like "FULL_BOOLEAN" but produces a BOOLSIDEFFECT warning when it encounters Boolean expressions that may induce side-effects; that is: expressions with side effects after the first Boolean operator - extrinsic functions, external calls and $INCREMENT().

GT.M picks up the value of [NO]FULL_BOOL[EAN|WARN] from the environment variable gtm_boolean. If gtm_boolean is undefined or evaluates to an integer zero (0), the initial setting the default "NOFULL_BOOLEAN", if it evaluates to an integer one (1), the initial setting is "FULL_BOOLEAN" and if it evaluates to integer two (2) the initial setting is "FULL_BOOLWARN".

VIEW "[NO]FULL_BOOL[EAN][WARN]" takes effect immediately for indirection and XECUTE.

VIEW "NOFULLBOOLEAN" produces an error when gtm_side_effects is on. For more information on the gtm_side_effects environment variable, refer to the Environment Variables section in the Basic Operations chapter of the Administration and Operations Guide.

"GDSCERT":value

Enables (value=1) or disables (value=0) database block certification.

Database block certification causes GT.M to check the internal integrity of every block as it writes the block. Block certification degrades performance and exists primarily as a tool for use by FIS. The default is GDSCERT:0.

"GVSRESET":"<region>"

Resets the process-specific fields that are part of the ZSHOW "G" result and database file header fields holding records reported by: GVSTAT, BG trace, buffer pool accounting and the TP block modification details. Note a VIEW "GVSRESET" performed by a process with read-only database access changes only the process-specific information and has no effect on the database file header. DSE CHANGE -FILEHEADER -GVSTATSRESET clears the same database file header fields as VIEW "GVRESET";

"GVDUPSETNOOP":value

Enables (VIEW "GVDUPSETNOOP":1) or disables (VIEW "GVDUPSETNOOP":0) duplication set optimization.

Duplicate set optimization prevents a SET that does not change the value of an existing node from performing the update or executing any trigger code specified for the node. By default, duplicate set optimization is enabled.

"JNLFLUSH"[:<region_list>]

Writes or flushes journaling buffers associated with the given region to permanent storage, for example, to disk. If the VIEW "JNLFLUSH" does not specify the optional region, GT.M flushes all journaled regions of the current Global Directory.

Normally GT.M writes journal buffers when it completes a transaction (unless TRANSACTIONID="BATCH"), fills the journal buffer or when some period of time passes with no journal activity.

JNLWAIT

Causes a process to pause until its journaling buffers have been written. JNLWAIT ensures that GT.M successfully transfers all database updates issued by the process to the journal file before the process continues. Normally, GT.M performs journal buffer

writes synchronously for TP updates, and asynchronously, while the process continues execution, for non-TP updates or TP updates with TRANSACTIONID=BATCH.

JNLWAIT operates only on those regions for which the current process has opened journal files. As all the journal activity for a TP transaction occurs at commit time, GT.M ignores JNLWAIT when inside a TP TRANSACTION ($TLEVEL > 0). For more information on journaling, refer to the "GT.M Journaling" chapter in the **GT.M Administration and Operations Guide**.

"JOBPID":"value"

Enables (value=1) or disables (value=0) the addition of the child process ID to the output and error file names used (either implicitly generated or explicitly defined) by the JOB command. The default is 0.

Using the value=1 option prevents the JOB command from overwriting output files each time the same JOB command executes.

"LABELS":"value"

Enables (value="LOWER") or disables (value="UPPER") case sensitivity for labels within routines.

It is important to have the same case handling at compile-time and run-time.

Because GT.M stores routines as regular files and file names are case sensitive on UNIX, GT.M always treats routine names as case sensitive.

"LINK":"[NO]RECURSIVE"

Enables ("LINK":"RECURSIVE") or disables ("LINK":"RECURSIVE") the ZLINK command to accept and relink routines on the GT.M invocation stack. With VIEW "LINK":"RECURSIVE" specified, the ZLINK command adds an executable routine even when a routine with the same name is active and available in the current stack. When a process links a routine with the same name as an existing routine, future calls use the new routine. Prior versions of that routine referenced by the stack remain tied to the stack until they QUIT, at which point they become inaccessible. This provides a mechanism to patch long-running processes.

The default is VIEW "LINK":"NORECURSIVE".

[NO]LOGN[ONTP][:intexpr]

Allows a process to dynamically change the logging of NONTPRESTART messages to the operator log established at process startup by the environment variables gtm_nontprestart_log_delta and gtm_nontprestart_log_first.

VIEW "NOLOGNONTP" turns off the logging of NONTPRESTART messages to the operator log.

VIEW "LOGNONTP"[:intexpr] turns on logging of NONTPRESTART messages to the operator log. If no intexpr is specified, GT.M uses the value of environment variable gtm_nontprestart_log_delta, if it is defined, and one otherwise (that is, it logs every transaction restart) A zero (0) or negative value of intexpr turns off the logging of NONTPRESTART messages.

Note that it is not possible to perform the operations of gtm_nontprestart_log_first with VIEW "LOGNONTP"[:intexpr].

[NO]LOGT[PRESTART][:intexpr]

Allows a process to dynamically change the logging of TPRESTART messages to the operator log established at process startup by the environment variables gtm_tprestart_log_delta and gtm_tprestart_log_first.

VIEW "NOLOGTPRESTART" turns off the logging of TPRESTART messages to the operator log.

VIEW "LOGTPRESTART"[:intexpr] turns on logging of TPRESTART messages to the operator log. If no intexpr is specified, GT.M uses the value of environment variable gtm_tprestart_log_delta, if it is defined, and one otherwise (that is, it logs every transaction restart). A zero (0) or negative value of intexpr turns off the logging of TPRESTART messages.

Note that it is not possible to perform the operations of gtm_tprestart_log_first with VIEW "LOGTPRESTART"[:intexpr].

LV_GCOL

Starts a data-space garbage collection, which normally happens automatically at appropriate times.

> **Note**
>
> There are no visible effects from LV_GCOL, LV_REHASH, and STP_GCOL except for the passage of time depending on the state of your process. FIS uses these VIEW "LV_GCOL","LV_REHASH","STP_GCOL" facilities in testing. They are documented to ensure completeness in product documentation. You may (or may not) find them useful during application development for debugging or performance testing implementation alternatives.

LV_REHASH

Starts a reorganization of the local variable look-up table, which normally happens automatically at appropriate times.

> **Note**
>
> There are no visible effects from LV_REHASH, LV_GCOL, and STP_GCOL except for the passage of time depending on the state of your process. FIS uses these VIEW "LV_GCOL","LV_REHASH","STP_GCOL" facilities in testing. They are documented to ensure completeness in product documentation. You may (or may not) find them useful during application development for debugging or performance testing implementation alternatives.

[NEVER]|[NO]LVNULLSUBS

Disallows, partially disallows, or allows local arrays to have empty string subscripts. The default is LVNULLSUBS.

NOLVNULLSUBS disallows any variant of SET to operate on a local array having an empty string subscript.

NEVERLVNULLSUBS disallows any variant of SET or KILL ($DATA(),$GET(),$ORDER(), and $QUERY()) to operate on a local array having an empty string subscript. An empty string as the last subscript in $ORDER() and $QUERY() has the semantic significance of requesting the next lexical item and is not subject to NULLSUBS errors.

LVNULLSUBS allows local arrays to have empty string subscripts.

At process startup, GT.M initializes [NEVER][NO]LVNULLSUBS from $gtm_lvnullsubs. Set the environment variable $gtm_lvnullsubsv to:

- 0 - equivalent to VIEW "NOLVNULLSUBS"

- 1 (the default) - equivalent to VIEW "LVNULLSUBS" or

- 2 - equivalent to VIEW "NEVERLVNULLSUBS".

> **⚠ Important**
>
> Remember that for global variables, empty string subscript checking is controlled by a database region characteristic. FIS recommends using LVNULLSUBS, NOLVNULLSUBS, or NEVERLVNULLSUBS for local variables and NULLSUBS options ALWAYS or NEVER for global variables.

"NOISOLATION":<expr>

where expr must evaluate to one of the following forms:

- "", that is, the empty string : turn off the feature for all globals for which it has previously been turned on

- "^gvn1,^gvn2,..." : turn on the feature for the globals in the list, turning it off for globals for which it has previously been turned on

- "+^gvn1,^gvn2,..." : add these globals to the list of globals that have this feature turned on

- "-^gvn1,^gvn2,..." : turn off the feature for these globals leaving the status for other globals unchanged

GT.M transaction processing permits the application to specify a set of globals that do not require GT.M to preserve Isolation, one of the "ACID" properties of TP. This shifts the responsibility for Isolation from GT.M to the application logic, and permits GT.M to relax its TP Isolation rules. This avoids TP restarts in certain cases thus improving the performance of the application. For example, if a global variable includes $JOB as a subscript, the application may be written and scheduled in such a way that no more than one process uses a node of that global at any given time. Specifying such a global as "NOISOLATED" avoids transaction restarts that occur when different processes concurrently update and access nodes that share the same GDS block.

The rules for enforcement by GT.M of Isolation, and therefore potentially Consistency, are relaxed for application-specified global variables in order to allow the application to manage these properties. GT.M is responsible for Atomicity and Durability, as well as for database integrity for all variables, and for Isolation and Consistency for any global variables for which the application does not accept responsibility.

Note that if an application incorrectly specifies a global to be NOISOLATED, severe, and possibly intermittent and difficult to diagnose damage to application-level integrity is likely to result. A thorough understanding of the application is necessary before declaring a global to be noisolated. GT.M preserves database integrity (accessibility) for NOISOLATED, as well as ISOLATED global variables.

GT.M ignores attempts to turn on (or off) the feature for globals that already have the feature turned on (or off). It is an error to modify the isolation-status of a global variable within a transaction across different references (either reads or writes) of that global variable. The VIEW command by itself is not considered to be a reference of the global variable. While not recommended programming practice, this means that a process can change a global's isolation-status within a transaction as long as it hasn't referenced it yet.

Any reads on a NOISOLATION global are validated at the time of the read and not re-validated at TCOMMIT time. This means that if the value that was read changed after the read but before the TCOMMIT, the transaction would still be committed. Therefore it is important that any reads on a NOISOLATED global (if any) should be of data insensitive to change with time (unchanging or where consistency with other data accessed by the transaction doesn't matter).

"PATCODE":"tablename"

Identifies the alternative table of unique patterns for use with the "?" operator to be loaded from the pattern definition file. For additional information, refer to Chapter 12: "Internationalization" (page 543).

"PATLOAD":"file-specification"

Identifies the file containing definitions of unique patterns for use with the "?" operator. These pattern definitions can be used in place of, or in addition to, the standard C, N, U, L, and P. For more information on creating the file-specification, refer to Chapter 12: "Internationalization" (page 543).

"POOLLIMIT":<region>:expr

VIEW "POOLLIMIT":<region>:expr, where expr is of the form n[%] provides a mechanism for a process that has the potential to "churn" global buffers to limit the potential impact on other processes by restricting the number of global buffers it uses. If the expression ends with a per-cent sign (%), the number is taken as an as a percentage of the configured global buffers and otherwise as an ordinal number of preferred buffers; standard M parsing and integer conversions apply. Preferred buffer values are limited to between 32 and one less than half the buffer pool inclusive; with the exception of zero (0) or 100 per cent, which turn off the limitation; specifications exceeding those limits provide the value of the nearer limit. If the argument specifies "*" for the region, the command applies to all regions. $VIEW("POOLLIMIT",<region>) returns the current value for the region as an ordinal number - zero (0) when there is no limit in place. This facility is designed for use by a relatively small subset of processes. Note that the poollimit mechanism does not completely restrict a process to a limited protion of the pool, but rather tends to restrict it to that portion. In addition, MUPIP REORG uses this facility to limit its buffers to a value established by the UNIX environment variable gtm_poollimit using the syntax described for VIEW "POOLLIMIT" with a default of 64 if gtm_poollimit is not specified. Note that this may slightly slow a standalone REORG but can be overridden by defining gtm_poollimit as 0 or "100%".

RCTLDUMP

Displays the created relinkctl files and the routines looked for in their related directories. An entry in these files does not mean that a given routine was found there. It merely means it was looked for there and shows a cycle number (which ZRUPDATE bumps) whose change indicates a new published version of the given object file. As it is a diagnostic tool for the new feature, FIS may remove or modify this VIEW option in subsequent releases.

> **Note**
>
> GT.M no longer supports VIEW "RCTLDUMP" as it has been supplanted by ZSHOW "A" and MUPIP RCTLDUMP.

RESETGVSTATS

Resets all the process-private global access statistics to 0. This is particularly useful for long running processes which would periodically like to restart the counting without requiring a shut down and restart.

[NO]STATSHARE"[:<region-list>]

VIEW "[NO]STATSHARE"[:<region-list>] enables or disables database statistics sharing for listed regions which permit such sharing. Without the region-list, the command acts on all regions enabled for sharing. When a targeted region has sharing disabled, STATSHARE has no effect.

This provides a fast and efficient mechanism for processes to share their database access statistics for other processes to monitor. Processes opt in or out with the VIEW "[NO]STATSHARE"[:<region-list>] command, defaulting to VIEW "NOSTATSHARE". At process startup, a value of 1, or any case-independent string or leading substrings of "TRUE" or "YES" in the environment variable gtm_statshare provides an initial setting of VIEW "STATSHARE". When a process changes whether it

is opting in or out, there is no change to the output of a ZSHOW "G" within that process. GT.M does not permit this form of the VIEW command within a TP transaction. Monitoring the statistics of other processes does not require opting-in.

The processes which opt-in for STATSHARE place their statistics as binary data in database files located in the directory specified by the gtm_statsdir environment variable. All processes that share statistics MUST use the same value for $gtm_statsdir. The ^%YGBLSTAT utility program gathers and reports statistics.

> **Note**
>
> A VIEW "[NO]STATSHARE" with no region sub-argument opens any unopened mapped regions and any enabled associated statsDB regions; the $gtm_statshare environment variable applies to databases as the application first uses them. When the last VIEW "[NO]STATSHARE" had no region sub-argument, regions implicitly share when the process first references them, but after a VIEW specifies selective sharing, regions don't implicitly share as they open.

STP_GCOL

Starts a string-pool garbage collection, which normally happens automatically at appropriate times.

> **Note**
>
> There are no visible effects from STP_GCOL, LV_GCOL and LV_REHASH except for the passage of time depending on the state of your process. FIS uses these VIEW "LV_GCOL","LV_REHASH","STP_GCOL" facilities in testing. They are documented to ensure completeness in product documentation. You may (or may not) find them useful during application development for debugging or performance testing implementation alternatives.

[NO]UNDEF

Enables or disables handling of undefined variables as errors. With UNDEF, GT.M handles all references to undefined local or global variables as errors. With NOUNDEF, GT.M handles all references to undefined local or global variables as if the variable had a value of the empty string. In other words, GT.M treats all variables appearing in expressions as if they were the argument of an implicit $GET(). UNDEF is the default.

The environment variable $gtm_noundef specifies the initial value value of [NO]UNDEF at process startup. If it is defined, and evaluates to a non-zero integer or any case-independent string or leading substring of "TRUE" or "YES", then GT.M treats undefined variables as having an implicit value of an empty string.

> **Note**
>
> NOUNDEF applies even in the case of an undefined FOR control variable, such as when a KILL or NEW command is used on the control variable, which may cause an unintended indefinite loop. For example, FOR A=1:1:10 KILL A results in an indefinite loop with VIEW "NOUNDEF".

"TRACE":value:<expr>

Traces GT.M program execution and generates profiling information about the lines and functions executed; with low impact on the run-time performance.

The feature turns on (value=1) or turns off (value=0) M-profiling. This expression must evaluate to a string containing the name of a GT.M global variable. The global may also have subscripts; however the subscripts must be literals or the special variable $JOB. For the $JOB process identifier description, refer to Chapter 8: "Intrinsic Special Variables" (page 291).

The expression is optional when turning M-profiling off, if it exists, it overrides the global variable set when M-profiling was turned on.

gtm_trace_gbl_name enables GT.M tracing at process startup. Setting gtm_trace_gbl_name to a valid global variable name instructs GT.M to report the data in the specified global when a VIEW command disables the tracing, or implicitly at process termination. This setting behaves as if the process issued a VIEW "TRACE" command at process startup. However, gtm_trace_gbl_name has a capability not available with the VIEW command, such that if the environment variable is defined but evaluates to zero (0) or, only on UNIX, to the empty string, GT.M collects the M-profiling data in memory and discards it when the process terminates (this feature is mainly used for in-house testing). Note that having this feature activated for process that otherwise do not open a database file (such as GDE) can cause them to encounter an error.

In addition, if a process issues a malformed VIEW command that attempts to turn tracing off, GT.M issues an error but retains all accumulated profiling data and continues tracing. If the tracing is still enabled at the process shutdown and the trace start specified a reporting location, GT.M attempts to place the trace data there. Note that if there is a problem updating the specified trace-reporting global variable, GT.M issues an error at process termination.

M-profiling uses a technique called Basic Block Counting where calls are made to special profiling functions at key points in a GT.M program. A trace consists of the following run-time data as output for each GT.M function, as well as for each GT.M statement:

- The number of times it is executed.

- The total CPU time, subject to the granularity of the operating system provided time functions, spent across all invocations for each function and each GT.M statement as five values: count, user time, system time, total time, and elapsed time.

VIEW "TRACE" also reports details of child processes using two aggregate entries -- "*RUN" for the current process and "*CHILDREN" for all of child processes spawned by the current process, each containing user, system, and combined CPU times. The "CHILD" category data excludes processes that result from the JOB command, PIPE devices OPENed with the INDEPENDENT device parameter and processes from PIPE devices that are still active.

Instead of modifying the generated code as done by common profiling tools, such as gprof, M-profiling operates entirely within the GT.M run-time system; therefore, this feature does not require a special compilation, has no effect on code size and minimizes run-time overhead.

When M-profiling is activated, it gathers profiling information for each line and GT.M function invocation. The reported time for a GT.M line is the time spent in generated code for that line, and does not include time spent in entreyrefs called from that line. When M-profiling is deactivated, the accumulated statistics are loaded into a GT.M global. GT.M profiling accumulates and provides the data; the user chooses tools and techniques to analyze the data.

The M-profiling information is stored in the variable in the following format:

- If the expression is a global variable without subscripts such as "^foo", the M-profiling information is stored in the nodes ^foo(<routine>,<label>) and ^foo(<routine>,<label>,<offset>), each holding a value in the form "<count>:<usertime>,:<systemtime>,:<total_time>".

- If the expression has a value such as "^foo("MYTRACE",$J)", the trace information is stored in the nodes ^foo("MYTRACE",<pid>,<routine>,<label>) and ^foo("MYTRACE",<pid>,<routine>,<label>,<offset>), each of which has a value in the form "<count>,<usertime>,<systemtime>,<total_time>" as described above.

- For FOR loops, information for each level of the loop is stored in the nodes as described above, with the extra subscipts "FOR LOOP". <for_level> is the value of the number of iterations at that level of the FOR loop.

Example:

```
GTM>zprint ^profiling
; In this example, query^profiling, order^profiling, and merge^profling perform the same operation -- store
▶ even-numbered subscripts of a global to a subscripted loc
al variable. M-profiling results show which yields the fastest execution between the three.
profiling
  kill ^TMP,^trc
  view "trace":1:"^trc"
  set ulimit=1500
  for i=1:1:ulimit set ^TMP(i)=i
  do qom("^TMP")
  view "trace":0:"^trc"
  zwrite ^trc
  quit
qom(y)
  do query(y)
  do order(y)
  do merge(y)
  quit
query(y)
  new i,qryval
  set i=0,y=$query(@y)
  for  quit:y=""   do
  .       set:i#2 qryval(i)=@y
  .       set y=$query(@y)
  .       set i=i+1
  quit
order(y)
  new i,ordval
  set x="",i=0,y=y_"(x)",x=$order(@y)
  for  quit:x=""  do
  .       set:i#2 ordval(i)=x
  .       set x=$order(@y)
  .       set i=i+1
  quit
merge(y)
  new i,merval
  set i=0,merval=0
  merge merval=@y
  for i=1:1:$order(merval(""),-1)  do
  .       kill:i#2 merval(i)
  quit
```

▼

On a Ubuntu system running GTM V6.1-000_x86_64, this example produces an output like the following:

```
GTM>do ^profiling
^trc("*CHILDREN")="0:0:0"
^trc("*RUN")="144009:76004:220013"
^trc("profiling","merge")="1:8001:12000:20001:16231"
```

```
^trc("profiling","merge",0)="1:0:0:0:5"
^trc("profiling","merge",1)="1:0:0:0:4"
^trc("profiling","merge",2)="1:0:0:0:4"
^trc("profiling","merge",3)="1:8001:0:8001:8044"
^trc("profiling","merge",4)="1:0:12000:12000:7992"
^trc("profiling","merge",4,"FOR_LOOP",1)=1500
^trc("profiling","merge",5)="1500:0:0:0:4"
^trc("profiling","merge",6)="1:0:0:0:174"
^trc("profiling","order")="1:12001:8001:20002:25720"
^trc("profiling","order",0)="1:0:0:0:8"
^trc("profiling","order",1)="1:0:0:0:6"
^trc("profiling","order",2)="1:0:0:0:90"
^trc("profiling","order",3)="1:0:8001:8001:7160"
^trc("profiling","order",3,"FOR_LOOP",1)=1501
^trc("profiling","order",4)="1500:0:0:0:6319"
^trc("profiling","order",5)="1500:12001:0:12001:12069"
^trc("profiling","order",6)="1500:0:0:0:0"
^trc("profiling","order",7)="1:0:0:0:63"
^trc("profiling","profiling",3)="1:0:0:0:9"
^trc("profiling","profiling",4)="1:52003:20001:72004:74499"
^trc("profiling","profiling",4,"FOR_LOOP",1)=1500
^trc("profiling","profiling",5)="1:0:0:0:14"
^trc("profiling","profiling",6)="1:0:0:0:10"
^trc("profiling","qom")="1:0:0:0:78"
^trc("profiling","qom",0)="1:0:0:0:18"
^trc("profiling","qom",1)="1:0:0:0:11"
^trc("profiling","qom",2)="1:0:0:0:9"
^trc("profiling","qom",3)="1:0:0:0:11"
^trc("profiling","qom",4)="1:0:0:0:5"
^trc("profiling","query")="1:72004:20001:92005:88031"
^trc("profiling","query",0)="1:0:0:0:5"
^trc("profiling","query",1)="1:0:0:0:14"
^trc("profiling","query",2)="1:0:0:0:108"
^trc("profiling","query",3)="1:12000:0:12000:7625"
^trc("profiling","query",3,"FOR_LOOP",1)=1501
^trc("profiling","query",4)="1500:8000:0:8000:28256"
^trc("profiling","query",5)="1500:52004:20001:72005:51919"
^trc("profiling","query",6)="1500:0:0:0:0"
^trc("profiling","query",7)="1:0:0:0:85"
```

- CPU times are reported in microseconds. 1 second = 1,000,000 microseconds.

- ^trc("*CHILDREN")="0:0:0" indicates that the main process did not spawn any child process.

- ^trc("*RUN")="144009:76004:220013" : the three pieces specify the aggregate User Time, System Time and Total Time values for the main process.

- ^trc("profiling","query",3,"FOR_LOOP",1)=1501 specifies the number of times the FOR loop was executed on line #3 of query^profiling.

- ^trc("profiling","merge")="1:8001:12000:20001:16231", ^trc("profiling","order")="1:12001:8001:20002:25720", ^trc("profiling","query")="1:72004:20001:92005:88031": the five pieces specify the aggregate Execution Count, User Time, System,Time, Total Time and the Elapsed Time of the code execution for merge^profiling, order^profling, and query^profiling. merge^profiling has the fastest execution time followed by order^profiling. query^profiling is the slowest amongst the three.

- ^trc("profiling","merge",3)="1:8001:0:8001:8044" and others like it specifies the cumulative Execution Count, User Time, System Time, Total Time and the Elapsed Time of the code execution of line 3 of merge^profiling.

- The M-profiling results are subject to the granularity of the operating system provided time functions. CPU time entries having 0:0:0 values indicate lightweight M mode having 0 to less than 1 microsecond.

Consider the following program that presents the output of this M-profiling result in a tabular report.

```
GTM>zprint ^tracereport
tracereport(gbl,label,rtn)
  set gap=15
  set $piece(x,".",gap*6)="" write x,!
  write "Line #",?gap,"Count",?gap*2,"User Time",?gap*3,"System Time",?gap*4,"Total Time",?gap*5,"Elapsed Time",!
  set $piece(x,".",gap*6)="" write x,!
  for  set gbl=$query(@gbl) quit:gbl=""  do
  .     if ($length(@gbl,":")=5)&($qsubscript(gbl,1)=rtn)&($qsubscript(gbl,2)=label) do
  ..         set gap=15 set lineno=$qsubscript(gbl,3)
  ..         if lineno="" write label," total",?gap set zp=""
  ..         else  write lineno,?gap set zp=label_"+"_lineno_"^"_rtn
  ..         for i=1:1:5 set gap=gap+15 write $piece(@gbl,":",i),?gap
  ..         write !
  ..         set maxlines=$qsubscript(gbl,3)
  for i=0:1:maxlines do
  .     set zp=label_"+"_i_"^"_rtn
  .     write "Line #",i,": ",?9
  .     zprint @zp

GTM>do ^tracereport("^trc","order","profiling")
..........................................................................
Line #          Count           User Time       System Time     Total Time      Elapsed Time
..........................................................................
order total     1               12001           8001            20002           25720
0               1               0               0               0               8
1               1               0               0               0               6
2               1               0               0               0               90
3               1               0               8001            8001            7160
4               1500            0               0               0               6319
5               1500            12001           0               12001           12069
6               1500            0               0               0               0
7               1               0               0               0               63
Line #0: order(y)
Line #1:    new i,ordval
Line #2:    set x="",i=0,y=y_"(x)",x=$order(@y)
Line #3:    for  quit:x=""  do
Line #4:    .       set:i#2 ordval(i)=x
Line #5:    .       set x=$order(@y)
Line #6:    .       set i=i+1
Line #7:    quit
```

This shows that order^profiling has an elapsed time of 25720 and the maximum elapsed time was on line #5, which was executed 1500 times.

```
GTM>do ^tracereport("^trc","merge","profiling")
..........................................................................
Line #          Count           User Time       System Time     Total Time      Elapsed Time
..........................................................................
```

Commands

Line #	Count	User Time	System Time	Total Time	Elapsed Time
merge total	1	8001	12000	20001	16231
0	1	0	0	0	5
1	1	0	0	0	4
2	1	0	0	0	4
3	1	8001	0	8001	8044
4	1	0	12000	12000	7992
5	1500	0	0	0	4
6	1	0	0	0	174

```
Line #0: merge(y)
Line #1:   new i,merval
Line #2:   set i=0,merval=0
Line #3:   merge merval=@y
Line #4:   for i=1:1:$order(merval("")),-1)  do
Line #5:   . kill:i#2 merval(i)
Line #6:   quit
GTM>
```

This shows that merge^profiling has an elapsed time of 16231 and the maximum elapsed time was on line #3, which was executed once.

Note that M-profiling results are reported for each line. While reporting time for a line containing an invocation of a label, M-profiling excludes the execution time of that label.

Here is an example:

```
GTM>do ^tracereport("^trc","qom","profiling")
.....................................................................................
Line #          Count           User Time     System Time     Total Time    Elapsed Time
.....................................................................................
qom total       1               0             0               0             78
0               1               0             0               0             18
1               1               0             0               0             11
2               1               0             0               0             9
3               1               0             0               0             11
4               1               0             0               0             5
Line #0: qom(y)
Line #1:   do query(y)
Line #2:   do order(y)
Line #3:   do merge(y)
Line #4:   quit
```

Notice that the execution of do merge(y) reports an Elapsed Time of 9 whereas merge^profiling reported an Elapsed Time of 1149.

You can write programs like tracereport.m to interpret the results of the M-profiling data and also use them to analyze your code execution path based on your unique requirements.

view "trace":1: "<gbl>" and **view "trace":0: "<gbl>"** commands enable and disable M-profiling.

To perform entryref-specific M-profiling without modifying the source program, use ZBREAK. For example, to perform M-profiling of the entryref merge^profiling, remove VIEW "TRACE" commands from profiling.m and then execute the following commands:

```
GTM>ZBREAK merge^profiling:"view ""TRACE"":1:""^mtrc"" write ""Trace"""
GTM>do ^profiling
Trace
```

```
GTM>view "TRACE":0:"^mtrc"

GTM>zwrite ^mtrc
^mtrc("*CHILDREN")="0:0:0"
^mtrc("*RUN")="132008:52003:184011"
^mtrc("GTM$DMOD","^")="1:0:0:0:4"
^mtrc("profiling","merge")="1:8001:0:8001:13450"
^mtrc("profiling","merge",1)="1:0:0:0:6"
^mtrc("profiling","merge",2)="1:0:0:0:5"
^mtrc("profiling","merge",3)="1:8001:0:8001:6188"
^mtrc("profiling","merge",4)="1:0:0:0:7149"
^mtrc("profiling","merge",4,"FOR_LOOP",1)=1500
^mtrc("profiling","merge",5)="1500:0:0:0:4"
^mtrc("profiling","merge",6)="1:0:0:0:63"
^mtrc("profiling","profiling")="1:0:0:0:9"
^mtrc("profiling","profiling",8)="1:0:0:0:4"
^mtrc("profiling","qom")="1:0:0:0:9"
^mtrc("profiling","qom",4)="1:0:0:0:4"
```

Example:

If prof.m is:

```
prof;
    set start=1
    set finish=1000
    view "TRACE":1:"^trc"
    kill cycle S max=$$docycle(start,finish,"cycle")
    view "TRACE":0:"^trc"
    zwrite ^trc
    quit
    ;
docycle(first,last,var)
    new i,currpath,current,maxcycle,n
    set maxcycle=1
    for current=first:1:last do cyclehelper
    quit maxcycle
    ;
cyclehelper
    set n=current
    kill currpath
    for i=0:1 quit:$data(@var@(n))!(1=n)  D
    .    set currpath(i)=n
    .    do iterate
    if 0<i do
    .    if 1=n set i=i+1
    .    else  set i=i+@var@(n)
    .    do updatemax
    .    set n="" for  set n=$O(currpath(n)) Q:""=n  S @var@(currpath(n))=i-n
    Q
    ;
iterate
    if 0=(n#2) set n=n/2
    else  set n=3*n+1
    quit
    ;
```

```
updatemax
    set:i>maxcycle maxcycle=i
    quit
    ;
```

On executing prof, the output looks like the following (times in the example were chosen for clarity of illustration and are not typical).

```
^trc("*CHILDREN")="0:0:0"
^trc("*RUN")="224014:12000:236014"
^trc("prof","cyclehelper")="1000:200013:0:200013:206318"
^trc("prof","cyclehelper",1)="1000:12001:0:12001:3202"
^trc("prof","cyclehelper",2)="1000:0:0:0:3766"
^trc("prof","cyclehelper",3)="1000:64004:0:64004:94215"
^trc("prof","cyclehelper",3,"FOR_LOOP",1)=3227
^trc("prof","cyclehelper",4)="2227:0:0:0:9864"
^trc("prof","cyclehelper",5)="2227:0:0:0:7672"
^trc("prof","cyclehelper",6)="1000:12000:0:12000:3758"
^trc("prof","cyclehelper",7)="432:0:0:0:1520"
^trc("prof","cyclehelper",8)="432:8000:0:8000:11003"
^trc("prof","cyclehelper",9)="432:0:0:0:3298"
^trc("prof","cyclehelper",10)="432:104008:0:104008:61564"
^trc("prof","cyclehelper",10,"FOR_LOOP",1)=2659
^trc("prof","cyclehelper",11)="1000:0:0:0:3424"
^trc("prof","docycle")="1:12001:0:12001:4886"
^trc("prof","docycle",0)="1:0:0:0:83"
^trc("prof","docycle",1)="1:0:0:0:36"
^trc("prof","docycle",2)="1:0:0:0:4"
^trc("prof","docycle",3)="1:12001:0:12001:4706"
^trc("prof","docycle",3,"FOR_LOOP",1)=1000
^trc("prof","docycle",4)="1:0:0:0:1718579845"
^trc("prof","iterate")="2227:12000:12000:24000:30240"
^trc("prof","iterate",1)="2227:0:0:0:8271"
^trc("prof","iterate",2)="2227:12000:0:12000:7727"
^trc("prof","iterate",3)="2227:0:0:0:7658"
^trc("prof","prof",4)="1:0:0:0:22"
^trc("prof","prof",5)="1:0:0:0:8"
^trc("prof","updatemax")="432:0:0:0:4276"
^trc("prof","updatemax",1)="432:0:0:0:1465"
^trc("prof","updatemax",2)="432:0:0:0:1496"
```

Example:

If fortypes.m is:

```
fortypes;
    new i,j,k,v
    set k=1
    view "TRACE":1:"^trc"
    for i=1:1:3  set v=i
    for i=1:1  set v=0  quit:i=3
    for i=1,2:1:4,6  set v=0
    for i=1:1,2  set v=0  quit:i=3
    for i=1:1:2  for j=1:1:3  set v=0
    for i=1:1:2
    .    for j=1:1:1  do
```

```
    ..            set v=0
    set j=5   for i=1:1:j   do
    .      set j=(j-1)
    for i=1:1:2   for j=1:1:3   do
    .      set v=0
    for i=1:1:2   do
    .        for j=1:1:3   set v=0
    for i=1:1:2   do
    .        for j=1:1:3   do
    ..            set v=0
    for i="foo","bar",1:1   set v=0   quit:i=3
    for   set k=k+1   quit:k=3
    for i=1:1:3   for j=1:1:(3-i)   set v=0
    for i=1:1:3   for j=1:1:(3-i)   for k=1:1:(j+1)   set v=0
    set k=3   view "TRACE":0:"^trc"
    zwrite ^trc
    quit
```

On executing fortypes, the output looks something like the following:

```
^trc("*CHILDREN")="4000:0:4000"
^trc("*RUN")="468029:48003:516032"
^trc("fortypes","fortypes",5)="1:0:0:0:9"
^trc("fortypes","fortypes",5,"FOR_LOOP",1)=3
^trc("fortypes","fortypes",7)="1:0:0:0:6"
^trc("fortypes","fortypes",7,"FOR_LOOP",1)=3
^trc("fortypes","fortypes",9)="1:0:0:0:6"
^trc("fortypes","fortypes",9,"FOR_LOOP",1)=5
^trc("fortypes","fortypes",11)="1:0:0:0:6"
^trc("fortypes","fortypes",11,"FOR_LOOP",1)=3
^trc("fortypes","fortypes",13)="1:0:0:0:8"
^trc("fortypes","fortypes",13,"FOR_LOOP",1)=2
^trc("fortypes","fortypes",13,"FOR_LOOP",2)=6
^trc("fortypes","fortypes",15)="1:0:0:0:4"
^trc("fortypes","fortypes",15,"FOR_LOOP",1)=2
^trc("fortypes","fortypes",19)="1:0:0:0:26"
^trc("fortypes","fortypes",19,"FOR_LOOP",1)=5
^trc("fortypes","fortypes",20)="5:0:0:0:4"
^trc("fortypes","fortypes",22)="1:0:0:0:27"
^trc("fortypes","fortypes",22,"FOR_LOOP",1)=2
^trc("fortypes","fortypes",22,"FOR_LOOP",2)=6
^trc("fortypes","fortypes",23)="6:0:0:0:3"
^trc("fortypes","fortypes",25)="1:0:0:0:11"
^trc("fortypes","fortypes",25,"FOR_LOOP",1)=2
^trc("fortypes","fortypes",26)="2:0:0:0:6"
^trc("fortypes","fortypes",26,"FOR_LOOP",1)=6
^trc("fortypes","fortypes",28)="1:0:0:0:8"
^trc("fortypes","fortypes",28,"FOR_LOOP",1)=2
^trc("fortypes","fortypes",29)="2:0:0:0:26"
^trc("fortypes","fortypes",29,"FOR_LOOP",1)=6
^trc("fortypes","fortypes",30)="6:0:0:0:4"
^trc("fortypes","fortypes",32)="1:0:0:0:8"
^trc("fortypes","fortypes",32,"FOR_LOOP",1)=5
^trc("fortypes","fortypes",34)="1:0:0:0:5"
^trc("fortypes","fortypes",34,"FOR_LOOP",1)=2
^trc("fortypes","fortypes",36)="1:0:0:0:8"
```

```
^trc("fortypes","fortypes",36,"FOR_LOOP",1)=3
^trc("fortypes","fortypes",36,"FOR_LOOP",2)=3
^trc("fortypes","fortypes",38)="1:0:0:0:14"
^trc("fortypes","fortypes",38,"FOR_LOOP",1)=3
^trc("fortypes","fortypes",38,"FOR_LOOP",2)=3
^trc("fortypes","fortypes",38,"FOR_LOOP",3)=7
```

"ZDATE_FORM":"value"

Determines whether four digit year code is active for $ZDATE() function. GT.M defaults to zero (0), that is, two digit output. For more usage information, refer to "$ZDate()" (page 262).

If no value is given with the VIEW command, it turns four digit code on. It is equivalent to the intrinsic special variable $ZDATEFORM. Use $ZDATEFORM to set this VIEW keyword. Also, logical name environment variable gtm_zdate_form may be used to set the initial value to this factor.

Examples of VIEW

Example:

```
GTM>Kill A

GTM>View "NOUNDEF"
GTM>Write A,?10,$L(A)
         0
GTM>
```

This demonstrates how a VIEW that specifies NOUNDEF prevents UNDEFined errors.

Example 2:

```
GTM>ZLink "NOSENSE"
%GTM-E-LABELMISSING Label referenced but
not defined:lab
%GTM-I-SRCNAM in source module /home/gtmuser1/.fis-gtm/V5.4-002B_x86/r/
NOSENSE.m
GTM>ZPrint ^NOSENSE
NOSENSE;
        Do lab
        Quit
LAB  Write !,"THIS IS NOSENSE"
        Quit
GTM>View "LABELS":"UPPER"
GTM>ZLink "NOSENSE.m"
GTM>Do ^NOSENSE
THIS IS NOSENSE
GTM>
```

This demonstrates use of VIEW "LABELS" to make label handling case insensitive. Notice that the routine was ZLINKed with an extension of .m to force a recompile and ensure that the object code and the run-time handling of labels is the same.

Write

The WRITE command transfers a character stream specified by its arguments to the current device.

The format of the WRITE command is:

```
W[RITE][:tvexpr] expr|*intexpr|fcc[,...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- An expression argument supplies the text of a WRITE.

- When a WRITE argument consists of a leading asterisk (*) followed by an integer expression, WRITE outputs one ASCII character associated with the ASCII code specified by the integer evaluation of the expression.

- WRITE arguments may also be format control characters; format control characters modify the position of a virtual cursor: an exclamation point (!) produces a new line, a number-sign (#) produces a new page and a question-mark (?) followed by an expression moves the virtual cursor to the column specified by the integer evaluation of the expression provided that the virtual cursor is to the "left" of the specified column; if the virtual cursor is not to the left of the specified column, then the text is printed at the current cursor position.

- An indirection operator and an expression atom evaluating to a list of one or more WRITE arguments form a legal argument for a WRITE.

- In the UTF-8 mode, the WRITE command uses the character set specified on the device OPEN as the character encoding of the output device. If character set specifies "M" or "UTF-8", GT.M WRITEs the data with no transformation. If character set specifies "UTF-16", "UTF-16LE" or "UTF-16BE", the data is assumed to be encoded in UTF-8 and WRITE transforms it to the character encoding specified by character set device parameter.

- If a WRITE command encounters an illegal character in UTF-8 mode, it produces a run-time error irrespective of the setting of VIEW "BADCHAR".

See Also

- "Write" (page 439)
- "WRITE Command" (page 383)
- "Deviceparameter Summary Table" (page 444)

Xecute

The XECUTE command makes an entry in the GT.M invocation stack and executes the argument as GT.M code.

The format of the XECUTE command is:

```
X[ECUTE]:tvexpr expr[:tvexpr][,...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The required expression specifies a fragment of GT.M source code. The maximum length of the expression is 8192 bytes.

- The optional truth-valued expression immediately following the argument expression specifies the argument postconditional and controls whether GT.M performs an XECUTE with that argument.

- An indirection operator and an expression atom evaluating to a list of one or more XECUTE arguments form a legal argument for an XECUTE.

- Run-time errors from indirection or XECUTEs maintain $STATUS and $ZSTATUS related information and cause normal error handling but do not provide compiler supplied information on the location of any error within the code fragment.

An explicit or implicit QUIT within the scope of the XECUTE, but not within the scope of any closer DO, FOR, XECUTE or extrinsic, returns execution to the instruction following the calling point. This may be the next XECUTE argument or another command. At the end of the code specified by the XECUTE argument expression, GT.M performs an implicit QUIT.

Because XECUTE causes run-time compilation in GT.M, and because it tends to obscure code, use XECUTE only when other approaches clearly do not meet your particular requirement.

GT.M compiles XECUTE <literal> at compile time when the literal is valid GT.M code that has minimal impact on the M virtual machine. An XECUTE literal containing GOTO, NEW, QUIT, (nested) XECUTE and indirection can't be precompiled because of the interaction of those features with the stack architecture of the M virtual machine. Precompiled XECUTE literals do not show up in $STATCK() as having a separate stack level, but rather "disappear" into the stack level of the original XECUTE. Please observe the following cautions:

- ensure you compile with the same GT.M version, $gtm_chset, $gtm_local_collate, $gtm_patnumeric, $gtm_pattern_file and $gtm_pattern_table values (or lack thereof) as those used to run your application.

- If the application changes the run time values controlled by those environment variables, use variable operands or indirection, rather than literals for operands with pattern match (?) or sorts-after (]]).

Note that indirection almost always performs better than an XECUTE that can't be precompiled. Note also that adding a QUIT at the end of an XECUTE that does not contain a FOR will leave it for run time compilation.

Examples of XECUTE

Example:

```
GTM>Xecute "Write ""HELLO"""
HELLO
GTM>
```

This demonstrates a simple use of Xecute.

Example:

```
Set x="" For Set x=$Order(^%x(x)) Quit:x=""  Xecute x
```

This $ORDER() loop XECUTEs code out of the first level of the global array ^%x. Note that, in most cases, having the code in a GT.M source file, for example TMPX.m, and using a Do ^TMPX improves efficiency.

See Also	"Trigger Definition File" (page 592) "$ZTrap" (page 331) "Exception Handling Facilities" (page 8)

ZAllocate

The ZALLOCATE command reserves the specified name without releasing previously reserved names. Other GT.M processes cannot reserve the ZALLOCATEd name with a ZALLOCATE or LOCK command.

The ZALLOCATE command provides compatibility with some other GT.M implementations. The M Development Committee chose to add the + and - delimiters to the LOCK command (incremental locking) rather than adopt the ZALLOCATE and ZDEALLOCATE approach. Therefore, when a design requires an incremental lock mechanism, LOCK +/- has the advantage over ZALLOCATE / ZDEALLOCATE of being part of the M standard. LOCK +/- also has the advantage of working symmetrically when routines using LOCKs are nested. That is, a ZALLOCATE command issued by a process for a named resource already ZALLOCATEd by that process results in no change of state. This means that routines that do ZALLOCATE followed by a ZDEALLOCATE on a named resource that is already ZALLOCATEd by the same process (at routine entry time), will end up ZDEALLOCATEing the named resource (which might not be desired). On the other hand, a LOCK + command issued by a process for a named resource already LOCKed by that process causes the LEVEL of the LOCK to be incremented (as seen in a ZSHOW "L" output). Every LOCK - command on that named resource causes the LEVEL to be decremented. When the LEVEL becomes 0, the named resource is no longer LOCKed.

For more information on troubleshooting LOCKs with the GT.M Lock Utility (LKE), refer to the appropriate chapter of the **GT.M Administration and Operations Guide**.

The format of the ZALLOCATE command is:

```
ZA[LLOCATE][:tvexpr] [(]nref[,...][)][:intexpr][,...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The nref argument specifies a name in the format of a GT.M name with or without subscripts, and with or without a preceding caret (^).

- Outside of transactions, only one process in an environment can ZALLOCATE (or LOCK) a particular resource name at any given time.

- Because the data storage in GT.M uses hierarchical sparse arrays and ZALLOCATE may serve to protect that data from inappropriate "simultaneous" access by multiple processes, ZALLOCATE treats resource names in a hierarchical fashion; a ZALLOCATE protects not only the named resource, but also its ancestors and descendants.

- When one or more nrefs are enclosed in parentheses (), ZALLOCATE reserves all the enclosed names "simultaneously," that is, it reserves none of them until all become available.

- The optional numeric expression specifies a time in seconds after which the command should timeout if unsuccessful; choosing 0 results in a single attempt. If a ZALLOCATE command specifies a timeout that do not exceed $ZMAXTPTIME and the resource name is locked on the final retry, the process may generate TPNOACID messages while it tries to ensure there is no possibility of a deadlock.

- An indirection operator and an expression atom evaluating to a list of one or more ZALLOCATE arguments form a legal argument for a ZALLOCATE.

For additional information on the GT.M locking mechanism, refer to the "LOCK" section in the M LOCK Utility chapter of **GT.M Administration and Operations Guide**.

If a ZALLOCATE command specifies a timeout, and GT.M acquires ownership of the named resource before the timeout elapses, ZALLOCATE sets $TEST to TRUE (1). If GT.M cannot acquire ownership of the named resource within the specified timeout, ZALLOCATE sets $TEST to FALSE (0). If a ZALLOCATE command does not specify a timeout, the execution of the command does not affect $TEST.

When given a list of nrefs, ZALLOCATE tries to reserve each nref from left to right in the order specified taking into account the timeout specified for each. If the timeout elapses before reserving an nref, GT.M terminates the ZALLOCATE command. Any nrefs already acquired as part of the current ZALLOCATE command stay acquired.

Examples of ZALLOCATE

Examples:

```
ZAllocate A
ZAllocate ^A
ZAllocate ^A(1)
ZAllocate (^B("smith"),^C("jones"))
ZAllocate @A
```

The first command ZALLOCATEs A; the second, ^A; the third, ^A(1) and the fourth, both ^B("smith") and ^C("jones") simultaneously. The last command ZALLOCATEs the resources named by the value of the variable A.

Example:

```
ZAllocate A,^B,@C
ZALLOCATE (A,B,C)
```

If ZALLOCATE arguments are enclosed in parentheses, the command waits until all names in the argument list become available before reserving any of the names. For example, in the statement ZA (A,B,C), if the resource named C is not available, ZALLOCATE waits until C becomes available before reserving A and B. Using the format illustrated in the first line above, can cause deadlocks because the resource names are reserved as they come available.

When a process attempts to ZALLOCATE a name currently ZALLOCATEd or LOCKed (with the LOCK command) by another process, the ZALLOCATEing process hangs until the other process releases the name. In the event that names remain unavailable for significant periods of time, timeouts allow the process issuing a ZALLOCATE to regain program control.

Example:

```
ZAllocate ^D:5
```

This example specifies a timeout of five seconds. If GT.M reserves ^D before the five seconds elapses, ZALLOCATE sets $TEST to TRUE. If GT.M cannot reserve ^D within the five second timeout, ZALLOCATE sets $TEST to FALSE.

At the time of ZALLOCATEing a name, no names previously reserved with ZALLOCATE or the LOCK command are released (similarly, LOCKing a name does not release names that have been ZALLOCATEd). For example, after ZALLOCATEing A and LOCKing B, LOCKing B does not release A, and ZALLOCATEing C does not release A or B.

ZDEALLOCATE releases ZALLOCATED resource names. The ZDEALLOCATE command can only release previously ZALLOCATEd (not LOCKed) names.

Resource name arguments for LOCKs and ZALLOCATEs intersect. That is, if one process holds a LOCK or ZALLOCATE, another process can neither LOCK nor ZALLOCATE any name falling in the hierarchy of the resource name held by the first process. When a process holds a LOCK or ZALLOCATE, that same process may also LOCK or ZALLOCATE resource names falling in the hierarchy of the currently held resource name. When a single process holds both LOCKs and ZALLOCATEs, a LOCK does not release the ZALLOCATEd resource(s) and a ZDEALLOCATE does not release the LOCKed resource(s).

Also see the description of the ZDEALLOCATE command described later in this chapter.

Example:

```
Lock ^AR(PNT)
.
.
.
```

```
ZAllocate ^AR(PNT,SUB)
.
.
.
Lock ^TOT(TDT)
.
.
ZDEALLOCATE ^AR(PNT,SUB)
```

This LOCKs ^AR(PNT)and all its descendents, then, after performing some unspecified commands, it ZALLOCATEs ^AR(PNT,SUB). ZALLOCATE does not imply any change to LOCKs or existing ZALLOCATEd resource names, therefore, the LOCK of ^AR(PNT) remains in effect. ^AR(PNT,SUB) is already protected by the LOCK. Next, because an unsigned LOCK releases all resource names currently LOCKed by the process, the routine releases ^AR(PNT) with the LOCK of ^TOT(TDT). This leaves the ZALLOCATE of ^AR(PNT,SUB). The name ^AR and all its subscripts except for ^AR(PNT) and those that begin with ^AR(PNT,SUB) are now available for LOCKing by other processes. Finally the routine releases ^AR(PNT,SUB) with a ZDEALLOCATE command. The ZDEALLOCATE does not affect the LOCK on ^TOT(TDT). Note that this example was constructed to illustrate the interaction between LOCK, ZALLOCATE and ZDEALLOCATE, and not to illustrate sound programming practice..

Because the ZALLOCATE command reserves names without releasing previously reserved names, it can lead to deadlocks. For example, a deadlock occurs if two users ZALLOCATE names A and B in the following sequence:

Deadlock Situation	
USER X	USER Y
ZAllocate A	ZAllocate B
ZAllocate B	ZAllocate A

To avoid deadlocks, use a timeout. Because unsigned LOCKs always release previously reserved names, such LOCKs inherently prevent deadlocks.

ZALLOCATE Operation Summary		
PREEXISTING CONDITION	COMMAND ISSUED	RESULT
Another user reserved M	ZA M	Your process waits
	LOCK M	Your process waits
	ZD M	No effect
You reserved M with LOCK M	ZA M	M is ZALLOCATEd and LOCKed; use both ZDEALLOCATE and LOCK (L or L -M) to clear M
	LOCK M	Release M and reserve M again
	ZD M	No effect
You reserved M with ZA M	ZA M	No effect
	LOCK M	M is ZALLOCATEd and LOCKed; use both ZDEALLOCATE and LOCK (L or L -M) to clear M

ZALLOCATE Operation Summary		
PREEXISTING CONDITION	**COMMAND ISSUED**	**RESULT**
	ZD M	No effect

ZBreak

The ZBREAK command sets or clears routine breakpoints during debugging.

The format of the ZBREAK command is:

```
ZB[REAK][:tvexpr] [-]entryref[:[expr][:intexpr]][,...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The required entryref specifies a location within a routine or a trigger at which to set or remove a breakpoint.

- The optional minus sign (-) specifies that ZBREAK remove the breakpoint; -* means remove all breakpoints.

- The optional expression specifies a fragment of GT.M code to XECUTE when GT.M execution encounters the breakpoint; if the ZBREAK argument does not specify an action, the default action is "BREAK".

- The optional integer expression immediately following the expression specifies a count of process transits through the breakpoint before the breakpoint action takes effect; once GT.M exhausts the count and the action takes effect, the action occurs every time the process encounters the breakpoint. If the action expression is omitted, the optional integer expression must be separated from the entryref by two adjacent colons (::).

- An indirection operator and an expression atom evaluating to a list of one or more ZBREAK arguments form a legal argument for a ZBREAK.

- If a concurrent process reloads a trigger in which a process has an active ZBREAK, GT.M automatically removes the breakpoint and issues a TRIGZBRKREM warning message when it refreshes the trigger; the TRIGZBRKREM warning message respects a message mask of 8 as maintained by the VIEW "BREAKMSG" command.

When GT.M encounters the entryref, GT.M suspends execution of the routine code and XECUTEs the breakpoint action before executing any of the commands on the line. For more information on entryrefs, see Chapter 5: "General Language Features of M" (page 68).

When the optional integer expression is used, GT.M activates the breakpoint on the intexpr-th time the process encounters the breakpoint during routine execution. Once GT.M activates the breakpoint, that breakpoint remains active for the process until explicitly replaced or removed, or until the process terminates.

For more information, refer to Chapter 4: "Operating and Debugging in Direct Mode" (page 50).

Examples of ZBREAK

Example:

```
GTM>ZPRint ^ZBTEST
ZBTEST;
    Do SUB
    Quit
```

```
SUB  Write !,"This is ZBTEST"
     Quit
GTM>ZBREAK SUB^ZBTEST
GTM>Do ^ZBTEST
%GTM-I-BREAKZBA, Break instruction encountered during ZBREAK action
At M source location SUB^ZBTEST
GTM>ZSHOW "B"
SUB^ZBTEST
```

This inserts a ZBREAK with a default action at SUB^ZBTEST. After GT.M encounters the BREAK, the ZSHOW "B" displays this as the only ZBREAK in the image.

Example:

```
GTM>ZBREAK -*

GTM>ZGOTO
GTM>ZBREAK SUB^ZBTEST:"W !,""Trace"""
GTM>Do ^ZBTEST
Trace
This is ZBTEST
GTM>
```

This removes all existing ZBREAKs with a ZBREAK -*. Note that it is not necessary to remove ZBREAKs before modifying them. It also clears the process invocation stack with an argumentless ZGOTO. Then it uses a ZBREAK to insert a trace-point. Every time GT.M executes the line to where ZBREAK has established a trace-point, it performs the specified action without entering Direct Mode.

Example:

```
ZBreak PRINT^TIME::5
```

This BREAKs execution at line PRINT in routine just before the fifth time the line is executed.

Example:

```
ZBREAK PRINT^TIME:"WRITE AVE BREAK":3
```

This inserts a ZBREAK action of WRITE AVE and BREAK before the third execution of PRINT^TIME.

ZCOMpile

The ZCOMPILE command invokes the GT.M compiler from within the GT.M run-time environment.

Within GT.M itself, ZCOMPILE provides the functionality of the mumps command, except for mumps -direct.

The format of the ZCOMPILE command is:

```
ZCOM[PILE][:tvexpr] expr[,...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The expression argument specifies one or more filenames, which may optionally include the .m extension. If the file specification does not include a .m extension, ZCOMPILE assumes a default file extension of ".m". Wildcards are acceptable in the filename specification. The filename specification can be optionally prefixed by qualifiers valid for a mumps command.

For a description of the arguments and qualifiers of the mumps command, refer to Chapter 3: "Development Cycle" (page 32).

The $ZCSTATUS intrinsic special variable which holds the value of the status code for the compilation performed by a run-time compilation command.

If the argument does not include compilation qualifiers, the commmand uses the $ZCOMPILE ISV for possible qualifiers.

Examples of ZCompile

Examples:

```
ZCOMPILE "EXAMPLE'.m"
```

This compiles EXAMPLE.m in the current working directory.

Example:

```
ZCOMPILE "-list A*.m"
```

This compiles all files starting with a [capital] A and an extension of .m in the current working directory and produces corresponding listing files for each source / object.

ZContinue

The ZCONTINUE command continues routine execution after a BREAK command or a <CTRL-C>.

The format of the ZCONTINUE command is:

```
ZC[ONTINUE][:tvexpr]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- Because ZCONTINUE changes the flow of execution away from control of the principal device back to the current routine, it is usually the final command on a line; however, if it is not, because the ZCONTINUE has no argument, at least two (2) spaces must follow the command to separate it from the next command on the line.

- If the process is not in Direct Mode, ZCONTINUE has no effect.

For more information, refer to Chapter 4: "Operating and Debugging in Direct Mode" (page 50).

ZDeallocate

The ZDEALLOCATE command releases a specified resource name or names previously reserved by the ZALLOCATE command. The ZDEALLOCATE command releases only the specified name(s) without releasing other names previously reserved with the ZALLOCATE or LOCK command.

The ZDEALLOCATE command provides compatibility with some other GT.M implementations. The M Development Committee choose to add the + and - delimiters to the LOCK command rather than adopt the ZALLOCATE and ZDEALLOCATE approach. Therefore, when a design requires an incremental lock mechanism, LOCK +/- has the advantage

of being part of the M standard. LOCK +/- also has the advantage of working symmetrically when routines using LOCKs are nested.

The format of the ZDEALLOCATE command is:

```
ZD[EALLOCATE][:tvexpr] [nref[,...]]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command

- The nref argument specifies a name in the format of a GT.M name with or without subscripts and with or without a leading caret (^).

- A ZDEALLOCATE with no argument releases all names currently reserved with ZALLOCATE by the process; in this case, at least two (2) spaces must follow the ZDEALLOCATE to separate it from the next command on the line.

- ZDEALLOCATEing a named resource that is not currently owned by the process has no effect.

- An indirection operator and an expression atom evaluating to a list of one or more ZDEALLOCATE arguments form a legal argument for a ZDEALLOCATE.

Examples of ZDEALLOCATE

Example:

For examples of ZALLOCATE, refer to "Examples of ZALLOCATE" (page 172).

ZEDit

The ZEDIT command invokes the editor specified by the EDITOR environment variable for GT.M and opens the specified file for editing. If the EDITOR environment variable is undefined, ZEDIT tries to invoke the UNIX vi editor.

By default, ZEDIT puts a new file into the first source directory in $ZROUTINES. You can specify a file path explicitly in the argument to the ZEDIT command, for example: the current working directory:

```
ZEDIT "./file"
```

The format of the ZEDIT command is:

```
ZED[IT][:tvexpr] [expr[,...]]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The optional expression(s) specifies the name of a file to edit; note the argument is an expression rather than a routinename; ZEDIT rejects arguments with a file extension of .o as illegal. A valid GT.M file name with no extension will be given an extension of .m; therefore it is not possible, through ZEDIT, to edit a file with a valid GT.M filename and no extension.

- If ZEDIT has an argument, it not only invokes the editor, but also sets $ZSOURCE=expr.

- If ZEDIT has no argument or expr="", the command acts as a ZEDIT $ZSOURCE; at least two (2) spaces must follow a ZEDIT command with no argument to separate it from the next command on the line.

- GT.M stores source code in files with standard operating system format; generally the file name is the same as the GT.M routinename with a default extention or type of .m.

- An indirection operator and an expression atom evaluating to a list of one or more ZEDIT arguments form a legal argument for a ZEDIT

If the expression includes a directory, ZEDIT searches only that directory. If $ZROUTINES is not null, a ZEDIT command that does not specify a directory uses $ZROUTINES to locate files. If $ZROUTINES is equal to an empty string, ZEDIT edits a file in the current working directory. For more information on $ZROUTINES, see the appropriate section in Chapter 8: "Intrinsic Special Variables" (page 291).

When the argument to a ZEDIT includes a file or path name, $ZSOURCE maintains that as a default for ZEDIT and ZLINK. For more information on $ZSOURCE see the appropriate section in Chapter 8: "Intrinsic Special Variables" (page 291).

Examples of ZEDIT

Example:

```
GTM>ZEDIT "BAL"
```

This invokes the editor for a file with a name of BAL and an extension of .m. Notice that BAL is a string literal.

Example:

```
GTM>Set prog="BAL"

GTM>ZEDit prog
```

This is similar to the first example except that it uses a variable argument rather than a string literal.

Example:

```
GTM>zedit ".login"
```

This invokes the editor for a file with the name .login. Notice that in this case the file is not a GT.M file, since .login starts with a period, and therefore, cannot be a GT.M file.

ZGoto

The ZGOTO command transfers control to various levels in the GT.M invocation stack. It also can transfer control from one part of the routine to another or from one routine to another using the specified entryref.

The format of the ZGOTO command is:

```
ZG[OTO][:tvexpr] [[intexpr][:entryref[:tvexpr]],...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The optional integer expression specifies the stack frame nesting level reached by performing the ZGOTO. If the optional integer expression specifies a negative level, ZGOTO treats it as $zlevel-intexpr.

- A ZGOTO with no argument returns control to the next command at the bottom of the stack (level 1); in this case, at least two (2) spaces must follow the command to separate it from the next command on the line.

- The optional entryref specifies a location to which ZGOTO transfers control.

- If ZGOTO specifies no entryref, it returns control to the next command at the level specified by the integer expression.

- The optional truth-valued expression immediately following the entryref specifies the argument postconditional and controls whether GT.M uses the argument.

- If the ZGOTO includes the level and the argument postconditional but not the entryref, two colons (::) separate the integer expression from the truth-valued expression.

- An indirection operator and an expression atom evaluating to a list of one or more ZGOTO arguments form a legal argument for a ZGOTO.

- ZGOTO accepts a trigger entryref (with a trailing hash-sign (#)); if the trigger is not currently loaded (by some previous trigger action), GT.M generates a ZLINKFILE error. Note that ZGOTO should be reserved for error handling and testing, as it is a very unstructured operation.

A ZGOTO command with an entryref performs a similar function to the GOTO command, with the additional capability of reducing the GT.M stack level. In a single operation, ZGOTO executes ($ZLEVEL - intexpr) implicit QUITs and a GOTO operation, transferring control to the named entryref. For more information on entryrefs, refer to Chapter 5: "General Language Features of M" (page 68).

The ZGOTO command leaves the invocation stack at the level specified by the integer expression. GT.M implicitly terminates any intervening FOR loops and unstacks variables stacked with NEW commands as appropriate.

A ZGOTO 0 from a call-in unwinds all the M stack frames and returns to the invoking C routine. For all other cases, ZGOTO 0 terminates the process.

Using ZGOTO 0:entryref invokes the "unlink all" facility. It allows a process to disassociate itself from all routines it has linked, releases memory, and continue execution with entryref as the only current entry in the M virtual stack. ZGOTO 0:entryref preserves local variables and IO devices across this transition and performs the following:

- Stops M-profiling (if active).

- Unwinds all routines in the M stack.

- Unlinks all routines, releases allocated memory, and closes any shared libraries containing GT.M generated object code.

- Purges all cached objects (code generated for XECUTE and indirection).

- Resets $ECODE, $REFERENCE, and $TEST to their initial (empty) values.

ZGOTO resembles HALT (and not QUIT) in that it causes an exit regardless of the number of active levels in the current invocation. ZGOTO resembles QUIT (and not HALT) in that it destroys the GT.M context and terminates the process only if the current GT.M invocation is at the base of the process. Understanding the difference between ZGOTO and HALT has an impact only in an environment where GT.M is invoked recursively from other languages.

ZGOTO $ZLEVEL:LABEL^ROUTINE produces identical results to GOTO LABEL^ROUTINE. ZGOTO $ZLEVEL-1 responds like a QUIT (followed by ZCONTINUE, if in Direct Mode). If the integer expression evaluates to a value greater than the current value of $ZLEVEL or less than zero (0), GT.M issues a run-time error.

If ZGOTO has no entryref, it performs some number of implicit QUITs and transfers control to the next command at the specified level. If ZGOTO has no argument, it behaves like ZGOTO 1, which resumes operation of the lowest level GT.M routine as displayed by ZSHOW "S". In the image invoked by $gtm_dist mumps -direct, a ZGOTO without arguments returns the process to Direct Mode.

ZGOTO provides a useful debugging tool in Direct Mode. However, because ZGOTO is not conducive to structured coding, it is best to restrict its use in production programs to error handling. For more information on GT.M error handling, refer to Chapter 13: "Error Processing" (page 562).

Examples of ZGOTO

Example:

```
GTM>ZGOTO
GTM>ZSHow
+1^GTM$DMOD (Direct mode)
GTM>
```

This uses ZGOTO to clear all levels of the GT.M invocation stack. ZSHOW with no arguments displays the stack.

Example:

```
SET $ZTRAP="ZGOTO "_$ZLEVEL_":^ERROR"
```

This SETs $ZTRAP to contain a ZGOTO, so if an error causes GT.M to XECUTE $ZTRAP, the routine ERROR executes at the same level as the SET command shown in the example.

ZHALT

The ZHALT command stops program execution and causes GT.M to return control to the invoking environment/program with a return code.

The format of the ZHALT command is:

```
ZHALT[:tvexpr] [intexpr]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether GT.M executes the command.

- The optional integer expression specifies the return code. If an integer expression is not specified, ZHALT returns 0. Because UNIX limits return codes to zero through 255, ZHALT returns intexpr modulo 256, unless the intexpr is non-zero but the intexpr modulo 256 is zero, in which case ZHALT returns a (non-success) value of 255 so that the return code is non-zero.

- If no arguments are specified, at least two (2) spaces must follow the command to separate it from the next command on the line. Note that additional commands do not serve any purpose unless the ZHALT has a postconditional.

- A ZHALT releases all shared resources held by the process, such as devices OPENed in GT.M, databases, and GT.M LOCKs. If the process has an active M transaction (the value of $TLEVEL is greater than zero (0)), GT.M performs a ROLLBACK prior to terminating.

Examples of ZHALT

Example:

```
GTM>zhalt 230
$ echo $?
230
```

Example:

```
GTM>zhalt 257
$ echo $?
1
```

ZHelp

The ZHELP command accesses the help information from the GTM help library or from any help library specified in the command argument.

The format of the ZHELP command is:

```
ZH[ELP][:tvexpr] [expr1[:expr2],...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The optional first expression specifies the help topic.

- If ZHELP has no argument or expr1="", ZHELP invokes base level help; at least two (2) spaces must follow a ZHELP command with no argument to separate it from the next command on the line.

- The optional second expression specifies the name of a Global Directory containing ^HELP.

- If ZHELP does not specify the second expression, the Global Directory defaults to $gtm_dist/gtmhelp.gld.

- An indirection operator and an expression atom evaluating to a list of one or more ZHELP arguments form a legal argument for a ZHELP

Examples of ZHELP

Example:

```
GTM>zhelp "func $data"
```

This lists the help for function $DATA, which is a subtopic of functions topic.

Example:

```
GTM>zhelp
```

This uses ZHELP to list all the keywords in the help library.

Example:

```
GTM>zhelp "ZSHOW"
```

This lists the help for command ZSHOW.

ZLink

The ZLINK command adds an executable GT.M routine to the current process if the current process does not contain a copy of a routine. If the current process contains a copy of a routine and the routine is not active, the ZLINK command replaces the

current routine process with a "new" version. If necessary, the ZLINK command compiles the routine prior to integrating it with the process.

With VIEW "LINK":"RECURSIVE" specified or by starting the process with the environment variable gtm_link set to "RECURSIVE", the ZLINK command adds an executable routine even when a routine with the same name is active and available in the current stack. When a process links a routine with the same name as an existing routine, future calls use the new routine. Prior versions of that routine referenced by the stack remain tied to the stack until they QUIT, at which point they become inaccessible. This provides a mechanism to patch long-running processes.

> ⚠️ **Important**
>
> An active routine is displayed with $STACK() or ZSHOW "S" of the M virtual stack. By default, an attempt to replace an active routine results in a run-time error . To replace an active routine with a new version, either use VIEW "LINK":"RECURSIVE" or remove the active routine from the stack using ZGOTO or the appropriate number of QUITs and then execute the ZLINK command.

The format of the ZLINK command is:

```
ZL[INK][:tvexpr] [expr1[:expr2][,...]]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The optional first expression specifies the pathname of a routine to ZLINK; if ZLINK has an argument, it not only adds the routine to the image, but also sets $ZSOURCE=expr.

- If ZLINK has no argument, or expr="", it uses value of $ZSOURCE as the routine specification filename; at least two (2) spaces must follow a ZLINK command with no argument to separate it from the next command on the line.

- The optional second expression specifies a string holding MUMPS command qualifiers delimited by a dash (-); the qualifiers control compile options when the current ZLINK requires a compile; if ZLINK omits the second expression, the command uses the $ZCOMPILE intrinsic special variable to determine the compile qualifiers.

- An indirection operator and an expression atom evaluating to a list of one or more ZLINK arguments form a legal argument for a ZLINK.

- When ZLINK encounters a bad object file it produces an INVOBJFILE error that identifies the problem object file.

When the ZLINK command specifies a file, GT.M sets $ZSOURCE to that filename. By default, ZLINK and ZEDIT use $ZSOURCE for a filename when they have a missing or null argument. A subsequent ZLINK without an argument is equivalent to ZLINK $ZSOURCE. For more information on $ZSOURCE, see the appropriate section in Chapter 8: "Intrinsic Special Variables" (page 291).

> 📌 **Note**
>
> In order to ensure compatibility with GT.M versions that do not permit the percent sign (%) in a file name, use an underscore (_) in place of the percent in the ZLINK file name for routines beginning with a percent sign.

If the expression includes an explicit directory, ZLINK searches only that directory. Otherwise, if $ZROUTINES is not null, a ZLINK command uses $ZROUTINES to locate files. If $ZROUTINES is null, ZLINK uses the current directory. For more information on $ZROUTINES, see the appropriate section in Chapter 8: "Intrinsic Special Variables" (page 291).

If the filename contains an explicit file extension, ZLINK processes the file according to the extension, object (.o) or source (usually .m). If the file name does not specify a file extension, ZLINK attempts to find and match both the object and source for a routine.

The following table illustrates how ZLINK processes the three possibilities of file extension.

ZLINK Operation Summary			
EXTENSION SPECIFIED	EXTENSION SOUGHT BY ZLINK		RESULT
	.o	.m	
.o	found	N/A	link only
	not found	N/A	error
.m	N/A	found	compile and link
	N/A	not found	error
None	not found	found	compile and link
	found	not found	link only
	not found	not found	error
	found .o file newer than .m and version okay	found .m file older than .o	link only
	found .o file older than .m or version mismatch	found .m file newer than .o	compile and link

ZLINK Compilation

If ZLINK compiles a routine and the -OBJECT= qualifier does not redirect the output, it places the resulting object file in the directory indicated by the search criteria. ZLINK incorporates the new object file into the image, regardless of its directory placement.

If the command does not specify compile qualifiers (with expr2) and $ZCOMPILE is null, GT.M uses the default M command qualifiers, -ignore, -labels=lower, -nolist, and -object. For more information on $ZCOMPILE, refer to the appropriate section in Chapter 8: "Intrinsic Special Variables" (page 291). For detailed descriptions of the M command qualifiers, see Chapter 3: "Development Cycle" (page 32).

For information on producing object files, but not adding them to the current image, see "ZCOMpile" (page 175).

Examples of ZLINK

Example:

```
GTM>ZLINK "test"
```

If ZLINK finds test.m or test.o, it adds the routine test to the current image. If ZLINK does not find test.o, or finds that test.o is older than test.m, GT.M compiles test.m to produce a new test.o, and adds the contents of the new object file to the image. This example assumes "test" is not on the current M stack - if it is on the stack, GT.M gives an error.

Example:

```
GTM>zlink "test.m":"-noobject -list"
```

This compiles the routine "test" and produces a listing but no object file. Because the example produces no object file, it must locate an existing object file (which might be the same as any copy in the current image); if there is noexisting object file, GT.M produces an error. While this example shows the use of compilation qualifiers with ZLINK, a -noobject -list compilation might better be done with ZCOMPILE.

Example:

```
GTM>zlink "sockexamplemulti2"
%GTM-E-LOADRUNNING, Cannot ZLINK an active routine sockexamplemulti2
GTM>zshow "S"
sockexamplemulti2+12^sockexamplemulti2     (Direct mode)
GTM>view "LINK":"RECURSIVE"
GTM>zlink "sockexamplemulti2"
GTM>
```

This example demonstrates how VIEW "LINK":"RECURSIVE" command ZLINKs a routine when its prior version is already there in the active M virtual stack.

Auto-ZLINK

If a GT.M routine refers to a routine that is not linked in the process memory, GT.M automatically attempts to ZLINK that routine. An auto-ZLINK is functionally equivalent to an explicit ZLINK of a routine without a specified directory or file extension.

The following GT.M commands and functions can initiate auto-ZLINKing:

- DO

- GOTO

- ZBREAK

- ZGOTO

- ZPRINT

- $TEXT()

GT.M auto-ZLINKs the routine if the following conditions are met:

- ZLINK can locate and process the routine file, as indicated in the previous ZLINK Operation Summary table

- The name of the routine is the same as the name of the source file; the only exception is that GT.M converts a leading percent sign (%) in a file name to an underscore (_).

Auto-ZLINK setup

This section describes the procedure to setup the auto-relink functionality. GT.M loads an object file linked from an object directory (in $ZROUTINES) with a *-suffix (i.e. auto-relink-enabled) into a shared memory segment (referred to henceforth as a Rtnobj shared memory segment). At the invocation of DO, GOTO, or ZGOTO, extrinsic functions, ZBREAK, ZPRINT or

$TEXT() that specify an entryref which includes a routine name (in contrast to a label without a routine name), GT.M processes (and MUPIP processes executing trigger logic) automatically relink ("auto-relink") and execute published new versions of routines.

> **Note**
>
> - Label references (that is, without a routine name), whether direct or through indirection, always refer to the current routine, and do not invoke auto-relink logic.
>
> - Use shell quoting rules when appending asterisks to directory names in the gtmroutines environment variable - asterisks must be passed in to GT.M, and not expanded by the shell.
>
> - GT.M accepts but ignores asterisk suffixes to directory names on 32-bit Linux on x86 platforms, where it does not provide auto-relinking.

The ZRUPDATE command publishes of new versions of routines to subscribers. To remove routines, delete the object files and publish the names of the deleted object files. Removal requires file names to be explicitly specified, because patterns with wildcards cannot match deleted files.

If the path to a file is non-existent, the request is ignored except in the case where one desires a currently shared object file (one that was accessed before it was deleted) to no longer be shared.

For each auto-relink enabled directory which a GT.M process accesses while searching through $ZROUTINES, GT.M creates a small control file (Relinkctl) in the directory identified by $gtm_linktmpdir (defaulting to $gtm_tmp, which in turn defaults to /tmp, if unspecified). The names of these files are of the form gtm-relinkctl-<murmur> where <murmur> is a hash of the realpath() to an auto-relink directory; for example: **/tmp/gtm-relinkctl-f0938d18ab001a7ef09c2bfba946f002)**. With each Relinkctl file, GT.M creates and associates a block of shared memory that contains associated control structures. Among the structures is a cycle number corresponding to each routine found in the routine directory; a change in the cycle number informs a process that it may need to determine whether there is a new version of a routine. Although GT.M only creates relinkctl records for routines that actually exist on disk, it may increment cycle numbers for existing relinkctl records even if they no longer exist on disk.

GT.M creates both the Relinkctl file and shared memory with permissions based on the logic described in the "IPC Permissions" column of the "Shared Resource Authorization Permissions" section in the Administration and Operations Guide, except that the object directory, rather than the database file, provides the base permissions.

The MUPIP RCTLDUMP command reports information related to relinkctl files and their associated shared memory segments.

The environment variable gtm_autorelink_keeprtn if set to 1, t[rue], or y[es] causes exiting processes to leave auto-relinked object code in the shared memory repositories, while if undefined, 0, f[alse] or n[o] causes exiting processes to purge any routines currently use by no processes. All values are case-independent. When gtm_autorelink_keeprtn is defined and TRUE:

- Process exit is simplified, with the performance gain - faster process termination - likely to be observable only when a large number of processes exit concurrently.

- Where routines are likely to be repeatedly used by other processes, such as in a production environment, leaving a routine in shared memory even when no longer used by existing processes, results in slightly faster linking of that routine by future processes, although the effect may not be observable except when an application frequently uses short-lived processes, such as GT.M routines invoked by web servers using a CGI interface.

FIS recommends that a directory in the $zroutines of a process be either auto-relink-enabled or auto-relink-disabled for the life of the process. Changing the auto-relink mode of the directory within a process is likely to result in counter-intuitive results..

As arguments, ZRUPDATE takes object file names, including wild-cards of the form accepted by $ZSEARCH(). If ZRUPDATE fails to find at least one file to match an argument with a wild card, it issues an INFO message (seen only if $PRINCIPAL has CENABLE). When the argument specifies an explicit name without a wild card, but there is no file in the directory or a corresponding entry in the Relinkctl, ZRUPDATE produces an error. ZRUPDATE issues most errors as FILEPARSE errors with a secondary error describing the actual issue although some errors, depending on the reason and path by which ZRUPDATE detects them, can be rather cryptic.

An explicit ZLINK or an auto-relink check the hash of an object and its replacement. If they are identical, GT.M may take no action to replace the current object, saving both memory and time.GT.M bypasses the dynamic link of a routine when it determines the requested object matches the currently linked object file under either of the following circumstances:

- auto-relink from the same directory

- explicit ZLINK where neither the original object nor the new object are auto-relinked.

In other cases GT.M always performs the dynamic link.

An explicit ZLINK from an auto-relink directory acts as an implicit ZRUPDATE.

Any ZBREAK in a routine disables that routine from auto-relinking by a process until all ZBREAKs are removed.

If recursive relink is not enabled, routines currently active in the M virtual machine stack are disabled from auto-relinking until they complete (or are removed from the stack by a ZGOTO).

Auto-zlink Benefits and Example

The benefits of auto-relink are as follows:

- Auto-relink provides the convenience of automatically running the current routine under most conditions.

  When combined with VIEW "LINK":"RECURSIVE", auto-relink automatically relinks routines even when they are active and available in the current stack. While it is possible to run auto-relink without VIEW "LINK":"RECURSIVE", routines currently active in a stack do not auto-relink and, if explicitly ZLINK'd, induce a LOADRUNNING error until they complete or are removed from the stack.

- Use of auto-relink loads routine object files into the shared memory. Therefore, the use of a given routine by multiple processes results in significant memory savings (one copy per system instead of one copy per user). This is analogous to the memory sharing from using shared object libraries, but allows dynamic updates, where shared libraries do not.

- When combined with routines explicitly compiled with the -embed_source option or auto-compiled with $ZCOMPILE set to "-embed_source", auto-relink may improve the performance of $TEXT() and ZPRINT as they access source code from shared memory instead of the disk.

- When $gtm_autorelink_keeprtn is defined and TRUE, applications that frequently invoke GT.M routines in short running processes (such as those over interfaces like CGI) may give better performance because it keeps routines in shared memory so that they can be reused when short running processes need them.

The use and setup of the auto-relink facility depends upon the requirements. Here is an example:

```
$ /usr/lib/fis-gtm/V6.2-001_x86_64/gtm
GTM>w $zroutines
/home/jdoe/.fis-gtm/V6.2-001_x86_64/o*(/home/jdoe/.fis-gtm/V6.2-001_x86_64/r /home/jdoe/.fis-gtm/r)
  /usr/lib/fis-gtm/V6.2-001_x86_64/plugin/o/_POSIX.so /usr/lib/fis-gtm/V6.2-001_x86_64/plugin/o(/usr/lib/fis-gtm/
V6.2-001_x86_64/plugin/r) /usr/lib/fis-gtm/V6.2-001_x86_64/libgtmutil.so /usr/lib/fis-gtm/V6.2-001_x86_64
```

▼

In $ZROUTINES, the *-suffix after the object directory enables the auto-relink facility. By default, the gtm/gtmprofile scripts that are available as part of GT.M distribution on sourceforge.net have auto-relink enabled.

With auto-relink enabled, GT.M loads an object file from an object directory into Rtnobj shared memory segment on an explicit ZLINK, implicit ZLINK (DO, GOTO, ZPRINT, $TEXT()), and extrinsic function invocations ($$) enabling the routines to be accessed by other concurrent/future processes.

> **Note**
>
> With auto-relink, GT.M creates an initial Rtnobj shared memory segment of 1 MiB (2 MiB or more if hugepages is configured) and allocates 92MiB of shared memory segment for managing the auto-relink facility. Therefore, always ensure that your system has adequate shared memory configured; if not, GT.M displays messages along the lines of:
>
> ```
> %GTM-E-SYSCALL, Error received from system call shmget() failed
> ```
>
> Refer to your OS documentation to configure shared memory limits (for example, on common Linux systems, the kernel.shmmax parameter in /etc/sysctl.conf).
>
> If your routines require more MiB shared memory, set the environment variable $gtm_autorelink_shm to an integer value (in powers of two). When auto-relink needs more shared memory for storing routines, GT.M automatically allocates twice the size of $gtm_autorelink_shm MiB for auto-relink operations.

```
GTM>zedit "myprogram.m"
GTM>
```

ZEDIT puts a new file into the first source directory in $ZROUTINES, that is, in the /home/jdoe/.fis-gtm/V6.2-001_x86_64/r directory.

```
GTM>do ^myprogram
```

The first invocation of an implicit ZLINK (DO, GOTO ZGOTO, ZPRINT, $TEXT() or function/extrinsic invocation) or an explicit ZLINK "myprogram.m" or ZRUPDATE "/home/jdoe/.fis-gtm/V6.2-001_x86_64/myprogram.o" creates a Relinkctl file if one does not already exist and the associated shared memory. The relinkctl file has a name associated with the hash of the directory to provide a pointer in the form of segment ids to shared memory so that processes can locate routines.

As the gtm_linktmpdir environment variable is not set by default in the gtm/gtmprofile scripts, GT.M stores the Relinkctl file in the directory pointed to by the gtm_tmp environment variable.

```
GTM>zshow "A"
Object Directory : /home/jdoe/.fis-gtm/V6.2-001_x86_64/o
Relinkctl filename : /tmp/fis-gtm/V6.2-001_x86_64/gtm-relinkctl-43b26ca8384ddbf74b94d90a830c0bc9
# of routines : 1
# of attached processes : 1
Relinkctl shared memory : shmid: 375586821 shmlen: 0x5800000
Rtnobj shared memory # 1 : shmid: 375619590 shmlen: 0x200000 shmused: 0x400 shmfree: 0x1ffc00 objlen: 0x280
rec#1: rtnname: myprogram cycle: 1 objhash: 0xd81f1cdcc275e13d numvers: 1 objlen: 0x280 shmlen: 0x400
```

ZSHOW "A" command displays information related to relinkctl file and the routine records that it points to in the shared memory segments. The routine records appears in the order in which they were inserted into the shared memory in the context of the relinkctl file.

```
GTM>zedit "myprogram2.m"
```

ZEDIT puts a new file into the first source directory in $ZROUTINES, that is, in the /home/jdoe/.fis-gtm/V6.2-001_x86_64/r directory.

```
GTM> zrupdate "/home/jdoe/.fis-gtm/V6.2-001_x86_64/o/*.o"
```

The ZRUPDATE command increments the cycle counter of those routine records whose object hash is different than the one last loaded in the Rtnobj shared memory. In this case, it would be rec#2, that is, myprogram.o. ZRUPDATE does not recompile/relink the routines. Instead, it instructs all current and future processes that the object code is out-of-date and must be auto-relinked (if required) on the next invocation. An explicit ZLINK or an auto-relink (whichever happens first) checks the hash of an object and its replacement and initiates recompile/relink on finding that are they not identical.

```
GTM>zshow "A":zru
```

If there are hundreds of routines, a command like ZSHOW "A":zru transfers the output of ZSHOW "A" to a local variable. The following example uses that result to display the information on all routines that contain the string passed to the disprtn function:

```
GTM>zprint ^disprtn
disprtn(rtn)
set x="" for set x=$order(zru("A",x)) quit:x="" write:$piece(zru("A",x),":",3)[rtn zru("A",x),!
quit ""
```

and produces a result like the following:

```
GTM>w $$^disprtn("myprogram")
rec#1: rtnname: myprogram2 cycle: 1 objhash: 0x436c855d5891e7cf numvers: 1 objlen: 0x370 shmlen: 0x400
rec#2: rtnname: myprogram cycle: 1 objhash: 0xd81f1cdcc275e13d numvers: 1 objlen: 0x280 shmlen: 0x400
GTM>
```

ZLINK, auto-ZLINK and Routine Names

In GT.M, the name of the source file determines the name of the GT.M routine. The file name of the object file is not required to match the name of the routine. Linking the object file makes the internal routine name (derived from the source file) known to GT.M. This can lead to potential confusion, however, since both ZLINK and auto-ZLINK use the name of the object file to find the routine. When the object file name differs from the name of the routine, auto-ZLINK generates a run-time error.

Note

Auto-ZLINK and ZLINK commands without a .m or .o file extension in their argument determine the need to recompile based on whether the object file was more recently modified than the source file using time in nanoseconds, as provided by the underlying system call. Note that, although the format of the file modification timestamps provides a nanosecond granularity, many supported OSs currently update the file timestamps with an accuracy of one second.

ZKill

The ZKILL command KILLs the data value for a variable name without affecting the nodes descended from that node.

The format of the ZKILL command is:

```
ZK[ILL][:tvexpr] glvn
```

The functionality of ZKILL is identical to ZWITHDRAW. For a comprehensive description of the format and usage, refer to "ZWIthdraw" (page 208).

ZMessage

The ZMESSAGE command raises an exception condition based on the specified message code.

The format of the ZMESSAGE command is:

```
ZM[ESSAGE][:tvexpr] intexpr[:expr2][:...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The required integer expression specifies the message code. There are two types of message codes:

  - Message codes from 150339592 are raised from GT.M. For examining the text of a message code, refer to $ZMESSAGE().

    The three least significant bits (lsb) of these message codes indicate the severity which determines the error handling action:

3 lsb	Severity	Action
0	Warning	XECUTEs $ETRAP or $ZTRAP and terminates the process
1	Success	Displays the associated message on STDERR* and continues execution. It does not invoke $ETRAP or $ZTRAP.
2	Error	XECUTEs $ETRAP or $ZTRAP
3	Information	Displays the associated message on STDERR* and continues execution. It does not invoke $ETRAP or $ZTRAP.
4	Severe/Fatal	Displays the associated message on STDERR and terminates the process.
5,6,7	Unassigned/Unsupported	-

  *: STDERR of the mumps process

- Message codes between 1 and 132 come from OS services. ZMESSAGE treats all such codes as a either a trappable error or a fatal event.

- ZMESSAGE can be used as a tool to simulate an error condition. The additional expressions specified after a colon ":" are the ordered context substitutions for the given exception condition. For example, if the message associated with the condition contains a substitution directive, passing a string as an additional expression causes the string to be inserted in the message text dat the point of the corresponding substitution directive.

- ZMESSAGE transforms two sets of error messages into SPCLZMSG errors:

  - The internal error messages which should not be user visible.

  - The error messages which are expected to be driven when their corresponding internal state is available. The list of such errors is as follows: CTRLC, CTRAP, JOBINTRRQST, JOBINTRRETHROW, REPEATERROR, STACKCRIT, SPCLZMSG, TPRETRY, UNSOLCNTERR.

  - ZMESSAGE is conceptually similar to SET $ECODE=",<expr>,".

Examples of ZMESSAGE

All of the following examples issue ZMESSAGE from Direct Mode where exception conditions do not invoke $ZTRAP.

Example:

```
GTM>ZMessage 2
%SYSTEM-E-ENO2, No such file or directory
```

This ZMESSAGE does not specify substitution text and the message does not include any substitution directives.

Example:

```
GTM>ZMESSAGE 150372994
%GTM-E-GVUNDEF, Global Variable undefined:
```

The message specified by this ZMESSAGE command includes a substitution directive but the command does not supply any text.

Example:

```
GTM>ZMESSAGE 150373850:"x"
%GTM-E-GVUNDEF, Undefined local variable: x
```

This ZMESSAGE command supplies the substitution text for the message.

GT.M treats its own odd-numbered conditions as "successful." GT.M handles successful conditions by displaying the associated message and continuing execution. GT.M treats its own even-numbered conditions as failures. GT.M handles failure conditions by storing the error information in $ZSTATUS and XECUTEing $ETRAP or $ZTRAP In Direct Mode, GT.M only reports failure conditions to the principal device and does not XECUTE $ETRAP or $ZTRAP or set $ZSTATUS; if $PRINCIPAL is in CENABLE mode, GT.M sends it Informational messages which are not errors but a form of success. System service errors do not follow the GT.M odd/even pattern.

ZPrint

The ZPRINT command displays the source code lines selected by its argument.

The format of the ZPRINT command is:

```
ZP[RINT][:tvexpr][entryref[:label[+intexpr]]][,...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- A ZPRINT with no argument prints the entire current routine or the current trigger. The current routine is the routine closest to the top of an invocation stack, as displayed by a ZSHOW "S"; in this case, at least two (2) spaces must follow the command to separate it from the next command on the line.

- The optional entryref specifies the location in a routine at which to start printing; the entryref can include either a routinename or a label plus a routinename in the format LABEL^ROUTINENAME or LABEL+OFFSET^ROUTINENAME; if the entryref does not contain a routinename, ZPRINT defaults to the current routine.

- The optional label following the entryref identifies a location at which to stop printing; the optional integer expression specifies an offset from the label; the label and offset together are referred to as a lineref and this lineref identifies the last line to print; if the offset is specified without the label, the offset in the optional lineref is always counted from the beginning of the routine, even when the entryref specifies a label.

- If the ZPRINT argument includes the colon (:) delimiter, then the argument must also include at least one component of the optional lineref.

- If the ZPRINT argument contains only the entryref, with no components of the optional lineref and the entryref contains a label or offset, ZPRINT displays only the one line that occurs at that entryref.

- If the entryref contains only a routinename, ZPRINT displays the entire routine.

- If the entryref contains a trigger name, ZPRINT displays its trigger code.

- If the entryref contains only a routinename and the argument includes the optional lineref, ZPRINT starts the display at the beginning of the routine.

- If the optional lineref specifies a line prior to the lineref specified within the entryref, ZPRINT does not display any lines.

- If the offset in the optional lineref specifies a line beyond the end of the routine, ZPRINT displays the remainder of the routine.

- If ZPRINT cannot locate the routine or if either of the labels does not appear in the routine, ZPRINT issues an error.

- An indirection operator and an expression atom evaluating to a list of one or more ZPRINT arguments form a legal argument for a ZPRINT.

Note that the routinename may only appear before the colon (:) delimiter. The integer expression offsets may be positive or negative, but they must always be delimited by a plus sign (+).

For more information on entryrefs, refer to Chapter 5: "General Language Features of M" (page 68).

Examples of ZPRINT

Example:

```
GTM>ZPRINT X^RTN
```

This example displays the line beginning with the label X in the routine RTN.

Example:

```
GTM>ZPRINT X^RTN:X+5

GTM>ZPRINT X+-5^RTN:X
GTM>ZPRINT X^RTN:X+-5^RTN
```

The first line displays the line beginning with the label X and the next 5 lines in routine RTN. The second line displays the 5 lines preceding label X in the same routine and the line beginning with label X. The third line generates a run-time error because the routine name must appear only before the colon in the argument.

Example:

```
GTM>zprint ^A#1#
 do ^test1
 do stop^test2
GTM>
```

This command displays the trigger code for trigger name A#1#.

```
ZPRINT ^x#/BREG : Print trigger routine user-named "x" in region BREG
ZPRINT ^x#1#/BREG : Print trigger routine auto-named "x#1" in region BREG
ZPRINT ^x#1#A/BREG : Print trigger routine auto-named "x#1", runtime disambiguated by "#A", AND in region BREG
ZPRINT +1^x#1#A/BREG : Print line 1 of trigger routine auto-named "x#1", runtime disambiguated by "#A", AND in
 region
▶ BREG
```

▼

These are some examples of disambiguator combinations.

ZRUPDATE

Publishes the new versions of routines to subscribers. The format of the ZRUPDATE command is:

```
ZRUP[DATE][:tvexpr] expr [,...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- expr contains a list of object file names, with or without wildcards, which ZRUPDATE publishes new versions to subscribers.

- To remove routines, delete the object files and publish the names of the deleted object files. Removal requires file names to be explicitly specified, because patterns with wildcards cannot match deleted files.

- ZRUPDATE rejects file-name arguments that are symbolic links or start with a percent-sign (%)

- ZRUPDATE recognizes question-mark (?) as a single character wild-card

- If the path to a file is non-existent, the request is ignored except in the case where one desires a currently shared object file (one that was accessed before it was deleted) to no longer be shared.

- To effect auto-relink, GT.M creates small temporary files in the directory referred to by $gtm_linktmpdir (defaulting to $gtm_tmp, which in turn defaults to /tmp, if unspecified). The names of these files are of the form gtm-relinkctl<md5sum> where <md5sum> is a hash of the realpath() to an auto-relink directory. The group and permissions match those for the directory as described in the section Shared Resources Authorization Permissions in Appendix E (GT.M Security Philosophy) of the UNIX Administration and Operations Guide. FIS recommends that all processes that share a directory whose contents are subject to ZRUPDATE use the same value for $gtm_linktmpdir so that all processes see update notifications - with different values of $gtm_linktmpdir, a ZRUPDATE by a process with one value of $gtm_linktmpdir would not be observed by a process with a different value of that environment variable.

- ZRUPDATE always updates the existing shared memory relinkctl information for a file with an existing entry.

ZSHow

The ZSHOW command displays information about the current GT.M environment.

The format of the ZSHOW command is:

```
ZSH[OW][:tvexpr][expr[:glvn][,...]]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The optional expression specifies one or more codes determining the nature of the information displayed.

- A ZSHOW with no argument defaults to ZSHOW "S"; in this case, at least two (2) spaces must follow the ZSHOW to separate it from the next command on the line.

- The optional global or local variable name specifies the destination for the ZSHOW output; if the ZSHOW argument does not contain a global or local variable name, ZSHOW directs its display to the current device ($IO).

- When the desination for the ZSHOW output is a local variable or the current device ($IO), ZSHOW sets the maximum length of a ZSHOW line output to 8192 bytes. ZSHOW stores information that does not fit within 8192 bytes in the next line.

- When the destination for the ZSHOW output is a global variable, ZSHOW sets the maximum length of a ZSHOW line output to the maximum database record size. ZSHOW stores information that does not fit within the maximum database record size as immediate descendants, using ordinal subscripts starting at one (1), of the node holding the beginning of the information.

- When the destination for the ZSHOW "V" output is a global variable, the %ZSHOWVTOLCL utility program can be used to restore data from that global variable into its original local variables. For more information refer to " %ZSHOWVTOLCL" (page 493).

- An indirection operator and an expression atom evaluating to a list of one or more ZSHOW arguments form a legal argument for a ZSHOW.

ZSHOW Information Codes

A ZSHOW argument is an expression containing a list of codes that select one or more types of information. The ZSHOW information codes are:

Code	Description
*	is the wildcard code which is same as ZSHOW "IVBDLGRC"; all excluding codes "A", "S", and "T".
A	the code "A" stands for Autorelink and provides output in the same format as MUPIP RCTLDUMP, but restricted to the routines contained in the relinkctl areas in use by the process issuing the command. ZSHOW "*" does not include ZSHOW "A" because of an expectation that the typical volume of the information does not provide a good return for its value. If you wish your error handling or INTRPT routines to dump this information, ask for it explicitly,

Code	Description
	possibly by doing a ZSHOW "A" into a local variable before doing a ZSHOW "*".
B	displays active ZBREAK breakpoints and their associated actions
C	provides the list of loaded external call packages and their routines. ZSHOW "C" does not report packages that are accessible but have not been accessed by the process.
D	displays information on currently OPEN devices; it tends to display information on device states that differ from the default
G	displays the access statistics for global variables and access to database file since process startup. When the process does not have access to the current shared statistics, ZSHOW "G" returns a question-mark (?) at the end of the output strings.
I	displays the current values of all intrinsic special variables
L	displays GT.M LOCKs and ZALLOCATEs held by the process
R	displays the GT.M invocation stack and a hash based on the MurmurHash3 algorithm of M source code for each routine on the stack.
S	displays the GT.M invocation stack - like R, but without the hash
T	displays the cross-region summary (total) lines associated with G and L codes. Lines associated with G end with a question-mark (?) when the process does not have access to the current shared statistics.
V	displays local and alias variables that are active in the current context

Codes may be upper- or lower-case. Invalid codes produce a run-time error. Multiple occurrences of the same code in one ZSHOW argument only produce one output instance of the corresponding information. The order of the first appearance of the codes in the argument determines the order of the corresponding output instances.

If you are using a local variable destination and place another code ahead of "V", the effect is to have the results of the earlier code also appear in the results of the "V" code.

When the wildcard (*) occurs in the list, it overrides all other codes and displays codes in the order of "IVBDLGRC" which produces the following information:

- intrinsic special variables

- local variables

- ZBREAK information

- device information

- LOCK and ZALLOCATE information

- Access statistics for global variables and database files(s)

- GT.M invocation stack and an MD5 checksum of M source code for each routine on the stack

- External call table entry name

If G occurs in the list, the following statistics are displayed. The table is in alphabetical order, but ZSHOW will display the statistics according to historical practice with the most recently added statistics displayed last. The statistics are displayed in a comma-separated list where each item has its mnemonic followed by a colon and a counter. GT.M reports counters in DECIMAL. Each counter has 8-byte (can get as high as 2**64). If these counters exceed 18 decimal digits (somewhere between 2**59 and 2**60), which is the current GT.M numeric representation precision threshold, their use in arithmetic expressions in GT.M results in loss of precision. The mnemonics are:

```
AFRA: # of waits for instance freeze to release critical sections
BREA: # of waits for block read & decryption
BTD : # of database Block Transitions to Dirty
BTS : # of times a dirty buffer was flushed so a BT could be reused
BUS : # of times db_csh_get could not determine whether a block was in cache or not
CAT : Critical section Total Acquisitions successes
CFE : Critical section Failed (blocked) acquisition total caused by Epochs
CFS : Square of CFT
CFT : Critical section Failed (blocked) acquisition Total
CQS* : Critical section acquisition Queued sleeps sum of Squares
CQT* : Critical section acquisition Queued sleeps Total
CTN : Current Transaction Number of the database for the last committed read-write transaction (TP and non-TP)
CYS* : Critical section acquisition processor Yields sum of Squares
CYT* : Critical section acquisition processor Yields Total
DEX : # of Database file EXtentions
DEXA: # of waits for database extension
DFL : # of Database FLushes of the entire set of dirty global buffers in shared memory to disk
DFS : # of times a process does an fsync of the database file. For example: a) after writing an epoch journal record, b) as part
 of database file extension c) during database rundown d) as part of mupip reorg -truncate etc.
DRD : # of Disk ReaDs from the database file (TP and non-TP, committed and rolled-back).This does not include reads that are
 satisfied by buffered globals for databases that use the BG (Buffered Global) access method.
DTA : # of DaTA operations (TP and non-TP)
DWT : # of Disk WriTes to the database file (TP and non-TP, committed and rolled-back). This does not include writes that
 are satisfied by buffered globals for databases that use the BG (Buffered Global) access method. GT.M always reports 0 for
 databases that use the MM (memory-mapped) access method as this has no real meaning in that mode.
GET : # of GET operations (TP and non-TP)
GLB : # of waits for bg access critical section
IDXH: # of index block cache hits
IDXM: # of index block cache misses
INC : # of $INCREMENT operations. The $INCREMENT function also increases the SET counter.
JBB : # of Journal Buffer Bytes updated in shared memory
JEX : # of Journal file EXtentions
JFB : # of Journal File Bytes written to the journal file on disk. For performance reasons, GT.M always aligns the beginning
 of these writes to file system block size boundaries. JFB counts all bytes including those needed for alignment in order to
 reflect the actual IO load on the journal file. Since the bytes required to achieve alignment may have already been counted
 as part of the previous JFB, processes may write the same bytes more than once, causing the JFB counter to typically be higher
 than JBB.
JFL : # of Journal FLushes of all dirty journal buffers in shared memory to disk. For example: when switching journal files etc.
JFS : # of Journal FSync operations on the journal file. For example: when writing an epoch record, switching a journal file
 etc.
JFW : # of Journal File Write system calls
JNL : # of waits for journal access critical section
JOPA: # of waits for journal open critical section
JRE : # of Journal Regular Epoch records written to the journal file (only seen in a -detail journal extract); these are written
 every time an epoch-interval boundary is crossed while processing updates
JRI : # of JouRnal Idle epoch journal records written to the journal file (only seen in a -detail journal extract); these are
 written when a burst of updates is followed by an idle period, around 5 seconds of no updates after the database flush timer
 has flushed all dirty global buffers to the database file on disk
```

```
JRL : # of Journal Records with a Logical record type (e.g. SET, KILL etc.) written to the journal file
JRO : # of Journal Records with a type Other than logical written to the journal file (e.g. AIMG, EPOCH, PBLK, PFIN, PINI, and
 so on)
JRP : # of Journal Records with a Physical record type (i.e. PBLK, AIMG) written to the journal file (these records are seen
 only in a -detail journal extract)
KIL : # of KILl operations (kill as well as zwithdraw, TP and non-TP)
KTG : # of of invoked KILL triggers
LKF : # of LocK calls (mapped to this db) that Failed
LKS : # of LocK calls (mapped to this db) that Succeeded
MLBA: # of waits for blocked LOCK
MLK : # of waits for LOCK access
NBR : # of Non-tp committed transaction induced Block Reads on this database
NBW : # of Non-tp committed transaction induced Block Writes on this database
NR0 : # of Non-tp transaction Restarts at try 0
NR1 : # of Non-tp transaction Restarts at try 1
NR2 : # of Non-tp transaction Restarts at try 2
NR3 : # of Non-tp transaction Restarts at try 3
NTR : # of Non-tp committed Transactions that were Read-only on this database
NTW : # of Non-tp committed Transactions that were read-Write on this database
ORD : # of $ORDer(,1) (forward) operations (TP and non-TP); the count of $Order(,-1) operations are reported under ZPR.
PRC : # of waits on exit
PRG : # of pre-read globals that were performed by the reader helper
QRY : # of $QueRY() operations (TP and non-TP)
SET : # of SET operations (TP and non-TP)
STG : # of invoked SET triggers
TBR : # of Tp transaction induced Block Reads on this database
TBW : # of Tp transaction induced Block Writes on this database
TC0 : # of Tp transaction Conflicts at try 0 (counted only for that region which caused the TP transaction restart)
TC1 : # of Tp transaction Conflicts at try 1 (counted only for that region which caused the TP transaction restart)
TC2 : # of Tp transaction Conflicts at try 2 (counted only for that region which caused the TP transaction restart)
TC3 : # of Tp transaction Conflicts at try 3 (counted only for that region which caused the TP transaction restart)
TC4 : # of Tp transaction Conflicts at try 4 and above (counted only for that region which caused the TP transaction restart)
TR0 : # of Tp transaction Restarts at try 0 (counted for all regions participating in restarting TP transaction)
TR1 : # of Tp transaction Restarts at try 1 (counted for all regions participating in restarting TP transaction)
TR2 : # of Tp transaction Restarts at try 2 (counted for all regions participating in restarting TP transaction)
TR3 : # of Tp transaction Restarts at try 3 (counted for all regions participating in restarting TP transaction)
TR4 : # of Tp transaction Restarts at try 4 and above (restart counted for all regions participating in restarting TP
 transaction)
TRB : # of Tp read-only or read-write transactions Rolled Back (excluding incremental rollbacks)
TRGA: # of mini-transaction completion
TRX : # of waits for transaction in progress
TTR : # of Tp committed Transactions that were Read-only on this database
TTW : # of Tp committed Transactions that were read-Write on this database
WFL : # of database flushes that were performed by the writer helpers
WFR : # of times a process slept while waiting for another process to read in a database block
WHE : # of writer helper epochs
WRL : # of times a process consistently slept(longer than WFR) while waiting for another process to read in a database block
ZAD : # of waits for region freeze off
ZPR : # of $order(,-1) or $ZPRevious() (reverse order) operations (TP and non-TP). The count of $Order(,1) operations are
 reported under ORD.
ZTG : # of of invoked ZTRIGGERs
ZTR : # of ZTRigger command operations
[NT]B[WR] mnemonics are satisfied by either disk access or, for databases that use the BG (buffered global) access method,
 global buffers in shared memory.
GT.M maintains the counters of the mnemonics marked with a * on AIX. On Linux, GT.M does not currently increment the counters of
 these mnemonics but retains them in the ZSHOW output for backward compatibility.
```

If an operation is performed inside a TP transaction, and not committed as a consequence of a rollback, or an explicit or implicit restart, GT.M still counts it.

KILL/GET/DATA/QUERY/ORDER/ZPREVIOUS operations on globals that never existed are not counted, while the same operations on globals that once existed but have since been killed are counted.

Name-level ORDER/ZPREVIOUS operations (for example, $ORDER(^a) with no subscripts) increment the count for each transition into a region as they process the global directory map up to the point they find a global with data.

Index block hit ratios for global buffers are IDXH/(IDXH+IDXM) but underrepresent total index block usage because of process-private caching.

> **Note**
>
> The use of comma-separated pieces for ZSHOW "G" allows for future releases of GT.M to provide additional data while facilitating upward compatibility of application code. Since FIS reserves the right to change the order in which statistics are reported in future versions of GT.M, application programs should use the names (mnemonics) when picking pieces from the string instead of relying on field position or ordering.

In addition, "G" also displays a line containing aggregated statistics (GLD:*,REG:* line) for all database files for the global directory and region name. If two or more regions (in the same or different global directories) map to the same physical database file, the ZSHOW "G" reports identical statistics for those two regions, but counts them only once across all database files in this line. It always reports the value for CTN as 0 because this statistic makes sense only for individual database files.

ZSHOW "G" can be used for a benchmark exercise. A process can make periodic commands to ZSHOW "G" and store the returned strings in a local variable - a fast storage mechanism in GT.M - for subsequent analysis.

Alternatively, since the $ZJOBEXAM() function by default performs a ZSHOW "*" which in turn automatically includes the "G" information code, invoking MUPIP INTRPT commands periodically on a particular process causes it to additionally record all global access statistics in the $ZJOBEXAM dump file.

ZSHOW "G" reports process private global access statistics only for regions whose corresponding segments have an access method of BG or MM in the global directory. For regions with other access methods, for example GT.CM GNP, which maps a region/segment to a remote database file, ZSHOW "G" does not report any process private statistics even though GT.M continuess to maintain aggregated statistics (across all processes) in the remote database file header.

If "L" occurs in the list, ZSHOW displays the current active M LOCKs and their corresponding LEVEL. On a active M lock, a LOCK+ increases LEVEL by 1 and LOCK- decreases the LEVEL by 1. GT.M increments MLG (M Locks Granted) by 1 for every LOCK successful LOCK acquiring action. GT.M treats LOCKs grouped into a single action by specifying them within parentheses as a single lock action. For example, LOCK (^SUCCESS1,^SUCCESS2) increments MLG by 1.

GT.M increment MLT (M Locks Timeout) by 1 for every failed (timeout) attempt to LOCK a resource.

Every user level lock request in turn translates to one or more calls to the database lock code (depending on the timeout and the number of lock names specified in the same lock command) which increments the LKS and/or LKF statistics of the ZSHOW "G" output appropriately.

When $PRINCIPAL input and output are different devices, ZSHOW "D" shows them as separate items identified as 0 for input and 0-out for output. ZSHOW "D" includes "TLS" in the second line of the output for an encrypted socket. ZSHOW "D" reports available information on both the local and remote sides of a TCP socket.

In UTF-8 mode, the ZSHOW command exhibits byte-oriented and display-oriented behavior as follows:

- ZSHOW targeted to a device (ZSHOW "*") aligns the output according to the numbers of display columns specified by the WIDTH deviceparameter.

- ZSHOW targeted to a local (ZSHOW "*":lcl) truncates data exceeding 2048KB at the last character that fully fits within the 2048KB limit.

- ZSHOW targeted to a global (ZSHOW "*":^CC) truncates data exceeding the maximum record size for the target global at the last character that fully fits within that record size.

- ZSHOW "L" displays the MLG and MLT M-lock statistics in one line just before displaying the LOCKs currently held by the process.

Examples of ZSHow

Example:

```
GTM>ZSHOW "db"
```

This command displays all devices with deviceparameters reflecting their current characteristics followed by any current ZBREAK locations with their corresponding actions.

Example:

```
GTM>ZSHOW "dbd"
```

This command displays the same output as the previous example.

Example:

```
GTM>ZSHOW "ax"
```

This command generates a run-time error.

Example:

```
LAB1   DO LAB2
       Quit
LAB2   Do LAB3
       Quit
LAB3   ZSHow
       Quit
```

Produces the results:

```
LAB3^RTN
LAB2^RTN
LAB1^RTN
```

Example:

```
GTM>ZSHOW "G"
```

For process that has access to two database files produces results like the following:

```
GLD:*,REG:*,SET:205,KIL:0,GET:1,DTA:0,ORD:0,ZPR:0,QRY:0,LKS:0,LKF:0,CTN:0,DRD:9,DWT:15,
NTW:203,NTR:4,NBW:212,NBR:414,NR0:0,NR1:0,NR2:0,NR3:0,TTW:1,TTR:0,TRB:0,TBW:2,TBR:6,
TR0:0,TR1:0,TR2:0,TR3:0,TR4:0,TC0:0,TC1:0,TC2:0,TC3:0,TC4:0,ZTR:0,DFL:0,DFS:0,JFL:0,JFS:0,JBB:0,JFB:0,JFW:0,JRL:0,JRP:0,
JRE:0,JRI:0,JRO:0,JEX:0,DEX:0,CAT:4,CFE:0,CFS:0,CFT:0,CQS:0,CQT:0,CYS:0,CYT:0,BTD:0,WFR:0,BUS:0,BTS:0,STG:0,KTG:0,ZTG:0,
DEXA:0,GLB:0,JNL:0,MLK:0,PRC:0,TRX:0,ZAD:0,JOPA:0,AFRA:0,BREA:0,MLBA:0,TRGA:0,WRL:0,PRG:0,WFL:0,WHE:0,INC:0
GLD:/home/gtmuser1/.fis-gtm/V5.4-002B_x86/g/mumps.gld,REG:DEFAULT,SET:205,KIL:0,GET:1,
DTA:0,ORD:0,ZPR:0,QRY:0,LKS:0,LKF:0,CTN:411,DRD:9,DWT:15,NTW:2
```

```
03,NTR:4,NBW:212,NBR:414,NR0:0,NR1:0,NR2:0,NR3:0,TTW:1,TTR:0,TRB:0,TBW:2,TBR:6,TR0:0,
TR1:0,TR2:0,TR3:0,TR4:0,TC0:0,TC1:0,TC2:0,TC3:0,TC4:0
GLD:/tmp/tst/test.gld,REG:DEFAULT,SET:205,KIL:0,GET:1,DTA:0,ORD:0,ZPR:0,QRY:0,LKS:0,LKF:0,
CTN:411,DRD:9,DWT:15,NTW:203,NTR:4,NBW:212,NBR:414,NR0:0,NR1:0,NR2:0,NR3:0,TTW:1,TTR:0,TRB:0,
TBW:2,TBR:6,TR0:0,TR1:0,TR2:0,TR3:0,TR4:0,TC0:0,TC1:0,TC2:0,TC3:0,TC4:0,ZTR:0,DFL:0,DFS:0,JFL:0,JFS:0,JBB:0,JFB:0,JFW:0,
JRL:0,JRP:0,JRE:0,JRI:0,JRO:0,JEX:0,DEX:0,CAT:4,CFE:0,CFS:0,CFT:0,CQS:0,CQT:0,CYS:0,CYT:0,BTD:0,WFR:0,BUS:0,BTS:0,STG:0,
KTG:0,ZTG:0,DEXA:0,GLB:0,JNL:0,MLK:0,PRC:0,TRX:0,ZAD:0,JOPA:0,AFRA:0,BREA:0,MLBA:0,TRGA:0,WRL:0,PRG:0,WFL:0,WHE:0,INC:0
```

Example:

```
GTM>ZSHOW "G"
```

Assuming that a GT.M process uses the global directory "/tmp/x1.gld" and opens two regions REG1 and REG2 corresponding to two database files, the above command produces results like the following:

```
GLD:*,REG:*,SET:0,KIL:0,GET:0,DTA:0,ORD:0,ZPR:0,QRY:0,LKS:0,LKF:0,CTN:0,DRD:0,DWT:0,NTW:0,
NTR:0,NBW:0,NBR:0,NR0:0,NR1:0,NR2:0,NR3:0,TTW:0,TTR:0,TRB:0,
TBW:0,TBR:0,TR0:0,TR1:0,TR2:0,TR3:0,TR4:0,TC0:0,TC1:0,TC2:0,TC3:0,TC4:0
GLD:/tmp/x1.gld,REG:REG1,SET:0,KIL:0,GET:0,DTA:0,ORD:0,ZPR:0,QRY:0,LKS:0,LKF:0,CTN:0,DRD:0,
DWT:0,NTW:0,NTR:0,NBW:0,NBR:0,NR0:0,NR1:0,NR2:0,NR3:0,TTW:0,
TTR:0,TRB:0,TBW:0,TBR:0,TR0:0,TR1:0,TR2:0,TR3:0,TR4:0,TC0:0,TC1:0,TC2:0,TC3:0,TC4:0,ZTR:0,DFL:0,DFS:0,JFL:0,JFS:0,JBB:0,
JFB:0,JFW:0,JRL:0,JRP:0,JRE:0,JRI:0,JRO:0,JEX:0,DEX:0,CAT:4,CFE:0,CFS:0,CFT:0,CQS:0,CQT:0,CYS:0,CYT:0,BTD:0,WFR:0,BUS:0,
BTS:0,STG:0,KTG:0,ZTG:0,DEXA:0,GLB:0,JNL:0,MLK:0,PRC:0,TRX:0,ZAD:0,JOPA:0,AFRA:0,BREA:0,MLBA:0,TRGA:0,WRL:0,PRG:0,WFL:0,
WHE:0,INC:0
GLD:/tmp/x1.gld,REG:REG2,SET:0,KIL:0,GET:0,DTA:0,ORD:0,ZPR:0,QRY:0,LKS:0,LKF:0,CTN:0,DRD:0,
DWT:0,NTW:0,NTR:0,NBW:0,NBR:0,NR0:0,NR1:0,NR2:0,NR3:0,TTW:0,
TTR:0,TRB:0,TBW:0,TBR:0,TR0:0,TR1:0,TR2:0,TR3:0,TR4:0,TC0:0,TC1:0,TC2:0,TC3:0,TC4:0,ZTR:0,DFL:0,DFS:0,JFL:0,JFS:0,JBB:0,
JFB:0,JFW:0,JRL:0,JRP:0,JRE:0,JRI:0,JRO:0,JEX:0,DEX:0,CAT:4,CFE:0,CFS:0,CFT:0,CQS:0,CQT:0,CYS:0,CYT:0,BTD:0,WFR:0,BUS:0,
BTS:0,STG:0,KTG:0,ZTG:0,DEXA:0,GLB:0,JNL:0,MLK:0,PRC:0,TRX:0,ZAD:0,JOPA:0,AFRA:0,BREA:0,MLBA:0,TRGA:0,WRL:0,PRG:0,WFL:0,
WHE:0,INC:0
```

Example:

```
GTM>ZSHOW "G":zgbl
```

This example redirects the output of ZSHOW "G" into a local variable zgbl:

```
zgbl("G",0)="GLD:*,REG:*,SET:0,KIL:0,GET:0,DTA:0,ORD:0,
ZPR:0,QRY:0,LKS:0,LKF:0,CTN:0,DRD:0,DWT:0,NTW:0,NTR:0,NBW:0,NBR:0,NR0:0,NR1:0,NR2:0,NR3:0,TTW:0,
TTR:0,TRB:0,TBW:0,TBR:0,TR0:0,TR1:0,TR2:0,TR3:0,TR4:0,TC0:0,TC1:0,TC2:0,TC3:0,TC4:0,ZTR:0,DFL:0,DFS:0,JFL:0,JFS:0,JBB:0,
JFB:0,JFW:0,JRL:0,JRP:0,JRE:0,JRI:0,JRO:0,JEX:0,DEX:0,CAT:4,CFE:0,CFS:0,CFT:0,CQS:0,CQT:0,CYS:0,CYT:0,BTD:0,WFR:0,BUS:0,
BTS:0,STG:0,KTG:0,ZTG:0,DEXA:0,GLB:0,JNL:0,MLK:0,PRC:0,TRX:0,ZAD:0,JOPA:0,AFRA:0,BREA:0,MLBA:0,TRGA:0,WRL:0,PRG:0,WFL:0,
WHE:0,INC:0"
zgbl("G",1)="GLD:/tmp/x1.gld,REG:REG1,SET:0,KIL:0,GET:0,DTA:0,ORD:0,ZPR:0,QRY:0,
LKS:0,LKF:0,CTN:0,DRD:0,DWT:0,NTW:0,NTR:0,NBW:0,NBR:0,NR0:0,NR1:0,NR2:0,
NR3:0,TTW:0,TTR:0,TRB:0,TBW:0,TBR:0,TR0:0,TR1:0,TR2:0,TR3:0,TR4:0,TC0:0,TC1:0,TC2:0,TC3:0,TC4:0,ZTR:0,DFL:0,DFS:0,JFL:0,
JFS:0,JBB:0,JFB:0,JFW:0,JRL:0,JRP:0,JRE:0,JRI:0,JRO:0,JEX:0,DEX:0,CAT:4,CFE:0,CFS:0,CFT:0,CQS:0,CQT:0,CYS:0,CYT:0,BTD:0,
WFR:0,BUS:0,BTS:0,STG:0,KTG:0,ZTG:0,DEXA:0,GLB:0,JNL:0,MLK:0,PRC:0,TRX:0,ZAD:0,JOPA:0,AFRA:0,BREA:0,MLBA:0,TRGA:0,WRL:0,
PRG:0,WFL:0,WHE:0,INC:0"
zgbl("G",2)="GLD:/tmp/x1.gld,REG:REG2,SET:0,KIL:0,GET:0,DTA:0,ORD:0,ZPR:0,QRY:0,LKS:0,
LKF:0,CTN:0,DRD:0,DWT:0,NTW:0,NTR:0,NBW:0,NBR:0,NR0:0,NR1:0,NR2:0,
NR3:0,TTW:0,TTR:0,TRB:0,TBW:0,TBR:0,TR0:0,TR1:0,TR2:0,TR3:0,TR4:0,TC0:0,TC1:0,TC2:0,TC3:0,TC4:0,ZTR:0,DFL:0,DFS:0,JFL:0,
JFS:0,JBB:0,JFB:0,JFW:0,JRL:0,JRP:0,JRE:0,JRI:0,JRO:0,JEX:0,DEX:0,CAT:4,CFE:0,CFS:0,CFT:0,CQS:0,CQT:0,CYS:0,CYT:0,BTD:0,
WFR:0,BUS:0,BTS:0,STG:0,KTG:0,ZTG:0,DEXA:0,GLB:0,JNL:0,MLK:0,PRC:0,TRX:0,ZAD:0,JOPA:0,AFRA:0,BREA:0,MLBA:0,TRGA:0,WRL:0,
PRG:0,WFL:0,WHE:0,INC:0"
```

Example:

```
GTM>LOCK ^FAIL:10
GTM>lock (^SUCCESS1,^SUCCESS2)
GTM>zshow "L"
MLG:1,MLT:1
LOCK ^SUCCESS1 LEVEL=1
```

```
LOCK ^SUCCESS2 LEVEL=1
```

This output shows that a process locked ^SUCCESS1 and ^SUCCESS2 and another the lock on ^FAIL failed due to time out.

Note that even though two lock resources ^SUCCESS1 and ^SUCCESS2 were specified in the LOCK command that succeeded, GT.M increments the MLG counter by only 1 because they are part of the same LOCK command. A ZSHOW "L":var by the same process (redirecting the output of ZSHOW into a local or global variable) would result in <var> holding the following contents.

```
var("L",0)="MLG:1,MLT:1"
var("L",1)="LOCK ^SUCCESS1 LEVEL=1"
var("L",2)="LOCK ^SUCCESS2 LEVEL=1"
```

Example:

```
GTM>ZSHOW "L":var
GTM>ZWRITE var
var("L",0)="MLG:1,MLT:1"
var("L",1)="LOCK ^SUCCESS1 LEVEL=1"
var("L",2)="LOCK ^SUCCESS2 LEVEL=1"
```

This example shows how ZSHOW "L" redirects it output into a local variable var.

Example:

Suppose a process runs LOCK (^SUCCESS1,^SUCCESS2) which succeeds and a LOCK +^FAIL:1 which times out due to another process holding that lock. A ZSHOW "L" at this point displays the following output.

Example:

```
GTM>ZSHOW "I"
$DEVICE=""
$ECODE=""
$ESTACK=0
$ETRAP=""
$HOROLOG="64813,13850"
$IO="/dev/pts/0"
$JOB=20264
$KEY=""
$PRINCIPAL="/dev/pts/0"
$QUIT=0
$REFERENCE=""
$STACK=0
$STORAGE=2147483647
$SYSTEM="47,gtm_sysid"
$TEST=1
$TLEVEL=0
$TRESTART=0
$X=0
$Y=20
$ZA=0
$ZALLOCSTOR=671584
$ZAUDIT=0
$ZB=""
$ZCHSET="M"
$ZCLOSE=0
$ZCMDLINE=""
```

```
$ZCOMPILE=""
$ZCSTATUS=0
$ZDATEFORM=0
$ZDIRECTORY="/path/to/the/current/directory"
$ZEDITOR=0
$ZEOF=0
$ZERROR="Unprocessed $ZERROR, see $ZSTATUS"
$ZGBLDIR="/path/to/the/global/directory/$gtmgbldir.gld"
$ZHOROLOG="64813,13850,790453,14400"
$ZICUVER=""
$ZININTERRUPT=0
$ZINTERRUPT="IF $ZJOBEXAM()"
$ZIO="/dev/pts/0"
$ZJOB=0
$ZKEY=""
$ZLEVEL=1
$ZMAXTPTIME=0
$ZMODE="INTERACTIVE"
$ZONLNRLBK=0
$ZPATNUMERIC="M"
$ZPIN="/dev/pts/0"
$ZPOSITION="+1^GTM$DMOD"
$ZPOUT="/dev/pts/0"
$ZPROMPT="GTM>"
$ZQUIT=0
$ZREALSTOR=694280
$ZRELDATE="20180614 00:33"
$ZROUTINES=". /usr/lib/fis-gtm/V6.3-007_x86_64
▶ /usr/lib/fis-gtm/V6.3-007_x86_64/plugin/o(/usr/lib/fis-gtm/V6.3-007_x86_64/plugin/r)"
$ZSOURCE=""
$ZSTATUS=""
$ZSTEP="B"
$ZSTRPLLIM=0
$ZSYSTEM=0
$ZTIMEOUT=-1
$ZTDATA=0
$ZTDELIM=""
$ZTEXIT=""
$ZTLEVEL=0
$ZTNAME=""
$ZTOLDVAL=""
$ZTRAP="B"
$ZTRIGGEROP=""
$ZTSLATE=""
$ZTUPDATE=""
$ZTVALUE=""
$ZTWORMHOLE=""
$ZUSEDSTOR=666047
$ZUT=1528962650791332
$ZVERSION="GT.M V6.3-007 Linux x86_64"
$ZYERROR=""
```

▼

This example displays the current value of all intrinsic special variables.

ZSHOW Destination Variables

ZSHOW may specify an unsubscripted or subscripted global or local variable name (glvn) into which ZSHOW places its output. If the argument does not include a global or local variable name, ZSHOW directs its output to the current device.

When ZSHOW directs its output to a variable, it adds two levels of descendants to that variable. The first level subscript contains a one-character string from the set of upper-case ZSHOW action codes, identifying the type of information. ZSHOW implicitly KILLs all descendants of the first level nodes. ZSHOW stores information elements at the second level using ascending integers, starting at 1.

When a ZSHOW "V" directs its output to a local variable (lvn), the result does not contain a copy of the descendants of the resulting "V" node.

Example:

```
GTM>Kill  Set b(1,"two")="test" ZSHow "v":a ZWRite
a("V",1)="b(1,""two"")=""test"""
b(1,"two")="test"
GTM>
```

This ZSHow stores all local variables in the local variable a. Note that ZSHOW does not replicate a("V") and a("V",1).

Example:

```
GTM>KILL  SET a(1,"D",3,5)="stuff",a(1,"X",2)="",a(1)=1
GTM>ZSHow "d":a(1)
GTM>ZWRite
a(1)=1
a(1,"D",1)="/dev/pts/1 OPEN TERMINAL NOPAST NOESCA NOREADS TYPE WIDTH=80 LENG=24
              EDIT "
a(1,"X",2)=""
GTM>
```

This ZSHOW stores the current open device information under a(1). Notice how the ZSHOW overlays the prior value of a(1,"D",3,5).

Example:

```
GTM>KILL ^ZSHOW

GTM>ZB -*,lab^rout ZSH "B":^ZSHOW
GTM>ZWRite ^ZSHOW
^ZSHOW("B",1)="LAB^ROUT"
GTM>
```

This ZSHOW stores the current ZBREAK information under the global variable ^ZSHOW.

Use of ZSHOW

Use ZSHOW as

- a debugging tool to display information on the environment.

- an error-handling tool to capture context information after an unpredictable error with output directed to a sequential file or a global.

- part of a context-switching mechanism in a server program that must manage multiple contexts.

- a development tool to determine the external call table entries available from the current process.

To minimize confusing data interactions, limit instances of directing ZSHOW output into variables holding other kinds of information and directing ZSHOW "V" output into local variables. For a comparison of ZSHOW "V" and ZWRITE, refer to "ZWRite" (page 209).

ZSTep

The ZSTEP command provides the ability to control GT.M execution. When a ZSTEP is issued from Direct Mode, execution continues to the beginning of the next target line and then GT.M XECUTEs the ZSTEP action. The keyword in the optional ZSTEP argument determines the class of eligible target lines.

The format of the ZSTEP command is:

```
ZST[EP][:tvexpr] [keyword[:expr]][,...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The optional keyword specifies the nature of the step; the keywords are INTO, OVER, and OUTOF.

- A ZSTEP with no argument performs the default action OVER; in this case, at least two (2) spaces must follow the ZSTEP to separate it from the next command on the line, which will be ignored.

- The optional expression specifies GT.M code to XECUTE when the ZSTEP arrives at its destination.

- If the ZSTEP argument does not contain an expression argument, ZSTEP defaults the action to the value of $ZSTEP, which defaults to "BREAK."

> **Note**
>
> The ZSTEP argument keywords are not expressions and ZSTEP does not accept argument indirection.

In Direct Mode, ZSTEP performs an implicit ZCONTINUE and therefore GT.M ignores all commands on the Direct Mode command line after the ZSTEP.

The keyword arguments define the class of lines where ZSTEP next pauses execution to XECUTE the ZSTEP action. When a ZSTEP command has multiple arguments, it ignores all arguments except the last.

ZSTEP Into

ZSTEP INTO pauses at the beginning of the next line, regardless of transfers of control. When the ZSTEPed line invokes another routine or a subroutine in the current routine, ZSTEP INTO pauses at the first line of code associated with the new GT.M stack level.

ZSTep OUtof

ZSTEP OUTOF pauses at the beginning of the next line executed after an explicit or implicit QUIT from the current GT.M invocation stack level. A ZSTEP OUTOF does not pause at lines associated with the current GT.M stack level or with levels invoked from the current level.

ZSTep OVer

ZSTEP OVER pauses at the beginning of the next line in the code associated with either the current GT.M stack level or a previous GT.M stack level if the ZSTEPed line contains an explicit or implicit QUIT from the current level. A ZSTEP OVER does not pause at lines invoked from the current line by DOs, XECUTEs or extrinsics.

ZSTEP Actions

The optional action parameter of a ZSTEP must contain an expression evaluating to valid GT.M code. By default, ZSTEP uses the value of $ZSTEP, which defaults to "B" ("BREAK"), and enters Direct Mode. When a ZSTEP command specifies an action, the process does not enter Direct Mode unless the action explicitly includes a BREAK command.

ZSTEP Interactions

ZSTEP currently interacts with certain other elements in the GT.M environment.

- When there is a <CTRL-C> and CTRAP=$CHAR(3), the CTRAP has priority over ZSTEP.

- When there is a <CTRL-n> and CTRAP=$CHAR(n) where n is 0-31, but not 3, GT.M recognizes the event at a READ and the recognition does not interact directly with ZSTEP, but competes "fairly" with other deferred events.

Use of ZSTEP

Use ZSTEP to incrementally execute a routine or series of routines. Execute any GT.M command from Direct Mode at any ZSTEP pause. To resume normal execution, use ZCONTINUE.

Note that ZSTEP arguments are keywords rather than expressions. They do not allow indirection, and argument lists have no utility.

ZSTEP actions that include commands followed by a BREAK perform some action before entering Direct Mode. ZSTEP actions that do not include a BREAK perform the command action and continue execution. Use ZSTEP actions that issue conditional BREAKs and subsequent ZSTEPs to do such things as test for changes in the value of a variable.

Examples of ZSTEP

Example:

```
GTM>ZSTEP INTO:"W ! ZP @$ZPOS W !"
```

This ZSTEP resumes execution of the current routine. At the beginning of the next line executed, the ZSTEP action ZPRINTs the source code for that line. Because the specified action does not contain a BREAK command, execution continues to the next line and all subsequent lines in the program flow.

Example:

```
GTM>Set curx=$get(x),zact="ZSTEP:$get(curx)=$get(x) INTO:zact Break:$get(curx)'=$get(x)"
GTM>ZSTEP INTO:zact
```

This sequence uses ZSTEP to invoke Direct Mode at the beginning of the first line after the line that alters the value of x.

ZSYstem

The ZSYSTEM command creates a child of the current process.

The format of the ZSYSTEM command is:

```
ZSY[STEM][:tvexpr] [expr][,...]]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The optional expression specifies the command passed to the shell; after processing the command, the shell returns control to GT.M. The maximum length for the optional expression is 4K bytes.

- An indirection operator and an expression atom evaluating to a list of one or more ZSYSTEM arguments form a legal argument for a ZSYSTEM.

The ZSYSTEM command creates a new process and passes its argument to a shell for execution. The new process executes in the same directory as the initiating process using the shell specified by the SHELL environment variable, or if that is not defined, the default shell (typically Bourne). The new process has the same operating system environment, such as environment variables and input/output devices, as the initiating process. The initiating process pauses until the new process completes before continuing execution. After control returns to GT.M, $ZSYSTEM contains the return status of the forked process.

A ZSYSTEM with a null argument creates a shell with the standard input and output devices. When the shell exits, control is returned to GT.M. For an interactive process, both stdin and stdout are generally the user's terminal, in which case the shell prompts for input until provided with an exit command. A ZSYSTEM with no arguments must be followed by two (2) spaces before any following command on the same line and is equivalent to a ZSYSTEM with a single null string argument.

If a ZSYSTEM command has multiple arguments, it starts a new process for each argument, one at a time. ZSYSTEM waits for one process to complete before starting the next one.

A ZSYSTEM command within a TP transaction, violates the property of Isolation. Consequently because of the way that GT.M implements transaction processing, a ZSYSTEM within a transaction may suffer an indefinite number of restarts ("live lock").

An indirection operator and an expression atom evaluating to a list of one or more ZSYSTEM arguments form a legal argument for a ZSYSTEM.

> **Note**
>
> PIPE devices are frequently a better alternative to ZSYSTEM commands as they have timeouts, can perform controlled co-processing, easily return more information and are more efficient where you need multiple operations.

Examples of ZSYSTEM

Example:

```
GTM>zsystem "ls *.m"
```

This uses ZSYSTEM to fork a process that then performs the ls command with *.m as an argument to ls. Once the command completes, the forked process terminates.

```
GTM>zsystem "echo 'hello world'"
hello world
```

This examples show a possible use of quoting with ZSYSTEM to run the echo command.

Example:

```
GTM>zsystem
$
```

This ZSYSTEM has no argument so the forked process prompts for input.

ZTCommit

The ZTCOMMIT command marks the end of a logical transaction within a GT.M program. ZTCOMMIT used with ZTSTART "fences" transactions (that is, marks the end and beginning). Fencing transactions allows the MUPIP JOURNAL facility to prevent incomplete application transactions consisting of multiple global updates from affecting the database during a database recovery. FIS strongly recommends the use of the M transaction processing commands such as TSTART and TCOMMIT rather than ZTSTART and ZTCOMMIT. FIS no longer tests the deprecated ZTSTART / ZTCOMMIT functionally.

The format of the ZTCOMMIT command is:

```
ZTC[OMMIT][:tvexpr] [intexpr]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The optional integer expression specifies the number of currently open ZTSTARTs for the ZTCOMMIT to close.

- A ZTCOMMIT with no argument closes one ZTSTART; in this case, at least two (2) spaces must follow the command to separate it from the next command on the line; with an argument of 0, ZTCOMMIT closes all open ZTSTARTs.

- When an application requires sub-transactions, it may nest ZTSTARTs and ZTCOMMITs to a maximum depth of 255. However, a ZTCOMMIT must "close" the outer-most ZTSTART before journaling accepts any part of the "transaction" as complete.

- An indirection operator and an expression atom evaluating to a list of one or more ZTCOMMIT arguments form a legal argument for a ZTCOMMIT.
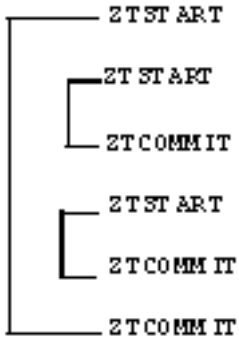
Examples of ZTCOMMIT

Example:

```
GTM>ZTCOMMIT 0
```

This ZTCOMMIT issued from Direct Mode would close any open ZTSTARTs.

Example:

This shows a transaction with two independent nested sub-transactions. For additional examples, refer to the ZTSTART examples.

ZTRigger

Invokes all triggers with signatures matching the global variable name and the command type of ZTR[IGGER]. The format of the ZTRIGGER command is:

```
ZTR[IGGER] gvn
```

- ZTRIGGER allows an application to invoke triggers without a specific global update.

- ZTRIGGER actions are Atomic whether they are executed inside or outside a TP transaction; but inside TP they remain process-private until the TCOMMIT of the full transaction. ZTRIGGER might be associated with a series of updates grouped into a TP transaction or to perform an implicit transaction without a TSTART/TCOMMIT.

- A ZTRIGGER operation sets $ZTRIGGEROP to ZTR.

Example:

```
GTM>write $ztrigger("S")
;trigger name: C#1#  cycle: 1
+^C -commands=ZTR -xecute="write ""ZTR trigger invoked"""
1
GTM>ztrigger ^C
ZTR trigger invoked
GTM>
```

ZTStart

The ZTSTART command marks the beginning of a logical transaction within a GT.M program. ZTSTART and ZTCOMMIT "fence" transactions (that is, mark the beginning and end). Fenced transactions prevent the MUPIP JOURNAL facility from recovering incomplete transactions. All ZTSTARTs must be matched with ZTCOMMITs before the journal processing facility recognizes the transaction as complete. FIS strongly recommends the use of the M transaction processing commands such as TSTART and TCOMMIT rather than ZTSTART and ZTCOMMIT. FIS no longer tests the deprecated ZTSTART / ZTCOMMIT functionally.

The format of the ZTSTART command is:

```
ZTS[TART][:tvexpr]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- Because ZTSTART has no argument, at least two (2) spaces must follow the command to separate it from the next command on the line.

For more information on Journaling and transaction fencing, refer to the "GT.M Journaling" chapter in the **GT.M Administration and Operations Guide**.

ZWIthdraw

The ZWITHDRAW command KILLs the data value for a variable name without affecting the nodes descended from that node.

The format of the ZWITHDRAW command is:

```
ZWI[THDRAW][:tvexpr] glvn
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The global or local variable name identifies the variable for which ZWITHDRAW removes the data value.

- An indirection operator and an expression atom evaluating to a list of one or more ZWITHDRAW arguments form a legal argument for a ZWITHDRAW.

ZWITHDRAW provides a tool to quickly restore a node to a state where it has descendants and no value-- that is, where $DATA for that node will have a value of 10 -- for the case where such a state has some control meaning. GT.M also provides the ZKILL command, with functionality identical to ZWITHDRAW.

Examples of ZWITHDRAW

Example:

```
Kill A
Set A="A",A(1)=1,A(1,1)=1
WRite $Data(A(1)),!
ZWIthdraw A(1)
WRite $D(A(1)),!
ZWRite A
Quit
```

produces the result:

```
11
10
A="A"
A(1,1)=1
```

This sets up local variables A and A(1) and A(1,1). It then deletes the data for A(1) with ZWITHDRAW. The ZWRITE command shows ZWITHDRAW KILLed A(1) but left A and A(1,1).

ZWRite

The ZWRITE command displays the current value of one or more local , alias variables, ISVs, or global variables. ZWRITE formats its output so that each item in the display forms a valid argument to a SET @ command. This means ZWRITE encloses string values in quotes and represents non-graphic (control) characters in $CHAR() syntax.

The format of the ZWRITE command is:

```
ZWR[ITE][:tvexpr] [zwrglvn[,...]]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The optional global or local variable name specifies the variable for ZWRITE to display.

- ZWRITE accepts several alternative syntaxes in place of subscripts; ZWRITE also accepts arguments specifying naked references to globals. Because ZWRITE is primarily a debugging tool, ZWRITE does not affect the naked indicator.

- ZWRITE accepts null subscripts in its arguments, when these are allowed, and reports array nodes that have null subscripts.

- A ZWRITE with no arguments displays all the currently available local variables; in this case, at least two (2) spaces must follow the command to separate it from the next command on the line.

- If the specified global or local variable name is unsubscripted, ZWRITE displays the unsubscripted variable and all subscripted descendants.

- If an asterisk (*) appears in the space normally occupied by the last subscript in a subscripted variable name, ZWRITE displays all variable nodes descended from the previously specified subscripts.

- ZWRITE accepts GT.M pattern-match syntax in place of both variable names and subscripts.

- ZWRITE <name>(), where <name> is a local or a global is treated as a synonym for ZWRITE <name>.

- A colon acts as a range operator for subscript values; ZWRITE displays all subscripts of the variable starting with the value on the left side of the colon and ending with the value on the right side of the colon; if the range delimiter has no left-hand value, or has the empty string as the left-hand value, the display begins at the first subscript; if the range delimiter has no right-hand value or has the empty string as the right-hand value, the display ends at the last subscript at that level; if the range delimiter has no values or empty strings on either side, ZWRITE displays all subscripts at that level; an empty subscript level also displays all subscripts at that level.

- An indirection operator and an expression atom evaluating to a list of one or more ZWRITE arguments form a legal argument for a ZWRITE.

- Long ZWRITE format records can be loaded.

- ZWRITE as applied to local variables and ZSHOW "V" are conceptually similar, with two differences:

  - ZWRITE allows the use of patterns to specify variables and subscripts to display whereas ZSHOW "V" applies to all local variables.

  - ZSHOW "V" optionally allows the output to be directed to a global or local variable, whereas ZWRITE always directs its output to the current output device.

- ZWRITE of a global variable:

- Maintains $REFERENCE to the last node it successfully processed.

- Does not recognize extended reference syntax, but if given a naked reference syntax immediately following an extended reference does use the global directory specified by that prior reference.

ZWRITE Format for Alias Variables

ZWRITE and ZSHOW "V" dump the values of alias variables, alias container variables, and the associated data as described below, in ZWRITE format. In the ZWRITE format, the contents of an array are displayed with the name associated with that array that appears first in the lexical ordering of names. GT.M displays both the unsubscripted and subscripted nodes and values, appending a notational space-semicolon-asterisk (";*") sequence to the unsubscripted value, if any. The ZWRITE format output can be read into a GT.M process with the commands Read x and Set @x (where x is any name) executed in a loop. ";*" acts as a comment ignored by the SET command. In the following example, since A and C are aliases associated with the same array, the nodes of that array are output with A, which occurs lexically before C, even though the values were assigned to C:

```
GTM>Set C=1,C("Malvern")="Wales",*A=C,*B(-3.14)=C

GTM>ZSHow "V" ; ZWRite would produce the same output
A=1 ;*
A("Malvern")="Wales"
*B(-3.14)=A
*C=A
GTM>ZWRite C ; Only one is name associated with the array on this ZWRite command
C=1 ;*
C("Malvern")="Wales"
GTM>
```

Continuing the example, if the variables selected for the ZWRITE command do not include any of the the associated alias variables, the output shows only the reference, not the data:

```
GTM>ZWRITE B ; B only has a container
*B(-3.14)=A
GTM>
```

When ZWRITE / ZSHOW "V" encounters an alias container for an array with no current alias variable, it uses a name $ZWRTACn as the made-up name of an alias variable for that array, where n is an arbitrary but unique integer. The SET command recognizes this special name, thus enabling the output of a ZWRITE / ZSHOW "V" to be used to recreate alias containers without associated alias variables. Continuing the above example:

```
GTM>Kill *A,*C ; Delete alias variables and associations, leaving only the container

GTM>ZWRite
$ZWRTAC=""
*B(-3.14)=$ZWRTAC1
$ZWRTAC1=3 ;*
$ZWRTAC1("Malvern")="Wales"
$ZWRTAC=""
GTM>
```

ZWRITE produces $ZWRTACn names to serve as data cell anchors which SET @ accepts as valid set left targets. $ZWRTACn names are created and destroyed when using ZWRITE output to drive restoration of a previously captured variable state. Except for their appearance in ZWRITE output and as left-hand side SET @ targets, they have no function. Other than SET, no other commands can use $ZWRTAC* in their syntax. Although $ZWRTACn superficially looks like an intrinsic special variable (ISV), they are not ISVs. $ZWRTACn with no subscripts can serve as the target (left side of the equals-sign) of a SET *

command. SET $ZWRTAC (no trailing integer) deletes all data cell associations with the $ZWRTAC prefixed aliases. GT.M only recognizes the upper-case unabbreviated name and prefix $ZWRTAC.

When ZWRITE displays values for an alias variable, it appends a " ;*" to the name which visually tags the alias without interfering with use of ZWRITE output as arguments to a SET @. ZWRITE can only identify alias variables when at least two aliases for the same data match its argument. When ZWRITE encounters an alias container variable with no current associated alias, it uses the ZWRTAC mechanism to expose the data; SET @ can restore data exposed with the ZWRTAC mechanism.

> ⚠️ **Caution**
>
> FIS strongly recommends that you should not create or manipulate your own $ZWRTACn "variables". They are not part of the supported functionality for implementing alias variables and containers, but are rather a part of the underlying implementation that is visible to you, the GT.M user. FIS can arbitrarily, for its own convenience change the use of $ZWRTAC in GT.M at any time. They are only documented here since you may see them in the output of ZWRITE and ZSHOW "V".

Examples of ZWRITE

Example:

```
GTM>ZWRITE ^?1"%"2U(0:":",)
```

This command displays the descendants of all subscripts between 0 and ":" of all global names starting with a "%" and having two upper case letters -- for example, "%AB".

Example:

```
GTM>ZWRITE A(,:,3)
```

This command displays all of the third level nodes with a subscript of 3 for local variable A.

Example:

```
ZWRITE ?1"A".E(?1"X"3N)
```

This displays data for any local variables starting with "A", optionally followed by any characters, and having any subscripts starting with "X" followed by three numerics.

Example:

```
GTM>Set A=1,*B=A ; Create an array and an alias association

GTM>ZWRite ; Show that the array and association exist
A=1 ;*
*B=A
```

Chapter 7. Functions

Revision History		
Revision V7.1-004	27 June 2024	• In "Argument Keywords of $VIEW()" (page 244), GTM-10588 - Removed the third value of "SPSIZE",\nsince the reserved space we used to have,\nto reduce the active memory usage no longer exists.
		• In "Examples of $VIEW()" (page 248), Added WFR,BUS,BTS,STG,KTG,ZTG,DEXA,GLB,JNL,MLK,PF to the $view("GVSTAT","DEFAULT")
		• In "Example of $ZBITAND()" (page 254), In the example, add command to set BITSTRINGA
		• In "$ZBITGET()" (page 255), fix typo
		• In "$ZBITNOT()" (page 256), fix typo
		• In "$ZBITXOR()" (page 258), Fix typo zbitor->zbitxor in the example
		• In "$ZCOLlate()" (page 259), Shortest form of $zcollate() is $zcol()
		• In "$ZFind()" (page 266), fixed a typo
Revision V7.1-003	23 November 2023	• In "$ZPEEK()" (page 275), Revise the Note on selection among multiple replication journal pools
Revision V7.1-001	26 June 2023	• In "Argument Keywords of $VIEW()" (page 244), Fix typo for JNLPOOL
Revision V7.1-000	04 April 2023	• In "Argument Keywords of $VIEW()" (page 244), add $VIEW("FULL_BOOLEAN") description string for new EXTENDED_BOOLEAN option.
		• In "$ZPEEK()" (page 275), add a note about $ZPEEK of jnlpool fields and multiple replication instances
Revision V7.0-005	02 December 2022	• In "Examples of $ZAUDitlog()" (page 253), Fix a typo in $zauditlog example description.
		• In "$ZAUditlog" (page 252), add $ZAUDITLOG()
		• In "$ZSOCKET()" (page 278), corrected typo
Revision V7.0-004	20 September 2022	• In "$ZSOCKET()" (page 278), add new $ZSOCKET() keywords for getsockopt() items. Disabled support for pre-TLSv1.2

Revision V7.0-003	24 June 2022	• In "Argument Keywords of $VIEW()" (page 244), add new $VIEW() keyword "DEVICE"
		• In "Examples of $VIEW()" (page 248), add new $VIEW() keyword "DEVICE"
		• In "$ZSOCKET()" (page 278), add new $ZSOCKET() keywords for getsockopt() items. Disabled support for pre-TLSv1.2
Revision V7.0-002	23 March 2022	• In "$ZJOBEXAM()" (page 268), add the second optional argument
		• In "$ZDATE Format Specification Elements" (page 263), qdd 'YYYY' as another option (after 'YEAR') for getting a four digit year.
		• In "$ZGetjpi()" (page 267), fixed a typo
Revision V7.0-001	24 November 2021	• In "$ZSOCKET()" (page 278), add an entry for the BLOCKING keyword.
Revision V6.3-012	08 April 2020	• In "Examples of $INCREMENT()" (page 224), fix case of variable name
Revision V6.3-010	31 October 2019	• In "Examples of $ZATRANSFORM()" (page 252), fix typo
		• In "$ZATRansform" (page 251), add changes for 2 (two) and -2 (minus two) as optional third arguments.
		• In "$ZCOnvert()" (page 260), remove the statement that character code "T" does not work in M mode; correct an example.
Revision V6.3-007	04 February 2019	• In "Argument Keywords of $VIEW()" (page 244), adjust probecrit output for V6.3-007
		• In "$ZLength()" (page 270), separate the last bullet points into two parts.
Revision V6.3-006	26 October 2018	• In "$ASCII()" (page 215), UTF-8 mode tweaks.
		• In "$Char()" (page 216), UTF-8 mode tweaks.
		• In "Examples of $ASCII()" (page 216), UTF-8 mode tweaks.
		• In "Examples of $CHAR()" (page 217), UTF-8 mode tweaks
		• In "Examples of $ZWIDTH()" (page 289), UTF-8 tweaks and minor corrections.
		• In "$Extract()" (page 218), UTF-8 mode tweaks.
		• In "$ZCOnvert()" (page 260), minor corrections.

		• In "$ZSUBstr()" (page 284), UTF-8 mode tweaks.
		• In "$ZWidth()" (page 289), minor corrections.
Revision V6.3-005	29 June 2018	• In "Argument Keywords of $VIEW()" (page 244), update the description of STATSHARE argument keyword.
		• In "$ZSOCKET()" (page 278), add note on SSLv3 default
		• In "$ZTRNLNM()" (page 288), state that expr6 is optional and add an example
Revision V6.3-004	23 March 2018	• In "Argument Keywords of $VIEW()" (page 244), add information about the STATSHARE keyword; improve the description of GVSTAT
Revision V6.3-003	12 December 2017	• In "Argument Keywords of $VIEW()" (page 244), fix format for gvstats
		• In "$ZDATA()" (page 261), typo "reflects" -> "reflect"
Revision V6.3-001	20 March 2017	• In "Argument Keywords of $VIEW()" (page 244), Update description of $VIEW("SPSIZE") to include two additional values returned.
		• In "Examples of $VIEW()" (page 248), updated gvstats output
		• In "$ZATRansform" (page 251), added the description of $ZATRANSFORM().
		• In "$ZCOLlate()" (page 259), add the description of $ZCOLLATE().
		• In "$ZPIece()" (page 273), removed spurious characters from an example;corrected the abbreviation for $ZPIECE()
		• In "$ZSIGPROC()" (page 283), removed Tru64.
		• In "$ZSOCKET()" (page 278), added the ALL, SESSION, OPTIONS, and TLS keywords
		• In "$ZWRite()" (page 290), added information about the second argument and information about string inflation going to zwrite format. Modified description to use "argument" as a user found the use of expression unclear.
Revision V6.2-001	27 February 2015	• Updated "$ZTRIgger()" (page 286) for V6.2-001 multi-line XECUTE string enhancements, $ZTRIGGER() return status, specified that $ZTRIGGER() always operates within a TP transaction even if it needs to

		implicitly create one, and added a note on the support of triggers in spanning regions.
		• Added a new section called "$ZSOCKET()" (page 278).
		• In "$ZPEEK()" (page 275), specified that $ZPEEK() function generates an UNDEF error when VIEW UNDEF is not set and format parameter is specified but is undefined and added the JNL[REG]:region and JBF[REG]:region keyword first arguments.
		• In "Argument Keywords of $VIEW()" (page 244), added "RTNCHECKSUM" as a new ARG1.
Revision V6.1-000	28 August 2014	• In "$ZGetjpi()" (page 267), corrected the descriptions of CSTIME and CUTIME keywords.
Revision V6.0-003	24 February 2014	• Added the description of "$ZPEEK()" (page 275).
Revision V6.0-001	21 March 2013	• Added the descriptions of "$ZWRite()" (page 290) and "$ZGetjpi()" (page 267).
		• Added information about TRIGGER and ZINTR return values for $STACK(lvl) in "$STack()" (page 238).

This chapter describes M language Intrinsic Functions implemented in GT.M. Traditional string processing functions have parallel functions that start with the letter "z". The parallel functions extend the byte-oriented functionality of their counterparts to UTF-8 mode. They are helpful when applications need to process binary data including blobs, binary byte streams, bit-masks, and so on.

Other functions that start with the letter "z" and do not have counterparts implement new functionality and are GT.M additions to the ANSI standard Intrinsic Functions. The M standard specifies standard abbreviations for Intrinsic Functions and rejects any non-standard abbreviations.

M Intrinsic Functions start with a single dollar sign ($) and have one or more arguments enclosed in parentheses () and separated by commas (,). These functions provide expression results by performing actions that are impossible or difficult to perform using M commands.

$ASCII()

Returns the integer ASCII code for a character in the given string. For a mumps process started in UTF-8 mode, $ASCII() returns the integer Unicode® UTF-8 code-point value of a character in the given string.

The format for the $ASCII function is:

```
$A[SCII](expr[,intexpr])
```

- The expression is the source string from which $ASCII() extracts the character it decodes.

- intexpr contains the position within the expression of the character that $ASCII() decodes. If intexpr is missing, $ASCII() returns a result based on the first character position.

- If intexpr evaluates to before the beginning or after the end of the expression, $ASCII() returns a value of negative one (-1).

$ASCII() provides a means of examining non-graphic characters in a string. When used with $CHAR(), $ASCII() also provides a means to perform arithmetic operations on the codes associated with characters.

$ZASCII() is the parallel function of $ASCII(). $ZASCII() interprets the string argument as a sequence of bytes (rather than a sequence of characters) and can perform all byte-oriented $ASCII() operations. For more information, refer to "$ZAscii()" (page 250).

Examples of $ASCII()

Example:

```
GTM>For i=0:1:3 Write !,$Ascii("Hi",i)
-1
72
73
-1
GTM>
```

This loop displays the result of $ASCII() specifying a character position before, first and second positions, and after the string.

Example:

```
GTM>Write $ZCHSET
UTF-8
GTM>Write $Ascii("主")
20027
GTM>Write $$FUNC^%DH("20027")
00004E3B
```

In this example, 20027 is the integer equivalent of the hexadecimal value 4E3B. U+4E3B is a character in the CJK Ideograph block of the Unicode® standard.

$Char()

Returns a string of one or more characters corresponding to integer ASCII codes specified in its argument(s). For a process started in UTF-8 mode, $CHAR() returns a string composed of characters represented by the integer equivalents of the Unicode® code-points specified in its argument(s).

The format for the $CHAR function is:

```
$C[HAR](intexpr[,...])
```

- The integer expression(s) specify the codes of the character(s) $CHAR() returns.

- The M standard does not restrict the number of arguments to $CHAR(). However, GT.M does limit the number of arguments to a maximum of 254. $CHAR() provides a means of producing non-graphic characters, as such characters cannot appear directly within an M string literal. When used with $ASCII(), $CHAR() can also perform arithmetic operations on the codes associated with characters.

- With VIEW "BADCHAR" enabled, $CHAR() produces a run-time error if any expression evaluates to a code-point value that is not a UTF-8 character. GT.M determines from ICU which characters are illegal.

- $ZCHAR() is the parallel function of $CHAR(). $ZCHAR() returns a sequence of bytes (rather than a sequence of characters) and can perform all byte-oriented $CHAR() operations. For more information, refer to "$ZCHar()" (page 259).

Examples of $CHAR()

Example:

```
GTM>write $char(77,85,77,80,83,7)
MUMPS
GTM>
```

This example uses $CHAR() to WRITE the word MUMPS and signal the terminal bell.

Example:

```
set nam=$extract(nam,1,$length(nam)-1)_$char($ascii(nam,$length(nam))-1)
```

This example uses $CHAR() and $ASCII() to set the variable nam to a value that immediately precedes its previous value in the set of strings of the same length as nam.

Example:

```
GTM>write $zchset
UTF-8
GTM>write $char(20027)
主
GTM>write $char(65)
A
```

In the above example, the integer value 20027 is the Unicode® character "主" in the CJK Ideograph block. Note that the output of the $CHAR() function for values of integer expression(s) from 0 through 127 does not vary with choice of the character encoding scheme. This is because 7-bit ASCII is a proper subset of UTF-8 character encoding scheme. The representation of characters returned by the $CHAR() function for values 128 through 255 differ for each character encoding scheme.

$Data()

Returns an integer code describing the value and descendent status of a local or global variable.

The format for the $DATA function is:

```
$D[ATA](glvn)
```

- The subscripted or unsubscripted global or local variable name specifies the target node.

- If the variable is undefined, $DATA() returns 0.

- If the variable has a value but no descendants, $DATA() returns 1.

- If the variable has descendants but no value, $DATA() returns 10.

- If the variable has a value and descendants, $DATA() returns 11.

- $ZDATA() extends $DATA() to reflects the current alias state of the lvn or name argument to identify alias and alias container variables. For more information, refer to "$ZDATA()" (page 261).

The following table summarizes $DATA() return values.

$DATA() Results		
VALUE		
	DESCENDANTS (NO)	DESCENDANTS (YES)
NO	0	10
YES	1	11

$DATA() return values can also be understood as a pair of truth-values where the left describes descendants and the right describes data 1 and where M suppresses any leading zero (representing no descendants).

Examples of $DATA()

Example:

```
GTM>Kill  Write $Data(a)
0
GTM>Set a(1)=1 Write $Data(a(1))
1
GTM>Write $Data(a)
10
GTM>Set a=0 Write $Data(a)
11
GTM>
```

This uses $DATA to display all possible $DATA() results.

Example:

```
lock ^ACCT(0)
if '$data(^ACCT(0)) set ^ACCT(0)=0
set (ACCT,^ACCT(0))=^ACCT(0)+1
lock
```

This uses $DATA() to determine whether a global node requires initialization.

Example:

```
for  set cus=$O(^cus(cus)) quit:cus=""  if $data(^(cus))>1 do WORK
```

This uses $DATA() to determine whether a global node has descendants and requires additional processing.

$Extract()

Returns a substring of a given string.

The format for the $EXTRACT function is:

```
$E[XTRACT](expr[,intexpr1[,intexpr2]])
```

- The expression specifies a string from which $EXTRACT() derives a substring.

- The first optional integer expression (second argument) specifies the starting character position in the string. If the starting position is beyond the end of the expression, $EXTRACT() returns an empty string. If the starting position is zero (0) or

negative, $EXTRACT() starts at the first character; if this argument is omitted, $EXTRACT() returns the first character of the expression. $EXTRACT() numbers character positions starting at one (1) (that is, the first character of a string is at position one (1)).

- The second optional integer expression (third argument) specifies the ending character position for the result. If the ending position is beyond the end of the expression, $EXTRACT() stops with the last character of the expression. If the ending position precedes the starting position, $EXTRACT() returns an empty string. If this argument is omitted, $EXTRACT() returns one character at most.

$EXTRACT() provides a tool for manipulating strings based on character positions.

For a mumps process started in UTF-mode, $EXTRACT interprets the string arguments as UTF-8 encoded. With VIEW "BADCHAR" enabled, $EXTRACT() produces a run-time error when it encounters a character in the reserved range of the Unicode® standard, but it does not process the characters that fall after the span specified by the arguments. The parallel function of $EXTRACT() is $ZEXTRACT(). Use $ZEXTRACT() for byte-oriented operations. For more information, refer to "$ZExtract()" (page 265).

$EXTRACT() can be used on the left-hand side of the equal sign (=) of a SET command to set a substring of a string. This construct permits easy maintenance of individual pieces within a string. It can also be used to right justify a value padded with blank characters. For more information on SET $EXTRACT(), refer to "Set" (page 144) in the Commands chapter.

Examples of $EXTRACT()

Example:

```
GTM>for i=0:1:3 write !,$extract("HI",i),"<"
<
H<
I<
<
GTM>
```

This loop displays the result of $EXTRACT(), specifying no ending character position and a beginning character position "before" first and second positions, and "after" the string.

Example:

```
GTM>For i=0:1:3 write !,$extract("HI",1,i),"<"
<
H<
HI<
HI<
GTM>
```

This loop displays the result of $EXTRACT() specifying a beginning character position of 1 and an ending character position "before, " first and second positions, and "after" the string.

Example:

```
GTM>zprint ^trim
trim(x)
    new i,j
    for i=1:1:$length(x) quit:" "'=$extract(x,i)
    for j=$length(x):-1:1 quit:" "'=$extract(x,j)
    quit $extract(x,i,j)
```

```
GTM>set str=" MUMPS "
GTM>write $length(str)
7
GTM>write $length($$^trim(str))
5
GTM>
```

This extrinsic function uses $EXTRACT() to remove extra leading and trailing spaces from its argument.

$Find()

Returns an integer character position that locates the occurrence of a substring within a string.

The format for the $FIND function is:

```
$F[IND](expr1,expr2[,intexpr])
```

- The first expression specifies the string within which $FIND() searches for the substring.

- The second expression specifies the substring for which $FIND() searches.

- The optional integer expression identifies the starting position for the $FIND() search. If this argument is missing, zero (0), or negative, $FIND() begins its search in the first position of the string.

- If $FIND() locates the substring, it returns the position after the last character of the substring. If the end of the substring coincides with the end of the string (expr1), it returns an integer equal to the length of the string plus one ($L(expr1)+1).

- If $FIND() does not locate the substring, it returns zero (0).

- For a process started in UTF-8 mode, $FIND() interprets the string arguments as UTF-8 encoded. With VIEW "BADCHAR" enabled, $FIND() produces a run-time error when it encounters a malformed character, but it does not process the characters that fall after the span specified by the arguments.

- $ZFIND() is the Z equivalent function $FIND(). Irrespective of the settings of VIEW "BADCHAR" and $ZCHSET, $ZFIND() interprets argument as a sequence of bytes (rather than a sequence of characters) and can perform byte-oriented $FIND() operations.For more information, refer to "$ZFind()" (page 266).

$FIND() provides a tool to locate substrings. The ([) operator and the two-argument $LENGTH() are other tools that provide related functionality.

Examples of $FIND()

Example:

```
GTM>write $find("HIFI","I")
3
GTM>
```

This example uses $FIND() to WRITE the position of the first occurrence of the character "I." The return of 3 gives the position after the "found" substring.

Example:

```
GTM>write $find("HIFI","I",3)
5
```

```
GTM>
```

This example uses $FIND() to WRITE the position of the next occurrence of the character "I" starting in character position three.

Example:

```
GTM>set t=1 for  set t=$find("BANANA","AN",t) quit:'t  write !,t
4
6
GTM>
```

This example uses a loop with $FIND() to locate all occurrences of "AN" in "BANANA". $FIND() returns 4 and 6 giving the positions after the two occurrences of "AN".

Example:

```
GTM>set str="MUMPS databases are hierarchical"
GTM>Write $find(str," ")
7
GTM>Write $find(str,"Z")
0
GTM>Write $find(str,"d",1)
8
GTM>Write $find(str,"d",10)
0
```

The above example searches a string for a sub string, and returns an integer value which corresponds to the next character position after locating the sub string.

$FNumber()

Returns a string containing a formatted number.

The format for the $FNUMBER function is:

```
$FN[UMBER](numexpr,expr[,intexpr])
```

- The numeric expression specifies the number that $FNUMBER() formats.

- The expression (second argument) specifies zero or more single character format control codes; if the expression contains any character other than the defined codes, $FNUMBER() generates a run-time error.

- The optional integer expression (third argument) specifies the number of digits after the decimal point. If the numeric expression has more digits than specified by this argument, $FNUMBER() rounds to obtain the result. If the numeric expression has fewer digits than specified by this argument, $FNUMBER() zero-fills to obtain the result.

- When the optional third argument is specified and the first argument evaluates to a fraction between -1 and 1, $FNUMBER() returns a number with a leading zero (0) before the decimal point (.).

$FNUMBER() formats or edits numbers, usually for reporting. For more information on rounding performed by $FNUMBER(), refer to "$Justify()" (page 224).

The formatting codes are:

- + : Forces a "+" on positive values.

- - : Suppresses the "-" on negative values.

- , : Inserts commas every third position to the left of the decimal within the number.

- T : Represents the number with a trailing, rather than a leading sign; positive numbers have a trailing space unless the expression includes a plus sign (+).

- P : Represents negative values in parentheses, positive values with a space on either side; combining with any other code except comma (,) causes a run-time error.

Examples of $FNUMBER()

Example:

```
GTM>do ^fnum
fnum;
  zprint ^fnum
  set X=-100000,Y=2000
  write "SUPPRESS NEGATIVE SIGN:",?35,$FNumber(X,"-"),!
  write "TRAILING SIGN:",?35,$FNumber(X,"T"),!
  write "NEGATIVE NUMBERS IN ():",?35,$FNumber(X,"P"),!
  write "COMMAS IN NUMBER:",?35,$FNumber(X,","),!
  write "NUMBER WITH FRACTION:",?35,$FNumber(X,"",2),!
  write "FORCE + SIGN IF POSITIVE:",?35,$FNumber(Y,"+"),!
SUPPRESS NEGATIVE SIGN:            100000
TRAILING SIGN:                    100000-
NEGATIVE NUMBERS IN ():           (100000)
COMMAS IN NUMBER:                 -100,000
NUMBER WITH FRACTION:             -100000.00
FORCE + SIGN IF POSITIVE:         +2000
```

Example:

```
set x=$fnumber(x,"-")
```

This example uses $FNUMBER() to SET x equal to its absolute value.

$Get()

Returns the value of a local or global variable if the variable has a value. If the variable has no value, the function returns a value specified by an optional second argument, and otherwise returns an empty string.

The format for the $GET function is:

```
$G[ET](glvn[,expr])
```

- The subscripted or unsubscripted global or local variable name specifies the node for which $GET() returns a value.

- If the global or local variable has a data value, $GET() returns the value of the variable.

- If the global or local variable has no data value, $GET() returns the value of the optional expression (second argument), or an empty string if the expression is not specified.

M defines $GET(x,y) as equivalent to:

```
$Select($Data(x)[0:y,1:x)
```

and $GET(x) as equivalent to:

```
$GET(x,"")
```

$GET() provides a tool to eliminate separate initialization of variables. This technique may provide performance benefits when used to increase the density of a sparse global array by eliminating nodes that would otherwise hold absent optional information. On the other hand, some uses of one argument $GET() can mask logic problems.

GT.M has a "NOUNDEF" mode of operation, which treats all variable references as if they were arguments to a one argument $GET(). The VIEW command controls "NOUNDEF" mode.

Examples of $GET()

Example:

```
setstatus;
        if '$data(^PNT(NAME,TSTR)) set STATUS="NEW TEST"
        else  if ^PNT(NAME,TSTR)="" set STATUS="WAITING FOR RESULT"
        else  set STATUS=^PNT(NAME,TSTR)
```

This example can be reduced to two lines of code by using $GET(), shown in the following example. However, by using $GET() in its one-argument form, the distinction between an undefined variable and one with a null value is lost:

```
set STATUS=$get(^PNT(NAME,TSTR))
if STATUS="" set STATUS="WAITING FOR RESULT"
```

This is solved by using the two-argument form of $GET():

```
set STATUS=$get(^PNT(NAME,TSTR),"NEW TEST")
if STATUS="" set STATUS="WAITING FOR RESULT"
```

$Increment()

Atomically adds (increments) a global variable by a numeric value. Note that increment is atomic, but the evaluation of the expression is not, unless inside a transaction (TStart/TCommit). The function also works on local variables, but has less benefit for locals as it does not (need to) provide ACID behavior.

The format of the $INCREMENT function is:

```
$INCREMENT(glvn[,numexpr])
```

- $I, $INCR, $INCREMENT, $ZINCR, and $ZINCREMENT are considered as valid synonyms of the full function name.

- $INCREMENT() returns the value of the glvn after the increment.

- If not specified, numexpr defaults to 1. Otherwise, $INCREMENT() evaluates the "numexpr" argument before the "glvn" argument.

- numexpr can be a negative value.

- Since it performs an arithmetic operation, $INCREMENT() treats glvn as numeric value. $INCREMENT treats glvn as if it were the first argument of an implicit $GET() before the increment. If the value of glvn is undefined $INCREMENT treats it as having empty string , which means it treats it as a numeric zero (0) (even if glvn is a global variable that resides on a remote node and is accessed through a GT.CM GNP server).

- If $INCREMENT() occurs inside a transaction ($TLevel is non-zero), or if glvn refers to a local variable, it is equivalent to SET glvn=$GET(glvn)+numexpr.

- If $INCREMENT() occurs outside a transaction ($TLevel is zero) and glvn refers to a global variable, the function acts as a SET glvn=$GET(glvn)+numexpr performed as an Atomic, Consistent and Isolated operation. Note that $INCREMENT() performs the evaluation of numexpr before it starts the Atomic, Consistent, Isolated incrementing of the glvn. If the region containing the glvn is journaled, then the $INCREMENT() is also Durable. Only BG, MM (OpenVMS only) and GT.CM GNP access methods are supported for the region containing the global variable (glvn). GT.CM OMI and GT.CM DDP access methods do not support this operation and there are no current plans to add such support.

- $INCREMENT() does not support global variables that have NOISOLATION turned ON (through the VIEW "NOISOLATION" command), and a $INCREMENT() on such a variable, triggers a GVINCRISOLATION run-time error.

- The naked reference is affected by the usage of global variables (with or without indirection) in the glvn and/or numexpr components. The evaluation of "numexpr" ahead of "glvn" determines the value of the naked reference after the $INCREMENT. If neither glvn or numexpr contain indirection, then $INCREMENT sets the naked reference as follows:

  - glvn, if glvn is a global, or

  - the last global reference in "numexpr" if glvn is a local, or

  - unaffected if neither glvn nor numexpr has any global reference.

Examples of $INCREMENT()

Example:

```
GTM>set i=1
GTM>write $increment(i)
2
GTM>write $increment(i)
3
GTM>write $increment(i)
4
GTM>write $increment(i)
5
GTM>write i
5
GTM>write $increment(i,-2)
3
GTM>write i
3
GTM>
```

This example increments the value of i by 1 and at the end decrements it by 2. Note that the default value for incrementing a variable is 1.

$Justify()

Returns a formatted string.

The format for the $JUSTIFY function is:

```
$J[USTIFY](expr,intexpr1[,intexpr2])
```

- The expression specifies the string to be formatted by $JUSTIFY().

- The first integer expression (second argument) specifies the minimum size of the resulting string. If the first integer expression is larger than the length of the expression, $JUSTIFY() right justifies the expression to a string of the specified length by adding leading spaces. Otherwise, $JUSTIFY() returns the expression unmodified unless specified by the second integer argument.

- The optional second integer expression (third argument) specifies the number of digits to follow the decimal point in the result, and forces $JUSTIFY() to evaluate the expression as numeric. If the numeric expression has more digits than this argument specifies, $JUSTIFY() rounds to obtain the result. If the expression had fewer digits than this argument specifies, $JUSTIFY() zero-fills to obtain the result.

- When the second argument is specified and the first argument evaluates to a fraction between -1 and 1, $JUSTIFY() returns a number with a leading zero (0) before the decimal point (.).

$JUSTIFY() fills expressions to create fixed length values. However, if the length of the specified expression exceeds the specified field size, $JUSTIFY() does not truncate the result (although it may still round based on the third argument). When required, use $EXTRACT() to perform truncation.

$JUSTIFY() optionally rounds the portion of the result after the decimal point. In the absence of the third argument, $JUSTIFY() does not restrict the evaluation of the expression. In the presence of the third (rounding) argument, $JUSTIFY() evaluates the expression as a numeric value. The rounding algorithm can be understood as follows:

- If necessary, the rounding algorithm extends the expression to the right with 0s (zeros) to have at least one more digit than specified by the rounding argument.

- Then, it adds 5 (five) to the digit position after the digit specified by the rounding argument.

- Finally, it truncates the result to the specified number of digits. The algorithm rounds up when excess digits specify a half or more of the last retained digit and rounds down when they specify less than a half.

- For a process started in UTF-8 mode, $JUSTIFY() interprets the string argument as UTF-8 encoded. With VIEW "BADCHAR" enabled, $JUSTIFY() produces a run-time error when it encounters a malformed character.

- $ZJUSTIFY() is the parallel function of $JUSTIFY(). Irrespective of the settings of VIEW "BADCHAR" and $ZCHSET, $ZJUSTIFY() interprets argument as a sequence of bytes (rather than a sequence of characters) and can perform all byte-oriented $JUSTIFY() operations. For more information, refer to "$ZJustify()" (page 269).

Examples of $JUSTIFY()

Example:

```
GTM>write ":",$justify("HELLO",10),":",!,":",$justify("GOODBYE",5),":"
:     HELLO:
:GOODBYE:
GTM>
```

This uses $JUSTIFY() to display "HELLO" in a field of 10 spaces and "GOODBYE" in a field of 5 spaces. Because the length of "GOODBYE" exceeds five spaces, the result overflows the specification.

Example:

```
GTM>write "1234567890",!,$justify(10.545,10,2)
1234567890
    10.55
```

```
GTM>
```

This uses $JUSTIFY() to WRITE a rounded value right justified in a field of 10 spaces. Notice that the result has been rounded up.

Example:

```
GTM>write "1234567890",!,$justify(10.544,10,2)
1234567890
     10.54
GTM>
```

Again, this uses $JUSTIFY() to WRITE a rounded value right justified in a field of 10 spaces. Notice that the result has been rounded down.

Example:

```
GTM>write "1234567890",!,$justify(10.5,10,2)
1234567890
     10.50
GTM>
```

Once again, this uses $JUSTIFY() to WRITE a rounded value right justified in a field of 10 spaces. Notice that the result has been zero-filled to 2 places.

Example:

```
GTM>write $justify(.34,0,2)
0.34
GTM>
```

This example uses $JUSTIFY to ensure that the fraction has a leading zero. Note the use of a second argument of zero in the case that rounding is the only function that $JUSTIFY is to perform.

$Length()

Returns the length of a string measured in characters, or in "pieces" separated by a delimiter specified by one of its arguments.

The format for the $LENGTH function is:

```
$L[ENGTH](expr1[,expr2])
```

- The first expression specifies the string that $LENGTH() "measures".

- The optional second expression specifies the delimiter that defines the measure; if this argument is missing, $LENGTH() returns the number of characters in the string.

- If the second argument is present and not an empty string, $LENGTH returns one more than the count of the number of occurrences of the second string in the first string; if the second argument is an empty string, the M standard specifies that $LENGTH() returns a zero (0).

- $LENGTH() provides a tool for determining the lengths of strings in two ways, characters and pieces. The two argument $LENGTH() returns the number of existing pieces, while the one argument returns the number of characters.

- For a process started in UTF-8 mode, $LENGTH() interprets the string argument(s) as UTF-8 encoded. With VIEW "BADCHAR" enabled, $LENGTH() produces a run-time error when it encounters a malformed character.

- $ZLENGTH() is the parallel function of $LENGTH(). Irrespective of the setting of VIEW "BADCHAR" and $ZCHSET, $ZLENGTH() interpets string arguments as a sequence of bytes (rather than characters) and can perform all byte-oriented $LENGTH() operations. For more information, refer to "$ZLength()" (page 270).

Examples of $LENGTH()

Example:

```
GTM>Write $length("KINGSTON")
8
GTM>
```

This uses $LENGTH() to WRITE the length in characters of the string "KINGSTON".

Example:

```
GTM>set x="Smith/John/M/124 Main Street/Ourtown/KA/USA"
GTM>write $length(x,"/")
7
GTM>
```

This uses $LENGTH() to WRITE the number of pieces in a string, as delimited by /.

Example:

```
GTM>write $length("/2/3/","/")
4
GTM>
```

This also uses $LENGTH() to WRITE the number of pieces in a string, as delimited by /. Notice that GT.M. adds one count to the count of delimiters (in this case 3), to get the number of pieces in the string (displays 4).

$NAme()

Returns an evaluated representation of some or all of a local or global variable name.

The format for the $NAME function is:

```
$NA[ME](glvn[,intexpr])
```

- The subscripted or unsubscripted global or local variable name, including naked references, specifies the name for which $NAME() returns an evaluated representation.

- When using NOUNDEF, $NAME() returns an empty string where appropriate for undefined variables.

- The optional integer expression (second argument) specifies the maximum number of subscript levels in the representation. If the integer expression is not provided or exceeds the actual number of subscript levels, $NAME() returns a representation of the whole name. If the integer expression is zero (0), $NAME() returns only the name. A negative integer expression produces a run-time error.

Examples of $NAME()

Example:

```
GTM>set X="A""B",^Y(1,X,"B",4)=""
```

```
GTM>write $name(^(3),3)
^Y(1,"A""B","B")
GTM>
```

This example sets up a naked reference and then uses $NAME() to display the first three levels of that four-level reference.

Example:

```
GTM>write $name(^(3),0)
^Y
GTM>
```

This example shows the name level for the same naked reference.

$Next()

Returns the next subscripted local or global variable name in collation sequence within the array level specified by its argument.

$NEXT() has been replaced by $ORDER(). $NEXT has been retained in the current standard only for compatibility with earlier versions of the standard. $NEXT() is similar to $ORDER(). However, $NEXT() has the deficiency that when it encounters negative one (-1) as a subscript, it returns the same result as when it finds no other data at the level. This deficiency is particularly disruptive because it occurs in the middle of the M collating sequence.

> ⚠️ **Caution**
>
> As $NEXT() has been removed from the standard in the MDC, you should use $ORDER.

The format for the $NEXT function is:

```
$N[EXT](glvn)
```

- The subscripted global or local variable name specifies the node following which $NEXT() searches for the next node with data and/or descendants; the number of subscripts contained in the argument implicitly defines the array level.

- If $NEXT() finds no node at the specified level after the specified global or local variable, it returns negative one (-1).

- If the last subscript in the subscripted global or local variable name is null or negative one (-1), $NEXT() returns the first node at the specified level.

$Order()

Returns the subscript of the next or prior local or global variable name in collation sequence within the array level specified by its first argument. In doing so, it moves in the direction specified by the second argument. In GT.M, when $ORDER() has an unsubscripted argument, it returns the next or previous unsubscripted local or global variable name in collating sequence.

The format for the $ORDER function is:

```
$O[RDER](glvn[,expr])
```

- The subscripted global or local variable name specifies the node from which $ORDER() searches for the next or previous node that has data and/or descendants. The number of subscripts contained in the argument implicitly defines the array level.

- The optional expression (second argument) specifies the direction for the $ORDER(); 1 specifies forward operation and -1 specifies reverse operation. Any other values for the expression will cause an error.

- GT.M extends the M standard to allow unsubscripted names. In this case, $ORDER() returns the next or previous unsubscripted name.

- If $ORDER() finds no node (or name) at the specified level after (or before) the specified global or local variable, it returns an empty string (" ").

- If the last subscript in the subscripted global or local variable name is null and the corresponding subscripted global or local variable has a matching null subscript, $ORDER() returns the next node after that with the null subscript at the specified level.

    If the last subscript in the subscripted global or local variable name is null and the corresponding subscripted global or local variable has no matching null subscript , $ORDER() returns first node at the specified level. If the last subscript in the subscripted global or local variable name is null and second argument is -1, $ORDER() always returns the last node at the specified level regardless of the existence a null subscript at the specified level. However when a global or local variable level includes a null subscript and $ORDER(glvn,-1) returns an empty string result, users must test separately for the existence of the node with the null subscript.

- $ORDER() can be used as a tool for retrieving data from M sparse arrays in an ordered fashion, independent of the order in which it was entered. In M, routines generally sort by SETting data into an array with appropriate subscripts and then retrieving the information with $ORDER().

- $ORDER() returns subscripts, not data values, and does not discriminate between nodes that have data values and nodes that have descendants. Once $ORDER() provides the subscript, the routine must use the subscript to access the data value, if appropriate. Using $ORDER() maintains the naked reference indicator, even if $ORDER() returns a null.

- GT.M optionally permits the use of null subscripts. This feature is enabled via the VIEW command for local variables and a REGION qualifier in GDE for global variables. When an application uses null subscripts, they are "invisible" in a $ORDER() loop so the application must test for them as a special case, perhaps using $DATA().

- $Order() returns local array subscripts with values that are numeric, but non-canonical (over 18 digit), as strings.

> **Note**
>
> Name-level $ORDER() always returns an empty string when used with extended references.

Examples of $ORDER()

Example:

```
GTM>zwrite
lcl(1)=3
lcl("x")=4
GTM>write $order(lcl(""))
1
```

This example returns the first node, that is 1, because the specified last subscript of the argument is null and lcl has no null subscript.

Example:

```
GTM>write $order(lcl(1))
x
```

This example returns the first node after lcl(1) that is x because lcl has no null subscript.

Example:

```
GTM>write $order(lcl(""),-1)
x
```

This example returns the last node that is, x, because the last subscript of the first argument is null and second argument is -1.

```
GTM>set lcl("")=2
GTM>zwrite
lcl("")=2
lcl(1)=3
lcl("x")=4
GTM>write $order(lcl(""))
1
```

This example returns the second node at the specified level because the null subscript at the end of the argument is ambiguous (does it specify starting at the beginning or starting at the real node with the null subscript?) and returning the subscript of the first node (an empty string) would tend to create an endless loop.

Example:

```
GTM>write $order(lcl(""),-1)
x
GTM>write $order(lcl("x"),-1)
1
```

Example:

```
GTM>kill  set (a(1),a(2000),a("CAT"),a("cat"),a("ALF"),a(12))=1
GTM>set x="" for  set x=$order(a(x)) quit:x=""  write !,x
1
12
2000
ALF
CAT
cat
GTM>kill a("CAT") set a(5,10)="woolworths",a("cat")="last"
GTM>set x="" for  set x=$order(a(x),-1) quit:x=""  write !,x
cat
ALF
2000
12
5
1
GTM>
```

This example uses a $ORDER() loop to display all the subscripts at the first level of local variable a, make some changes in a, and then display all the subscripts in reverse order. Notice that $ORDER() returns only the existing subscripts in the sparse array and returns them in M collation sequence, regardless of the order in which they were entered. Also, $ORDER() does not differentiate between node A(5), which has only descendants (no data value), and the other nodes, which have data values.

Example:

```
GTM>kill set (%(1),tiva(2),A(3),tiv(4),Q(5),%a(6))=""
GTM>set x="%"
```

```
GTM>write:$data(@x) !,x for  set x=$order(@x) quit:x=""  write !,x
%
%a
A
Q
tiv
tiva
x
GTM>set $piece(x,"z",32)=""
GTM>write:$data(@x) !,x for  set x=$order(@x,-1) quit:x=""  write !,x
x
tiva
tiv
Q
A
%a
%
GTM>
```

This example uses $ORDER() to display the current local variable names in both forward and reverse order. Notice that the first ([^]%) and last ([^]zzzzzzzz) names require handling as special cases and require a $DATA() function.

Example:

```
set acct="",cntt=""
for  fet acct=$order(^acct(acct)) quit:acct=""  do
. for  set cntt=$order(^acct(acct,cntt)) do WORK
quit
```

This uses two nested $ORDER() loops to cycle through the ^acct global array and perform some action for each second level node.

$Piece()

Returns a substring delimited by a specified string delimiter made up of one or more characters. In M, $PIECE() returns a logical field from a logical record.

The format for the $PIECE function is:

```
$P[IECE](expr1,expr2[,intexpr1[,intexpr2]])
```

- The first expression specifies the string from which $PIECE() computes its result.

- The second expression specifies the delimiting string that determines the piece "boundaries"; if this argument is an empty string, $PIECE() returns an empty string.

- If the second expression does not appear anywhere in the first expression, $PIECE() returns the entire first expression (unless forced to return an empty string by the second integer expression).

- The optional first integer expression (third argument) specifies the beginning piece to return; if this argument is missing, $PIECE() returns the first piece.

- The optional second integer expression (fourth argument) specifies the last piece to return. If this argument is missing, $PIECE() returns only one piece unless the first integer expression is zero (0) or negative, in which case it returns a null string. If this argument is less than the first integer expression, $PIECE() returns an empty string.

- If the second integer expression exceeds the actual number of pieces in the first expression, $PIECE() returns all of the expression after the delimiter selected by the first integer expression.

- The $PIECE() result never includes the "outside" delimiters; however, when the second integer argument specifies multiple pieces, the result contains the "inside" occurrences of the delimiter.

- $PIECE() can also be used as tool for efficiently using values that contain multiple elements or fields, each of which may be variable in length.

- Applications typically use a single character for a $PIECE() delimiter (second argument) to minimize storage overhead, and increase efficiency at run-time. The delimiter must be chosen so the data values never contain the delimiter. Failure to enforce this convention with edit checks may result in unanticipated changes in the position of pieces within the data value. The caret symbol (^), backward slash (\), and asterisk (*) characters are examples of popular visible delimiters. Multiple character delimiters may reduce the likelihood of conflict with field contents. However, they decrease storage efficiency, and are processed with less efficiency than single character delimiters. Some applications use control characters, which reduce the chances of the delimiter appearing in the data but sacrifice the readability provided by visible delimiters.

- A SET command argument can have something that has the format of a $PIECE() on the left-hand side of its equal sign (=). This construct permits easy maintenance of individual pieces within a string. It also can be used to generate a string of delimiters. For more information on SET $PIECE(), refer to "Set" (page 144).

- $PIECE() can also be used as target in a SET command to change part of the value of a node. Also, when SET arguments have multiple parenthesized (set-left) targets and a target is used as a subscript in more than one item in the list of targets that follow, all the targets use the before-SET value (not the after-SET value) in conformance to the M-standard. For more information on SET $PIECE(), refer to "Set" (page 144).

- For a proces started in UTF-8 mode, $PIECE() interprets the string arguments as UTF-8 encoded. With VIEW "BADCHAR" enabled, $PIECE() produces a run-time error when it encounters a malformed character, but it does not process the characters that fall after the span specified by the arguments.

- $ZPIECE() is the parallel function of $PIECE(). Irrespective of the settings of VIEW "BADCHAR" and $ZCHSET, $ZPIECE() interprets string arguments as a sequence of bytes (rather than a sequence of characters) and can perform all byte-oriented $PIECE() operations. For more information, refer to "$ZPIece()" (page 273).

Examples of $PIECE()

Example:

```
GTM>for i=0:1:3 write !,$piece("1 2"," ",i),"<"
<
1<
2<
<
GTM>
```

This loop displays the result of $PIECE(), specifying a space as a delimiter, a piece position "before," first and second, and "after" the string.

Example:

```
GTM>for i=-1:1:3 write !,$piece("1 2"," ",i,i+1),"<"
<
1<
1 2<
```

```
2<
<
GTM>
```

This example is similar to the previous example except that it displays two pieces on each iteration. Notice the delimiter (a space) in the middle of the output for the third iteration, which displays both pieces.

Example:

```
for p=1:1:$length(x,"/") write ?p-1*10,$piece(x,"/",p)
```

This example uses $LENGTH() and $PIECE() to display all the pieces of x in columnar format.

Example:

```
GTM>set $piece(x,".",25)="" write x
.......................
```

This SETs the 25th piece of the variable x to null, with a delimiter of a period. This produces a string of 24 periods preceding the null.

Example:

```
GTM>set ^x=1,$piece(^a,";",3,2)=^b
```

This example leaves the naked indicator to pointing to the global ^b.

$Qlength()

Returns the number of subscripts in a variable name. The format is:

```
$QL[ENGTH] (namevalue)
```

- The namevalue has the form of an evaluated subscripted or unsubscripted global variable.

- $QLENGTH() returns a value which is derived from namevalue. If namevalue has the form NAME(s1, s2,..., sn), then the function returns n; if the name is unsubscripted, $QLENGTH() yields a length of zero (0).

- $QLENGTH() only affects the naked indicator if the string in question is stored in a global variable.

Examples of $QLENGTH()

Example:

```
GTM>write $data(^|"XXX"|ABC(1,2,3,4))
0
GTM>set X=$name(^(5,6))
GTM>write $qlength(X)
5
```

The number of subscripts in x is 5. Notice that the name and the environment preceding it do not contribute to the count. Refer to $NAme() section earlier in this chapter for an understanding of the $NAME function.

$QSubscript()

Returns a component of a variable name.

The format of the $QSUBSCRIPT function is:

```
$QS[UBSCRIPT](namevalue, intexpr)
```

- The namevalue has the form of an evaluated subscripted or unsubscripted global or local variable name.

- The intexpr selects the component of the name as follows:

  - -2 : is reserved but may be "error",

  - -1 : for environment,

  - 0 : for the unsubscripted name,

  - 1 : for the first subscript,

  - 2 : for the second subscript, and so on.

- If the second argument selects a component that is not part of the specified name, $QSUBSCRIPT() returns an empty string ("").

Examples of $QSUBSCRIPT()

Example:

Assume that X is defined as in the "Examples of $Qlength()" earlier in this chapter;

```
write X
X="^|""XXX""|ABC(1,2,3,5,6)"
GTM>write $qsubscript(X,-2)
error
GTM>WRITE $qsubscript(X,-1)
XXX
GTM>WRITE $qsubscript(X,0)
^ABC
GTM>WRITE $qsubscript(X,1)
1
GTM>WRITE $qsubscript(X,4)
5
GTM>WRITE $qsubscript(X,7)
""
```

$Query()

Returns the next subscripted local or global variable node name, independent of level, which follows the node specified by its argument in M collating sequence and has a data value.

The format for the $QUERY function is:

```
$Q[UERY](glvn)
```

- The subscripted or unsubscripted global or local variable name specifies the starting node from which $QUERY() searches for a node with a data value.

- If $QUERY() finds no node after the specified global or local variable, it returns an empty string.

- With stdnullcoll, if $Data(glvn(""))=1 (or 11), $Query(glvn("")) returns glvn(1) (assuming glvn(1) exists). Applications looking for a node with a "null" subscript must use $D(glvn("")) to test the existence of glvn(""). $Q(glvn("...")) never returns the starting-point (glvn("")) even though glvn("") may exist.

$QUERY() can be used as a tool for scanning an entire array for nodes that have data values. Because $QUERY() can return a result specifying a different level than its argument, the result provides a full variable name. This contrasts with $ORDER(), which returns a subscript value. To access the data value at a node, a $ORDER() return can be used as a subscript; however, a $QUERY() return must be used with indirection. Because arrays tend to have homogeneous values within a level but not between levels, $QUERY() is more useful as a tool in utility programs than in application programs. The $QUERY() can be useful in avoiding nested $ORDER loops.

Note that the standard does not unambiguously define the state of the naked reference indicator after a $QUERY(). While in GT.M after $QUERY(), the naked reference indicator reflects the $QUERY() argument, NOT its result.

Examples of $QUERY()

Example:

```
set ^X(1,2,3)="123"
set ^X(1,2,3,7)="1237"
set ^X(1,2,4)="124"
set ^X(1,2,5,9)="1259"
set ^X(1,6)="16"
set ^X("B",1)="AB"
```

The tree diagram below represents the structure produced by the preceding routine.
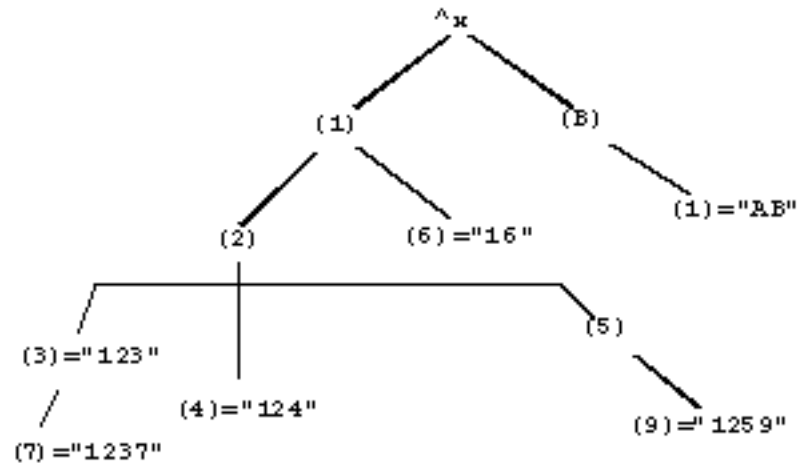
The following routine:

```
set y="^X"
for  set y=$query(@y) quit:y=""  write !,y,"=",@y
```

produces the results:

```
^X(1,2,3)=123
^X(1,2,3,7)=1237
^X(1,2,4)=124
^X(1,2,5,9)=1259
```

```
^X(1,6)=16
^X("B",1)=AB
```

Example:

```
GTM>zwrite lcl
lcl("")=1
lcl(1)=1
lcl(1,2)=2
lcl(1,2,"")=3
lcl(1,2,"","")=4
lcl(1,2,"","",4)=5
lcl(1,2,0)=6
lcl(1,2,"abc",5)=7
lcl("x")=1
GTM>set y="lcl"
GTM>for  set y=$query(@y) quit:y=""  write !,y,"=",@y
```

This example produces the results:

```
lcl("")=1
lcl(1)=1
lcl(1,2)=2
lcl(1,2,"")=3
lcl(1,2,"","")=4
lcl(1,2,"","",4)=5
lcl(1,2,0)=6
lcl(1,2,"abc",5)=7
lcl("x")=1
```

Note that the result is the same as the ZWRITE output.

$Random()

Returns a random integer from a range specified by its argument.

The format for the $RANDOM function is:

```
$R[ANDOM](intexpr)
```

- The integer expression specifies the upper exclusive limit of a range of integers from which $RANDOM() may pick a result; $RANDOM() never returns a number less than zero (0).

- If $RANDOM() has an argument less than one (1), it generates a run-time error.

- $RANDOM can generate numbers up to 2147483646 (that is 2GB - 2).

$RANDOM() provides a tool for generating pseudo-random patterns useful in testing or statistical calculations. $RANDOM() results fall between zero (0) and one less than the argument.

Random number generators use factors from the environment to create sequences of numbers. True random number generation requires a source of what is known as "noise". Pseudo-random numbers appear to have no pattern, but are developed using interactions between factors that vary in ways not guaranteed to be entirely random. In accordance with the M standard, the GT.M implementation of $RANDOM() produces pseudo-random numbers.

Examples of $RANDOM()

Example:

```
GTM>for i=1:1:10 write $random(1)
0000000000
GTM>
```

This shows that when $RANDOM() has an argument of one (1), the result is too confined to be random.

Example:

```
set x=$random(100)+1*.01
```

This $RANDOM() example produces a number between 0 and 99. The example then shifts with addition, and scales with multiplication to create a value between .01 and 1.

$REverse()

Returns a string with the characters in the reverse order from that of its argument.

The format for the $REVERSE function is:

```
$RE[VERSE](expr)
```

- The expr in the syntax is the string to be reversed.

Examples of $REVERSE()

Example:

```
GTM>write $reverse(123)
321
GTM>write $reverse("AbCDe")
"eDCbA"
```

$Select()

Returns a value associated with the first true truth-valued expression in a list of paired expression arguments.

The format for the $SELECT function is:

```
$S[ELECT](tvexpr:expr[,...])
```

- $SELECT() evaluates expressions from left to right.

- If a truth-valued expression is TRUE (1), $SELECT() returns the corresponding expression after the colon (:) delimiter.

- Once $SELECT() finds a TRUE, the function does not process any remaining arguments.

- If $SELECT() finds no TRUE truth-value in its list of arguments, the function generates a run-time error.

- $SELECT() does not have any effect on $TEST.

$SELECT() is one of a limited set of functions that permit an indefinite number of arguments. $SELECT() provides a means of selecting from a list of alternatives.

Generally, the last $SELECT() argument has numeric literal one (1) for a truth-value to prevent run-time errors, and to provide a "default" value.

Examples of $SELECT()

Example:

```
GTM>for i=3:-1:0 write !,$select(i=1:"here",i=2:"come",i=3:"Watson")
Watson
come
here
%GTM-E-SELECTFALSE, No argument to $SELECT was true
GTM>
```

This loop uses $SELECT() to WRITE a series of strings. Because there is no true argument on the fourth iteration, when i=0, $SELECT() produces an error.

Example:

```
set name=$select(sex="M":"Mr. ",sex="F":"Ms. ",1:"")_name
```

This example uses $SELECT() to add a prefix to the name based on a sex code held in the variable sex. Notice that the default handles the case of a missing or incorrect code.

Example:

```
if $select(x=+x:x,x="":0,"JANAPRJULOCT"[x:1,1:0) do THING
```

This uses $SELECT() to perform complex logic as the truth-valued expression argument to an IF command.

$STack()

Returns strings describing aspects of the execution environment.

The format for the $STACK function is:

```
$ST[ACK](intexpr[,expr])
```

- The intexpr identifies the M virtual machine stack level (as described by the standard), on which the function is to provide information.

- The optional second argument is evaluated as a keyword that specifies a type of information to be returned as follows:

  - "MCODE" the line of code that was executed.

  - "PLACE" the address of the above line of code or the symbol at ("@") to indicate code executed from a string value.

    > **Note**
    >
    > For run-time errors, GT.M does not provide a "PLACE" within a line (unlike it does for compilation errors), but it reports a label, offset, and routine.

  - "ECODE" either an empty string, or the error code(s) that was added at this execution level.

- When $STACK has only one argument, values corresponding to available stack levels specify a return value that indicates how the level was created, as follows:

- If intexpr is zero (0), the function returns information on how GT.M was invoked.

- If intexpr is minus one (-1), the function returns the highest level for which $STACK can return information. Note that, if $ECODE="", $STACK(-1) returns the same value as the $STACK ISV.

- If intexpr is greater than zero (0) and less than or equal to $STACK(-1), indicates how this level of process stack was created ("DO", "TRIGGER" - for a stack level invoked by a trigger, "XECUTE", or "$$" - for an extrinsic function).

- $STACK(lvl) reports "ZINTR" for a stack level invoked by MUPIP INTRPT.

- If intexpr is greater than $STACK (-1), the function returns an empty string.

- During error handling, $STACK() return a snapshot of the state of the stack at the time of error. Even if subsequent actions add stack levels, $STACK() continues to report the same snapshot for the levels as of the time of the error. $STACK() reports the latest stack information only after the code clears $ECODE.

- $STACK() assists in debugging programs.

> **Note**
>
> $STACK() returns similar information to ZSHOW "S" when ""=$ECODE, but when $ECODE contains error information, $STACK() returns information as of the time of a prior error, generally the first entry in $ECODE. For $STACK() to return current information, be sure that error handing code does a SET $ECODE="" before restoring the normal flow of control.

Examples of $STACK()

Example:

```
/usr/lib/fis-gtm/V5.4-002B_x86/gtm -run ^dstackex
dstackex;
  zprint ^dstackex
  write !,$STACK
  xecute "WRITE !,$STACK"
  do Label
  write !,$$ELabel
  write !,$STACK
  quit

Label
  write !,$STACK
  do DLabel
  quit

ELabel()
  quit $STACK

DLabel
  write !,$STACK
  quit
0
```

```
1
1
2
1
```

Example for error processing:

```
GTM>zprint ^debugerr
debugerr;
 set dsm1=$stack(-1)
 write !,"$stack(-1):",dsm1
 for l=dsm1:-1:0 do
 . write !,l
 . for i="ecode","place","mcode" write ?5,i,?15,$stack(l,i),!
GTM>
```

The above example can be used to display a trace of the code path that led to an error.

Example:

```
GTM>zprint ^dstacktst
dstacktst(x)        ; check $stack() returns with and without clearing $ecode
 set $etrap="do ^debugerr"
label
 if x>0 set $ecode=",U1," ; if condition
 else  set $ecode=",U2," ;  else condition
 quit
GTM>do ^dstacktst(0)
$stack(-1):2
2    ecode
     place      debugerr+3^debugerr
     mcode       for l=dsm1:-1:0 do
1    ecode      ,U2,
     place      label+2^dstacktst
     mcode       else  set $ecode=",U2," ;  else condition
0    ecode
     place      +1^GTM$DMOD
     mcode
%GTM-E-SETECODE, Non-empty value assigned to $ECODE (user-defined error trap)
GTM>do ^dstacktst(1)
$stack(-1):1
1    ecode      ,U2,
     place      label+2^dstacktst
     mcode       else  set $ecode=",U2," ;  else condition
0    ecode
     place      +1^GTM$DMOD
     mcode
%GTM-E-SETECODE, Non-empty value assigned to $ECODE (user-defined error trap)
GTM>set $ecode=""
GTM>do ^dstacktst(1)
$stack(-1):2
2    ecode
     place      debugerr+3^debugerr
     mcode       for l=dsm1:-1:0 do
1    ecode      ,U1,
     place      label+1^dstacktst
```

```
     mcode       if x>0 set $ecode=",U1," ; if condition
0    ecode
     place       +1^GTM$DMOD
     mcode
%GTM-E-SETECODE, Non-empty value assigned to $ECODE (user-defined error trap)
GTM>
```

This example shows how SETing $ECODE=.. makes $STACK() reports current information. Notice how ^do dstacktst(0) and ^dostacktst(1) without clearing $ECODE in between displays information frozen at the time of the first error (else condition).

$Text()

Returns source text for the line specified by its argument.

The format for the $TEXT function is:

```
$T[EXT](entryref)
```

- The entryref specifies the label, offset, and routine (or trigger name) of the source line that $TEXT() returns.

- If the label+offset combination do not fall within the routine, $TEXT returns a null string.

- If the entryref explicitly or implicitly specifies an offset of zero (0) from the beginning of the routine (or trigger name), $TEXT() returns the routine name or trigger name.

- If the entryref does not specify a routine/trigger, GT.M assumes the current routine/trigger, that is, the routine/trigger at the top of a ZSHOW "S."

- A GT.M extension to $TEXT() permits negative offsets; however, every offset must still be preceded by a plus sign (+) delimiter, (for example, LABEL+-3). If a negative offset points to a line prior to the zero line, $TEXT() generates a run-time error.

$TEXT() provides a tool for examining routine source code and the name of the current routine or trigger. $TEXT() assists, along with the ZPRINT command, in debugging programs. $TEXT() also allows the insertion of small tables of driver information into a routine. Because $TEXT() is not very efficient and the table-driven technique is generally best suited to minimal program changes, this approach is best used for prototyping and the tables should reside in global variables for production.

If $TEXT() cannot access the source file for the current object, either because it is not in the location from which it was compiled or because the process does not have access to some piece of the path to the source, or if the located source does not match the object currently in use by the process, $TEXT() returns an empty string.

Examples of $TEXT()

Example:

```
for i=1:1 set x=$text(+i) quit:x=""  write !,x
```

This loop uses $TEXT() to write out the entire source for the current routine.

Example:

```
GTM>write $text(+0)
GTM$DMOD
```

```
GTM>write $text(+1)
GTM>
```

This uses $TEXT() to WRITE the name of the current routine, then it tries to access the source and returns an empty string. This occurs because the default Direct Mode image is compiled by FIS and delivered without source. The exact failure message may vary.

$TRanslate()

Returns a string that results from replacing or dropping characters in the first of its arguments as specified by the patterns of its other arguments.

The format for the $TRANSLATE function is:

```
$TR[ANSLATE](expr1[,expr2[,expr3]])
```

- The first expression specifies the string on which $TRANSLATE() operates. If the other arguments are omitted, $TRANSLATE() returns this expression.

- The optional second expression specifies the characters for $TRANSLATE() to replace. If a character occurs more than once in the second expression, the first occurrence controls the translation, and $TRANSLATE() ignores subsequent occurrences. If this argument is omitted, $TRANSLATE() returns the first expression without modification.

- The optional third expression specifies the replacement characters for positionally corresponding characters in the second expression. If this argument is omitted or shorter than the second expression, $TRANSLATE() drops all occurrences of characters in the second expression that have no replacement in the corresponding position of the third expression.

- For a process started in UTF-8 mode, the algorithm of $TRANSLATE() treats the string arguments as UTF-8 encoded. With VIEW "BADCHAR" enabled, $TRANSLATE() produces a run-time error when it encounters a malformed character.

- Irrespective of the settings of VIEW "BADCHAR" and $ZCHSET, $ZTRANSLATE() interprets argument as a sequence of bytes (rather than a sequence of characters) and performs all byte-oriented $TRANSLATE() operations. For more information, refer to "$ZTRanslate()" (page 285).

- $TRANSLATE() provides a tool for tasks such as changing case and doing encryption. For examples of case translation, refer to the ^%LCASE and ^%UCASE utility routines.

The $TRANSLATE() algorithm can be understood as follows:

- $TRANSLATE() evaluates each character in the first expression, comparing it character by character to the second expression looking for a match. If there is no match in the second expression, the resulting expression contains the character without modification.

- When it locates a character match, $TRANSLATE() uses the position of the match in the second expression to identify the appropriate replacement for the original expression. If the second expression has more characters than the third expression, $TRANSLATE() replaces the original character with a null, thereby deleting it from the result. By extension of this principle, if the third expression is missing, $TRANSLATE() deletes all characters from the first expression that occur in the second expression.

Examples of $TRANSLATE()

Example:

```
GTM>write $translate("ABC","CB","1")
```

```
A1
GTM>
```

- First, $TRANSLATE() searches for "A" (the first character in the first expression, "ABC") within the second expression ("CB"). Since "A" does not exist in the second expression, it appears unchanged in the result.

- Next, $TRANSLATE() searches for "B" (the second character in the first expression) within the second expression ("CB"). Because "B" holds the second position in the second expression ("CB"), $TRANSLATE() searches for the character holding the second position in the third expression. Since there is no second character in the third expression, $TRANSLATE() replaces "B" with a null, effectively deleting it from the result.

- Finally, $TRANSLATE() searches for "C" (the third character in the first expression) within the second expression ("CB"), finds it in the first position, and replaces it with the number 1, which is in the first position of the third expression. The translated result is "A1."

> **Note**
>
> While this example provides an explanation for the work done by $TRANSLATE(), it does not necessarily correspond to how GT.M implements $TRANSLATE().

Example:

```
GTM>write $translate("A","AA","BC")
B
GTM>
```

This $TRANSLATE() example finds the first occurrence of "A" in the second expression, which holds the first character position, and substitutes the character in the first position of the third expression.

Example:

```
GTM>write $translate("BACKUP","AEIOU")
BCKP
GTM>
```

Because the $TRANSLATE() has only two parameters in this example, it finds the characters in the first expression that also exist in the second expression and deletes them from the result.

$View()

Returns information about an environmental factor selected by the arguments. In GT.M, the first argument contains a keyword identifying the environmental factor and, where appropriate, subsequent arguments select among multiple possible occurrences of that factor.

The format for the $VIEW() function is:

```
$V[IEW](expr1[,expr2])
```

- The first expression specifies a keyword identifying the target factor for $VIEW() to examine.

- The second expression differentiates between multiple possible targets for some keywords. $VIEW() requires the second expression for some keywords and does not permit it for others.

Argument Keywords of $VIEW()

$VIEW() provides a means to access GT.M environmental information. When GT.M permits modification of the factors accessible with $VIEW(), the VIEW command generally provides the means for effecting the change.

$VIEW() Argument Keywords		
ARG 1	ARG 2	RETURN VALUE
"BADCHAR"	none	In UTF-8 mode processes, enables or disable the generation of an error when character-oriented functions encounter malformed byte sequences (illegal characters). The default is 1.
"BREAKMSG"	none	Value of the break message mask; GT.M defaults this to 31.
"DEVICE"	Device name	Device type (FIFO, NULL, PIPE, RMS, SOCKET, or TERMINAL) and device status (OPEN or CLOSED) separated by a colon (":")
"FREEBLOCKS"	region	Number of free database blocks in a given region.
"FREEZE"	region	Process-id of a process that has frozen the database associated with the region specified (using DSE or MUPIP).

If the region is currently not frozen, returns zero. |
| "FULL_BOOLEAN" | none | Returns a string describing the current compiler setting. The default is "GT.M Boolean short-circuit". $VIEW("FULL_BOOLEAN") reports "Standard Boolean evaluation side effects" when it is not explicitly set, but that mode of operation is required by the setting of gtm_side_effects, and "Standard Boolean side-effect warning" when warnings have been specified. The function reports "Extended Boolean evaluation without short-circuiting" when "EXTENDED_BOOLEAN" behavior is enabled. |
| "GDSCERT" | none | Truth Value indicating whether Database block certification is currently enabled or disabled.

To enable or disable Database block certification, use the VIEW "GDSCERT" command. |
"GVACCESS_METHOD"	region	Access method of the region.
"GVFILE"	region	Name of the database associated with the region.
"GVFIRST"	none	Name of the first database region in the current global directory; functionally equivalent to $VIEW("GVNEXT","").
"GVNEXT"	region	Name of the next database region after the given one in alphabetical order (or M collation sequence); "" for region starts with the first region. A return value of "" means that the global directory defines no additional regions.
"GVSTAT"	region	A read-only process cannot update the database including the database file header where GVSTATS are stored. Another process with write access to a database, such as MUPIP RUNDOWN, can flush its read statistics from the associated shared memory to GVSTATS.
"ICHITS"	none	Number of indirection cache hits since GT.M process startup.

$VIEW() Argument Keywords		
ARG 1	**ARG 2**	**RETURN VALUE**
		Indirection cache is a pool of compiled expressions that GT.M maintains for indirection and XECUTE.
"ICMISS"	none	Number of indirection cache misses since GT.M process startup.
"JNLACTIVE"	region	can return the following values: • -1 (internal error) • 0 journaling is disabled • 1 journaling is enabled but closed (OFF) • 2 journaling is enabled and open (ON)
"JNLFILE"	region	Journal file name associated with the region.
"JNLPOOL"	replication instance file	Returns the replication instance file name for the current Journal Pool and an empty string when there is no Journal Pool. Specifying a second expression for $VIEW("JNLPOOL") provides a means of iterating through active Journal Pools. If the second expression is an empty string, the function returns the replication instance file name associated with the instance first attached by the process or. if the process has not previously engaged with any instance, the string "*". If the file name specified in the second expression does not match the replication instance file name for any of the active Journal Pools the function returns the string "*". Otherwise, the function returns the file name of the Journal Pool attached after the Journal Pool with the specified file name. Note the two argument form of $VIEW("JNLPOOL") does not change the current Replication Instance. Note also that the current Journal Pool may not be associated with the last global accessed by an extended reference.
"JNLTRANSACTION"	none	Index showing how many ZTSTART transaction fences have been opened (and not closed).
"LABELS"	none	Truth value showing whether label case sensitivity is ON (1 for "LOWER") or OFF (0 for "UPPER"); GT.M defaults to 1.
"LINK"	none	Returns the current relink recursive setting of ZLINK.
"LV_CREF"	local variable name (lvn)	returns the number of references by alias containers to the array associated with an unsubscripted local variable name specified as a second expr (for example a quoted string); it returns a zero for a variable without any associated alias container.
"LV_GCOL"	none	returns the number of data-spaces recovered during a local variable data-space garbage collection it triggers; such collections normally happen automatically at appropriate times.
"LV_REF"	local variable name (lvn)	returns the total number of references to the data-space associated with an unsubscripted local variable name specified as a second expr (for example a quoted string).

$VIEW() Argument Keywords		
ARG 1	ARG 2	RETURN VALUE
"LVNULLSUBS"	none	Truth value showing whether null subscripts are permitted in local arrays (1 for "LVNULLSUBS") or not (0 for "NOLVNULLSUBS"); GT.M defaults to 1.
"NOISOLATION"	global	The current isolation-status of the specified global variable which must have a leading "^" in its specification. This function returns 1 if GT.M has been instructed to not enforce the ACID property of Isolation (that is, "NOISOLATION" has been specified) and 0 otherwise. By default, GT.M ensures Isolation, that is, a $VIEW command will return 0. The isolation-status of a global variable can be turned on and off by the VIEW "NOISOLATION" command.
"PATCODE"	none	Name of the active patcode table; GT.M defaults this to "M".
"POOLLIMIT"	region	The current limit on global buffers for the region .
"PROBECRIT"	region	Acquires and releases a critical section for the region (the "probe"), returning a string containing following fields, some of of which always have zero (0) values because they are no longer used: • CPT - nanoseconds for the probe to get the critical section • CFN - 0 • CQN - 0 • CYN - 0 • CQF - 0 • CQE - 0 • CAT - total of critical section acquisitions successes
"REGION"	gvn	Name of the region(s) holding the specified gvn. If gvn spans more than one region, this function returns region name in an order where the first region is the region to which the unsubscripted global variable name maps; and other regions are in the order in which they would be encountered by traversing the subscripts of gvn in order (with duplicates removed). gvn is a subscripted or unsubscripted global variable name in the same form as that generated by $NAME(). You can use $NAME() inside $VIEW() to ensure that subscripts are in a correct form, for example, $VIEW("REGION",$NAME(^abcd(1,2E4))) instead of $VIEW("REGION","^abcd(1,20000)").
"RTNCHECKSUM"	routine name	Source code check-sum for the most recently ZLINK'd version of the specified routine name (these check-sums use a 128 bit hash based on the MurmurHash3 algorithm).

$VIEW() Argument Keywords		
ARG 1	**ARG 2**	**RETURN VALUE**
"RTNNEXT"	routine name	Name of the next routine in the image after the given one; "" (empty string) for routinename starts with the first routine in ASCII collating sequence and a return value of the empty string indicates the end of the list.
"SPSIZE"	none	Returns a string with two comma separated values: Number of bytes currently allocated as process working storage: GT.M manages this space as what is commonly called a heap, and uses the term stringpool to refer to it. The GT.M garbage collector reclaims unused space from the stringpool from time to time, and GT.M automatically expands the stringpool as needed by the application program; Number of bytes currently used by the process.
"STATSHARE"	region	Returns 0 when the process has sharing disabled, 1 when it has sharing enabled, and 2 when sharing is enabled selectively for regions. For a process to store statistics in the stats db, the database must be enabled for sharing and the process must have opted in to share. VIEW "STATSHARE" with no region argument enables sharing for all regions and VIEW "STATSHARE":"REGION_NAME" enables sharing selectively for a region. $VIEW("STATSHARE","REGION_NAME") returns whether a process has opted to share statistics for a region.
"STKSIZ"	none	Returns the GT.M stack size in bytes.
"TOTALBLOCKS"	region	Total number of database blocks in a given region.
"TRANSACTIONID"	NULL or transaction level	Transaction ID specified in the particular level (when the transaction level is specified). The first level TSTART is returned if the level is not specified as second argument. **Note** A NULL string is returned if the specified level (explicitly or implicitly) is greater than the current value of $TLEVEL.
"UNDEF"	none	Truth value showing whether undefined variables should be treated as having a null value (1 for "UNDEF"; 0 for "NOUNDEF"); GT.M defaults to 0.
"ZDATE_FORM"	none	Integer value showing whether four digit year code is active for $ZDATE(); GT.M defaults to 0 (for "YY" format). Use the environment variable gtm_zdate_form to set the initial value of this factor. For usage examples, refer to "$ZDate()" (page 262).

⚠️ **Important**

FIS uses the LC_CREF, LV_GCOL, LV_REF keywords in testing and is documenting them to ensure completeness in product documentation. They may (or may not) be useful during application development for debugging or performance testing implementation alternatives.

Examples of $VIEW()

Example:

```
GTM>Set a=1,*b(1)=a
GTM>write $view("LV_CREF","a")," ",$view("LV_CREF","b")
1 0
GTM>write $view("LV_REF","a")," ",$view("LV_REF","b")
2 1
GTM>
```

This example creates an alias variable and an alias container variable and checks the number of both container references and total references to the cells associated with both a and b.

Example:

```
GTM>Set *a(1)=b,*b(1)=a
GTM>kill *a,*b
GTM>write $view("LV_GCOL")
2
GTM>
```

This example creates two cross associated alias containers, destroys their ancestor nodes with KILL * and uses $VIEW("LV_GCOL") to force a clean-up of the abandoned data-spaces. In the absence of the $VIEW("LV_GCOL"), GT.M would do this automatically at some subsequent convenient time.

Example:

```
GTM>write $view("GVSTAT","DEFAULT")
SET:203,KIL:12,GET:203,DTA:2,ORD:23,ZPR:21,QRY:0,LKS:0,LKF:0,CTN:44,DRD:103,DWT:59,
NTW:24,NTR:55,NBW:27,NBR:138,NR0:0,NR1:0,NR2:0,NR3:0,TTW:17,TTR:5,TRB:0,TBW:32,
TBR:80,TR0:0,TR1:0,TR2:0,TR3:0,TR4:0,TC0:0,TC1:0,TC2:0,TC3:0,TC4:0,ZTR:7,DFL:9,
DFS:0,JFL:0,JFS:0,JBB:0,JFB:0,JFW:0,JRL:0,JRP:0,JRE:0,JRI:0,JRO:0,JEX:0,DEX:0,
CAT:35,CFE:0,CFS:0,CFT:0,CQS:0,CQT:0,CYS:0,CYT:0,BTD:13,WFR:0,BUS:0,BTS:0,STG:0,
KTG:0,ZTG:0,DEXA:0,GLB:0,JNL:0,MLK:0,PRC:0,TRX:0,ZAD:0,JOPA:0,AFRA:0,BREA:0,MLBA:0,
TRGA:0,WRL:0,PRG:0,WFL:0,WHE:0,INC:0
GTM>
```

These are statistics associated with the DEFAULT region. Refer to "ZSHOW Information Codes" (page 193) for information on the parameters.

Example:

Given the following global directory configuration:

```
GDE>add -name a(1:10)      -region=a1
GDE>add -name a(10,1)      -region=a2
GDE>add -name a(10,2)      -region=a3
GDE>add -name a(120:300)   -region=a4
```

```
GDE>add -name a(60:325)      -region=a5
GDE> show -name
 *** NAMES ***
 Global         Region
 -----------------------------------------------------------------------
 *              DEFAULT
 a(1:10)        A1
 a(10,1)        A2
 a(10,2)        A3
 a(60:120)      A5
 a(120:300)     A4
 a(300:325)     A5
```

Here are some $VIEW("REGION",gvn) outputs:

```
GTM>write $view("REGION","^a(1)")
A1
GTM>write $view("REGION","^a(10)")
DEFAULT,A2,A3
GTM>w $view("REGION","^a(60)")
A5
GTM>w $view("REGION","^a")
DEFAULT,A1,A2,A3,A5,A4
```

Some examples of $VIEW("DEVICE",name) usage:

```
GTM> WRITE $VIEW("DEVICE","0")
TERMINAL:OPEN
```

This indicates the $PRINCIPAL device is a terminal and it is open (which is usually the case for $PRINCIPAL.) The $ZPIN and $ZPOUT intrinsic special variables can be used as the device name to select to corresponding side of a split $PRINCIPAL device.

```
GTM> OPEN "f.txt"
GTM> CLOSE "f.txt":NODESTROY
GTM> WRITE $VIEW("DEVICE","f.txt")
RMS:CLOSED
```

$ZAHandle()

$ZAHANDLE() returns a unique identifier (handle) for the array associated with a name or an alias container; for an subscripted lvn, it returns an empty string. To facilitate debugging, the handle is a printable string representation of a hexadecimal number. The only meaningful operation on the value returned by a call to $ZAHANDLE() is to compare it for equality with the value returned by another call. Changing nodes within the array doesn't change its handle. $ZAHANDLE() returns different results for copies of an array.

Example:

```
GTM>set A=1,*B(1)=A
GTM>write "$zahandle(A)=""",$zahandle(A),""" $zahandle(B(1))=""",$zahandle(B(1)),""""
$zahandle(A)="17B8810" $zahandle(B(1))="17B8810"
GTM>set A("Subscript")="Value" ; Change array - but $ZAHandle() doesn't change
GTM>write "$zahandle(A)=""",$zahandle(A),""" $zahandle(B(1))=""",$zahandle(B(1)),""""
$zahandle(A)="17B8810" $zahandle(B(1))="17B8810"
GTM>merge D=A ; A copy of the data has a different $zahandle()
GTM>Write "$ZAHandle(A)=""",$ZAHandle(A),""" $ZAHandle(D)=""",$ZAHandle(D),""""
```

```
$zahandle(A)="17B8810" $zahandle(D)="17B8C10"
GTM>
```

Since GT.M does not provide a way for a function to return an array or alias variable as its result, the uniqueness of $ZAHandle() can be exploited to effect this capability, by placing the result in a local variable with an agreed prefix (e.g., "%") and its $ZAHANDLE() as a suffix. The handle can be returned as the value.

```
$ /usr/lib/fis-gtm/V5.4-002B_x86/gtm -run retval
retval          ; Return an array / object from a function
    ;;Data for the object array
    ;;Albert Einstein,14-March-1879
    ;;Arthur Eddington,28-December-1882
    ;;
    zprint      ; Print this program
    new tmp1,tmp2,tmp3
    for i=3:1 set tmp1=$text(+i),tmp2=$piece(tmp1,";;",2) quit:'$length(tmp2)  do
    .set tmp3="%"_$$NewPerson($piece(tmp2,",",1),$piece(tmp2,",",2))
    .set @("*Relativists("_(i-2)_")")="_tmp3)
    .kill @("*"_tmp3)
    kill tmp1,tmp2,tmp3
    write "------------",!
    write "Array of objects of relativists:",!
    zwrite
    quit
    ;
NewPerson(name,birthdate)     ; Create new person object
    new lname,fname,dob,tmp1,tmp2 ; New variables used by this function
    set lname=$Piece(name," ",2),fname=$Piece(name," ",1)
    set dob=$$FUNC^%DATE(birthdate)
    set tmp1("fname")=fname,tmp1("lname")=lname,tmp1("dob")=dob
    set tmp2=$ZAHandle(tmp1)
    set @("*%"_tmp2_"=tmp1")
    quit tmp2
------------
Array of objects of relativists:
$ZWRTAC=""
*Relativists(1)=$ZWRTAC1
$ZWRTAC1("dob")=13952
$ZWRTAC1("fname")="Albert"
$ZWRTAC1("lname")="Einstein"
*Relativists(2)=$ZWRTAC2
$ZWRTAC2("dob")=15337
$ZWRTAC2("fname")="Arthur"
$ZWRTAC2("lname")="Eddington"
i=5
$ZWRTAC=""
$
```

$ZAscii()

Returns the numeric byte value (0 through 255) of a given sequence of octets (8-bit bytes).

The format for the $ASCII function is:

```
$ZA[SCII](expr[,intexpr])
```

- The expression is the sequence of octets (8-bit bytes) from which $ZASCII() extracts the byte it decodes.

- The optional integer expression contains the position within the expression of the byte that $ZASCII() decodes. If this argument is missing, $ZASCII() returns a result based on the first byte position. $ZASCII() starts numbering byte positions at one (1), (the first byte of a string is at position one (1)).

- If the explicit or implicit position is before the beginning or after the end of the expression, $ZASCII() returns a value of negative one (-1).

- $ZASCII() provides a means of examining bytes in a byte sequence. When used with $ZCHAR(), $ZASCII() also provides a means to perform arithmetic operations on the byte values associated with a sequence of octets (8-bit bytes).

Examples of $ZASCII()

Example:

```
GTM>for i=0:1:4 write !,$zascii("主",i)

-1
228
184
187
-1
GTM>
```

This UTF-8 mode example displays the result of $ZASCII() specifying a byte position before, first, second and third positions, and after the sequence of octets (8-bit bytes) represented by 主. In the above example, 228, 184, and 187 represents the numeric byte value of the three-byte in the sequence of octets (8-bit bytes) represented by 主.

$ZATRansform

Returns the transformed representation of the first argument expr in a normalized form using the alternative transform specified by the second argument intexpr; the transformed representation can be used as an operand to the follows (]) or sorts-after (]]) operator such that, if both operands are in the normalized form, the result is independent of alternative collation. The format for the $ZATRANSFORM() function is:

```
$ZATRANSFORM(expr,intexpr[,{0|1|2|-2}][,{0|1}])
```

- The expression specifies the string to transform.

- The intexpr specifies the ID of the alternative transform to use.

- The optional third argument specifies:

  - zero (0): the transform is to normalized form

  - one (1): the reverse transform from the normalized to the native form

  - two (2): the character which collates immediately after the first character of the first argument, or the empty string if no character does.

  - minus two (-2): character which collates immediately before the first character of the first argument, or the empty string if no character does.

- The optional fourth argument specifes whether to use standard M collation of numbers before strings, the default or zero (0), or to sort all values as strings (1).

Please see "$ZCOLlate()" (page 259) for a similar alternative.

The 2 and -2 work in M mode for the 'M' collation (collation 0), or any user defined collation which supplies the necessary plugin functionality. In UTF-8 mode, these argument values produce a ZATRANSCOL error. If the plugin for the specified collation does not support this "next character" functionality, the function produces a COLLATIONUNDEF error in response to an an attempt to invoke it. If an external collation library is used and encounters a gtm_ac_xutil failure, ZATRANSFORM produces an ERR_ZATRANSCOL. If an external collation library is used and does not supply a gtm_ac_xutil function, ZATRANSFORM produces an ERR_COLLATIONUNDEF when operations -2 or 2 are specified. To use these operations with external collation libraries, the libraries must supply a function called 'gtm_ac_xutil'. For more information on the following signature and characteristics of the gtm_ac_xutil function, refer to "Transform Utility Routine (gtm_ac_xutil)" (page 550).

Examples of $ZATRANSFORM()

Example:

```
GTM>write $zatransform("John Smythe",1)]$zatransform("Jane Smith",2)
0
GTM>
```

This example uses $ZATRANSFORM() and two (here unspecified) collation definitions to compare the ordering of two (literal) expressions as GT.M would collate them if there was a way to collate them together. The result indicates that the first would collate before the second.

$ZAUditlog

Sends its argument to an audit logger/listener process. This function requires setting the AZA_ENABLE audit logging facility in the $gtm_dist/restrict.txt file. For information on setting up the AZA_ENABLE audit logging facility, refer to "Configuring the Restriction Facility" section in the **GT.M Administration and Operations Guide**.The format for the $ZAUDITLOG() function is:

```
ZAUDITLOG(expr)
```

- expr specifies the string to send for audit logging

- A return of TRUE (1) indicates successful logging, FALSE (0) indicates logging is not enabled; a trappable RESTRICTEDOP error indicates logging is enabled but not working.

- $ZAUDITLOG() identifies its message with src=4, and like other GT.M logging facilities, records the location of GT.M distribution, uid, euid, pid, tty, and the command / argument(s).

- If LGDE is specified as an option for the AZA_ENABLE facility, GDE logs all commands. GT.M ignores this option if specified with other A*_ENABLE audit logging facilities. When it fails to log a command, GDE issues a GDELOGFAIL error. The following table characterizes $ZAUDITLOG() and GDE audit logging behavior:

$ZAUDITLOG() / GDE logging Characteristics				
AZA_ENABLE	LGDE	Logging success	GDE audit logging	$ZAUDITLOG() result
Yes	Yes	Yes	Yes	1

$ZAUDITLOG() / GDE logging Characteristics				
AZA_ENABLE	LGDE	Logging success	GDE audit logging	$ZAUDITLOG() result
Yes	No	Yes	No	1
Yes	Yes	No	GDELOGFAIL error	RESTRICTEDOP error
Yes	No	No	No	RESTRICTEDOP error
No	N/A	N/A	No	0

Examples of $ZAUDitlog()

Example:

```
GTM>write $zauditlog("Name Change for "_ip)
1
GTM>
```

This example uses $ZAUDITLOG() to log a literal label concatenated with a variable (identifier). The return indicates the logging was successful.

$ZBIT Functions

A series of functions beginning with $ZBIT lets you manipulate a bit stream. Internally, GT.M stores a bit stream in the form of a bit string. A bit string embeds a bit stream in such a way that the first byte specifies the number of trailing bits in the last byte that are not part of the bit-stream. In this way, GT.M is able to store bit-streams of lengths other than multiples of 8 bits in byte format. So for example, a first byte of value of zero (0) indicates that all of the bits in the last byte belong to the bit-stream, while a one (1) indicates the last bit is excluded and a seven (7) indicates that only the first bit in the last byte belongs to the bit-stream.

If you have to convert a character string into a bit string then add a leading byte to that character string so that all $ZBIT functions can recognize it. The most common and straightforward way of doing this is to concatenate a $CHAR(n) on the front of the character string, where the value of n is zero through seven (0-7) – most commonly zero (0). If you pass a bit string as an argument to a routine that is expecting a character string, then that caller routine must strip off the first (and possibly the last) byte so that it can recognize the character string.

This section contains the description of all $ZBIT function and an example of using $ZBIT functions to turn a character into a bit stream and return a coded value. However, the most appropriate use of these functions may include the formation of checksums, handling of bit-data (say pixels from a scan), or interfacing with a routine that requires bit-oriented arguments.

$ZBITAND()

Performs a logical AND function on two bit strings and returns a bit string equal in length to the shorter of the two arguments (containing set bits in those positions where both of the input strings have set bits). Positions corresponding to positions where either of the input strings have a cleared bit, also have cleared bits in the resulting string.

The format for the $ZBITAND() function is:

```
$ZBITAND(expr1,expr2)
```

- The first expression specifies one of the bit strings that is input to the AND operation.

- The second expression specifies the other bit string that is input to the AND operation.

Example of $ZBITAND()

```
GTM>set BITSTRINGA=$zbitset($zbitset($zbitstr(8,0),2,1),8,1)
; The binary representation of A is 01000001
GTM>set BITSTRINGB=$zbitset($zbitset($zbitstr(8,0),2,1),7,1)
; The binary representation of B is 01000010
GTM>set BITSTRINGAB=$zbitand(BITSTRINGA,BITSTRINGB)
GTM>for i=1:1:8 write $zbitget(BITSTRINGAB,i)
01000000
```

This examples uses $ZBITAND to perform a bitwise AND operation on A and B.

```
A = 01000001
B = 01000010
A bitwise AND B = 0100000
```

$ZBITCOUNT()

Returns the number of ON bits in a bit string.

The format for the $ZBITCOUNT function is:

```
$ZBITCOUNT(expr)
```

- The expression specifies the bit string to examine.

Example of $ZBITCOUNT()

Example:

```
GTM>set BITSTRINGA=$ZBITSET($ZBITSET($ZBITSTR(8,0),2,1),8,1)
; The binary representation of A is 01000001
GTM>set BITSTRINGB=$zbitset($zbitset($zbitstr(8,0),2,1),7,1)
; The binary representation of B is 01000010
GTM>Set BITSTRINGC=$zbitor(BITSTRINGA,BITSTRINGB)
; A OR B=01000011
GTM>write $zbitcount(BITSTRINGA)
2
GTM>write $zbitcount(BITSTRINGB)
2
GTM>write $zbitcount(BITSTRINGC)
3
GTM>
```

This example displays the number of ON bits in BITSTRINGA, BITSTRINGB, and BITSTRINGC.

$ZBITFIND()

Performs the analog of $FIND() on a bit string. It returns an integer that identifies the position after the first position equal to a truth-valued expression that occurs at, or after, the specified starting position.

The format for the $ZBITFIND function is:

```
$ZBITFIND(expr,tvexpr[,intexpr])
```

- The expression specifies the bit string to examine.

- The truth-valued expression specifies the bit value for which $ZBITFIND() searches (1 or 0).

- The optional integer argument specifies the starting position at which to begin the search. If this argument is missing, $ZBITFIND() begins searching at the first position of the string. $ZBIT functions count the first bit as position one (1).

If the optional integer argument exceeds the length of the string, or if the function finds no further bits, $ZBITFIND() returns a zero value.

Examples of $ZBITFIND()

Example:

```
GTM>Set BITSTRINGA=$ZBITSET($ZBITSET($ZBITSTR(8,0),2,1),8,1)
; The binary representation of A is 01000001
GTM>write $zbitfind(BITSTRINGA,1,3)
9
GTM>
```

This example searches for bit value 1 starting from the 3rd bit of BITSTRINGA.

$ZBITGET()

Returns the value of a specified position in the bit string.

The format for the $ZBITGET function is:

```
$ZBITGET(expr,intexpr)
```

- The expression specifies the bit string to examine.

- The integer argument specifies the position in the string for which the value is requested. If the integer argument is negative, zero, or exceeds the length of the bit string, it is rejected with a run-time error. $ZBIT functions count the first bit as position one (1).

Examples of $ZBITGET()

Example:

```
GTM>set BITSTRINGA=$zbitset($zbitset($zbitstr(8,0),2,1),8,1)
; The binary representation of A is 01000001
GTM>for i=1:1:8 write $zbitget(BITSTRINGA,i)
01000001
GTM>
```

This examples uses $ZBITGET() to display the binary representation of A.

$ZBITLEN()

Returns the length of a bit string, in bits.

The format for the $ZBITLEN function is:

```
$ZBITLEN(expr)
```

- The expression specifies the bit string to examine.

Examples of $ZBITLEN()

```
GTM>set BITSTR=$zbitstr(6,1)

GTM>write $zbitlen(BITSTR)
6
GTM>
```

This example displays the length of a bit string of 6 bits.

$ZBITNOT()

Returns a copy of the bit string with each input bit position inverted.

The format for the $ZBITNOT function is:

```
$ZBITNOT(expr)
```

- The expression specifies the bit string whose inverted bit pattern becomes the result of the function.

Examples of $ZBITNOT()

```
GTM>set BITSTRINGA=$zbitset($zbitset($zbitstr(8,0),2,1),8,1)
; The binary representation of A is 01000001
GTM>for i=1:1:8 write $zbitget($zbitnot(BITSTRINGA),i)
10111110
GTM>
```

This example displays inverted bits for all the bits in BITSTRINGA.

$ZBITOR()

Performs a bitwise logical OR on two bit strings, and returns a bit string equal in length to the longer of the two arguments (containing set bits in those positions where either or both of the input strings have set bits). Positions that correspond to positions where neither input string has a set bit have cleared bits in the resulting string.

The format for the $ZBITOR function is:

```
$ZBITOR(expr1,expr2)
```

- The first expression specifies one of the bit strings that is input to the OR operation.

- The second expression specifies the other bit string that is input to the OR operation.

Examples of $ZBITOR()

```
GTM>set BITSTRINGA=$zbitset($zbitset($zbitstr(8,0),2,1),8,1)
; The binary representation of A is 01000001
GTM>set BITSTRINGB=$zbitset($zbitset($zbitstr(8,0),2,1),7,1)
; The binary representation of B is 01000010
GTM>set BITSTRINGC=$zbitor(BITSTRINGA,BITSTRINGB)
; A OR B=01000011
GTM>write BITSTRINGC
C
GTM>
```

This example displays the result of BITSTRINGA bitwise ORed with BITSTRINGB.

$ZBITSET()

Returns an edited copy of the input bit string with a specified bit set to the value of the truth-valued expression.

The format for the $ZBITSET function is:

```
$ZBITSET(expr,intexpr,tvexpr)
```

- The expression specifies the input bit string.

- The integer expression specifies the position of the bit to manipulate. Arguments that are negative, zero, or exceed the length of the bit string produce a run-time error. $ZBIT functions count the first bit as position one (1).

- The truth-valued expression specifies the value to which to set the specified bit (0 or 1).

Examples of $ZBITSET()

```
GTM>set X="A",Y=$extract($zbitset($char(0)_X,3,1),2) zwrite
X="A"
Y="a"
```

This example changes the case of the ASCII letter A to the corresponding lowercase version.

$ZBITSTR()

Returns a bit string of a specified length with all bit positions initially set to either zero or one.

The format for the $ZBITSTR function is:

```
$ZBITSTR(intexpr[,tvexpr])
```

- The integer expression specifies the length of the bit string to return; arguments that exceed the maximum length of 253,952 produce a run-time error.

- The optional truth-valued expression specifies the value to which all bit positions should initially be set (0 or 1). If this argument is missing, the bits are set to zero.

Examples of $ZBITSTR()

```
GTM>set BITSTR=$zbitstr(6,1)
```

This example sets the value of expression BITSTR to 6 bit with all bits set to 1.

$ZBITXOR()

Performs a bitwise exclusive OR on two bit strings, and returns a bit string equal in length to the shorter of the two arguments (containing set bits in those position where either but not both of the input strings have set bits). Positions that correspond to positions where neither or both input string has a set bit have cleared bits in the resulting string.

The format for the $ZBITXOR function is:

```
$ZBITXOR(expr1,expr2)
```

- The first expression specifies one of the bit strings that is input to the XOR operation.

- The second expression specifies the other bit string that is input to the XOR operation.

Examples of $ZBITXOR()

```
GTM>set BITSTRINGA=$zbitset($zbitset($zbitstr(8,0),2,1),8,1) ; The binary representation of A is 01000001
GTM>set BITSTRINGB=$zbitset($zbitset($zbitstr(8,0),2,1),7,1); The binary representation of B is 01000010
GTM>set BITSTRINGC=$zbitxor(BITSTRINGA,BITSTRINGB) ; A XOR B=00000011
GTM>for i=1:1:8 write $zbitget(BITSTRINGC,i)
00000011
GTM>
```

This example displays the result of the bitwise XOR of A and B.

Examples of $ZBIT Functions

Example:

```
ZCRC(X)
    new R,I,J,B,X1,K
    set R=$zbitstr(8,0)
    for I=1:1:$length(X) Set R=$zbitxor(R,$$bitin($A(X,I)))
    quit $$bitout(R)

bitin(X) ;CONVERT A BYTE TO A BIT STRING
    set X1=$zbitstr(8,0)
    for J=1:1:8 set B=X#2,X=X\2 if B set X1=$zbitset(X1,J,1)
    quit X1

bitout(X) ; CONVERT A BITSTRING TO A NUMBER
    set X1=0
    for K=1:1:8 I $zbitget(X,K) set X1=X1+(2**(K-1))
    quit X1
```

This uses several $ZBIT functions to turn a character into a bit stream and return a coded value.

While this example illustrates the use of several of the $ZBIT functions, the following example produces identical results if you need to code the function illustrated above for production.

```
ZCRC(X)
    new R,I,J,B,X1,K
    set R=$zbitstr(8,0)
```

```
    for I=1:1:$length(X) Set R=$zbitxor(R,$char(0)_$extract(X,I))
    quit $ascii(R,2)
```

This example illustrates the use of $Char() to specify the number of invalid bits that exist at the end of the character string. In this case there are zero invalid bits.

$ZCHar()

Returns a string composed of bytes represented by the integer octet values specified in its argument(s).

The format for the $ZCHAR() function is:

```
$ZCH[AR](intexpr[,...])
```

- The integer expression(s) specify the numeric byte value of the byte(s) $ZCHAR() returns.

- GT.M limits the number of arguments to a maximum of 254. $ZCHAR() provides a means of producing byte sequences. In the UTF-8 mode, $ZCHAR() returns a malformed characters for numeric byte values 128 to 255. In the M mode, $ZCHAR() can create valid UTF-8 characters that includes bytes in the range 128-255.

> **Note**
>
> The output of $ZCHAR() for values of integer expression(s) from 0 through 127 does not vary with choice of the character encoding scheme. This is because 7-bit ASCII is a proper subset of UTF-8 character encoding scheme. The representation of characters returned by $ZCHAR() for values 128 through 255 differ for each character encoding scheme.

- When used with $ZASCII(), $ZCHAR() can also perform arithmetic operations on the byte values of the bytes associated with a sequence of octets (8-bit bytes).

Example of $ZCHAR()

Example:

```
GTM>write $zchar(228,184,187,7)
主
GTM>
```

This example WRITEs the byte sequence represented by 主 and signals the terminal bell.

$ZCOLlate()

Returns the transformed representation of the first argument glvn in a normalized form using the alternative transform specified by the second argument intexpr; the return can be used as an operand to the follows (]) or sorts-after (]]) operator such that, if both operands are in the normalized form, the result is independent of alternative collation.

The format for the $ZCOLLATE() function is:

```
$ZCOL[late](glvn,intexpr[,{0|1}])
```

- The subscripted or unsubscripted global or local variable name specifies the key to transform.

- The integer expression specifies the ID of the alternative transform to use.

- The optional third argument specifies whether the transform is to normalized form, by default or if zero (0), or, if one (1), the reverse transform from the normalized to the native form.

Note that because the forward transform is to the GDS global storage format, the reverse transform always shows a global form. This is not material when the result is used for most comparisons, but for some uses the applcation might need to remove the leading up-arrow (^).

Please see the section on $ZATRANSFORM() for a similar alternative.

Example of $ZCOLlate()

Example:

```
GTM>write $zwrite($zcollate("A("""foo""")",0))
"A"_$C(0,255)_"foo"_$C(0,0)
GTM>write $zcollate($zcollate("A("""foo""")",0),0,1)
^A("foo")
GTM>
```

The first WRITE in this example shows the readable form or the value produced by the $ZCOLLATE() revealing the details of how GT.M internally represents this key using default (M) collation. The second WRITE shows how the combination of the transform and reverse transform restores the value to the original representation.

$ZCOnvert()

Returns its first argument as a string converted to a different encoding. The two argument form changes the encoding for case within a character set. The three argument form changes the encoding scheme.

The format for the $ZCONVERT() function is:

```
$ZCO[NVERT](expr1, expr2,[expr3])
```

- The first expression is the string to convert. If the expression contains a code-point value that is not in the character set, $ZCONVERT() generates a run-time error.

- In the two argument form, the second expression specifies a code that determines the form of the result. In the three-argument form, the second expression specifies a code that controls the character set interpretation of the first argument. If the expression does not evaluate to one of the defined codes corresponding to a valid code for the number of available arguments, $ZCONVERT() generates a run-time error.

- The optional third expression specifies the a code that determines the character set of the result. If the expression does not evaluate to one of the defined codes $ZCONVERT() generates a run-time argument. The three-argument form is not supported in M mode.

- The valid (case insensitive) character codes for expr2 in the two-argument form are:

  - U converts the string to UPPER-CASE. "UPPER-CASE" refers to words where all the characters are converted to their "capital letter" equivalents. $ZCONVERT() retains characters already in UPPER-CASE "capital letter" form unchanged.

  - L converts the string to lower-case. "lower-case" refers to words where all the letters are converted to their "small letter" equivalents. $ZCONVERT() retains characters already in lower-case or having no lower-case equivalent unchanged.

  - T converts the string to title case. "Title case" refers to a string with the first character of each word in upper-case and the remaining characters in the lower-case. $ZCONVERT() retains characters already conforming to "Title case" unchanged.

- The valid (case insensitive) codes for character set encoding for expr2 and expr3 in the three-argument form are:

  - "UTF-8"-- a multi-byte variable length Unicode® encoding form.

  - "UTF-16LE"-- a multi-byte 16-bit Unicode® encoding form in little-endian.

  - "UTF-16BE"-- a multi-byte 16-bit Unicode® encoding form in big-endian.

  - "UTF-16"-- a multi-byte 16-bit Unicode® encoding form which uses the same endian level as that of the current system.

> **Note**
>
> When UTF-8 mode is enabled, GT.M uses the ICU Library to perform case conversion. As mentioned in the Theory of Operation section, the case conversion of the strings occurs according to UTF-8 code-point values. This may not be the linguistically or culturally correct case conversion, for example, of the names in the telephone directories. Therefore, application developers must ensure that the actual case conversion is linguistically and culturally correct for their specific needs. The two-argument form of the $ZCONVERT() function in M mode does not use the ICU Library to perform operation related to the case conversion of the strings.

Examples of $ZCONVERT()

Example:

```
GTM>write $zconvert("Happy New Year","U")
HAPPY NEW YEAR
```

Example:

```
GTM>Write $zconvert("HAPPY NEW YEAR","T")
Happy New Year
```

Example:

```
GTM>Set T8="主要雨在西班牙停留在平原"
GTM>Write $Length(T8)
12
GTM>Set T16=$zconvert(T8,"UTF-8","UTF-16LE")
GTM>Write $length(T16)
%GTM-E-BADCHAR, $ZCHAR(129,137,232,150) is not a valid character in the UTF-8 encoding form
GTM>Set T16=$ZCOnvert(T16,"UTF-16LE","UTF-8")
GTM>Write $length(T16)
9
```

In the above example, $LENGTH() function triggers an error because it takes only UTF-8 encoding strings as the argument.

$ZDATA()

Extends $DATA() to reflect the current alias state of the lvn or name argument to identify alias and alias container variables. It treats variables joined through pass-by-reference as well as TP RESTART variables within a transaction as alias variables. However, it does not distinguish nodes having alias containers among their descendants.

In addition to the four standard M results from $DATA(), $ZDATA() returns:

- 100 for an uninitialized alias or alias container

- 101 for an alias or alias container with no descendants

- 111 for an alias or alias container with descendants

Existing $DATA() tests for data and descendants report on alias and alias container variables, as well as other variables in the standard fashion. When an application uses alias and alias container variables $ZDATA() supplies additional information when needed.

Examples of $ZDATA()

Example:

```
GTM>set a=1,*b(1)=a,*c=d
GTM>write $data(a)," ",$zdata(a)
1 101
GTM>write $data(b)," ",$zdata(b)
10 10
GTM>write $data(c)," ",$zdata(c)
0 100
GTM>write $data(d)," ",$zdata(d)
0 100
GTM>write $data(b(1))," ",$zdata(b(1))
1 101
GTM>set b(1,2)=2
GTM>write $data(b(1))," ",$zdata(b(1))
11 111
GTM>write $data(b(1,2))," ",$zdata(b(1,2))
1 1
GTM>
```

$ZDate()

Returns a date and/or time formatted as text based on an argument formatted in the manner of $HOROLOG. For information on the format of $HOROLOG, refer to Chapter 8: "Intrinsic Special Variables" (page 291).

The format for the $ZDATE function is:

```
$ZD[ATE](expr1[,expr2[,expr3[,expr4]]]])
```

- The first expression specifies in $HOROLOG format the date and/or time that $ZDATE() returns in text format. If the output requires only the date or the time, the other piece of the argument that is delimited by a comma (,) may be null.

- The optional second expression specifies a string providing $ZDATE() with a "picture" of the desired output format. If this argument is missing or null, $ZDATE() uses the default format string "MM/DD/YY". If the optional second expression exceeds 64 characters, $ZDATE() generates a run-time error.

- The optional third expression specifies a list of 12 month codes, separated by commas (,), that $ZDATE() uses in formatting text months called for by the "MON" picture, (that is, $ZDATE() outputs $PIECE(expr3,",",month-number) when "MON" appears in the second expression). If this argument is missing or null, $ZDATE() uses three-character English abbreviations for months.

- The optional fourth expression specifies a list of seven day codes, separated by commas (,), which $ZDATE() uses in formatting text days of the week called for by the "DAY" picture, $ZDATE() outputs $PIECE (expr4,",",day-of-week-number) when "DAY" appears in the second expression; if this argument is missing or null, $ZDATE() uses three-character English abbreviations for days of the week.

- $ZDATE() returns 31-Dec-1840 as a date representation of day 0.

$ZDATE() provides an easy and flexible tool for putting M internal date/time ($HOROLOG) formats into more user-friendly formats.

> ⚠️ **Warning**
>
> $ZDATE() generates an error for input date values greater than 31-Dec-999999 (364570088) or less than 01-JAN-1840 (-365) and for time values greater than a second before midnight (86399) or less than 0 (zero).

The Intrinsic Special Variable $ZDATEFORM determines the output format for years. The default value is zero (0), in which case $ZDATE() with one argument (no format specification) uses a "YY" (two digit) format for all years. If $ZDATEFORM is one (1), a "YYYY" (four digit) format is used for years later than 1999. For all other values of $ZDATEFORM, "YYYY" (four digit) format is used for all years. $ZDATEFORM does not affect $ZDATE() when the format argument is specified.

The following table summarizes the usage of $ZDATE() when only first argument is specified.

Value of $ZDATEFORM	$ZDATE() Output Format
0	2 digits
1	4 digits for years 2000 and after 2 digits otherwise (for years ranging between 1840, 1999)
other	4 digits

$ZDATE Format Specification Elements

This section lists the $ZDATE format specification elements. $ZDATE() format specifications must appear in upper case. When any alphabetic characters in format specifications are in lower case, $ZDATE() generates a run-time error.

YY: Outputs the rightmost two digits of the year.

YEAR or YYYY: Outputs the year as a four-digit number.

YYYYYY: Outputs the year as a six-digit number.

MM: Outputs the month as a two-digit zero-filled number between 01 and 12.

MON: Outputs the month as a three-letter abbreviation. (You can modify the output further using expr3).

DD: Outputs the day of the month as a two-digit zero-filled number between 01 and 31.

DAY: Outputs the day of the week as a three-letter abbreviation. (You can modify the output further using expr4).

24: Outputs the hour of the day as a zero-filled number between 00 and 23.

12: Outputs the hour of the day as a zero-filled number between 01 and 12.

60: Outputs the minute of the hour as a zero-filled number between 00 and 59.

SS: Outputs the second of the minute as a zero-filled number between 00 and 59.

AM: Outputs the letters AM and PM depending on the time.

+: Inserts a plus sign (+) in the output string

-: Inserts a minus sign (-) in the output string.

.: Inserts a period (.) in the output string.

,: Inserts a comma (,)in the output string.

/: Inserts a slash (/) in the output string.

:: Inserts a colon (:) in the output string.

;: Inserts a semi-colon (;) in the output string.

*: Inserts an asterisk (*) in the output string.

> **Note**
>
> A blank space inserts a blank space in the output string.

Examples of $ZDATE()

Example:

```
GTM>write $horolog,!,$zdate($H)
62109,60946
01/18/11
GTM>
```

This displays $HOROLOG and then uses $ZDATE() to display today's date. The output shown would appear if today were the eighteenth day of January, 2011.

Example:

```
GTM>write $zdate($H,"DD-MON-YEAR")
18-JAN-2011
GTM>
```

This uses the second argument to specify a text format different from the default.

Example:

```
GTM>set m="Januar,Februar,Marz,April,Mai,Juni,Juli,August,"
GTM>set m=m_"September,October,November,Dezember"
GTM>write $zdate($horolog,"DD-MON-YEAR",m)
18-Januar-2011
GTM>
```

This is similar to the prior example, however it uses the third argument to specify the months in German.

Example:

```
GTM>set d="Dimanche,Lundi,Mardi,Mercredi,Jeudi,Vendredi,Samedi"
GTM>write $zdate($H,"DAY, DD/MM/YY","",d)
Mardi, 18/01/2011
GTM>
```

This example displays the eighteenth of January, however it uses the fourth argument to specify the days of the week in French.

Example:

```
GTM>write !,$zdate($H,"12:60:SS AM")
10:35:51 PM
GTM>
```

This example shows hours, minutes, and seconds in a 12 hour clock with an AM/PM indicator.

Example:

```
GTM>write !,$zdate(",36524","24-60")
10-08
GTM>
```

This example shows hours and minutes on a 24 hour clock. Notice that the first argument must provide the time in the second comma delimiter piece to match $HOROLOG format.

Example:

```
GTM>write $zdateform
0
GTM>write $zdate($H)
01/18/11
GTM>set $zdateform=1
GTM>write $zdate($horolog)
01/18/2011
GTM>write $zdate($horolog,"MM/DD/YY")
01/18/11
```

This example converts the output format for years from the default ("YY") format to the four digit format ("YYYY") using the Intrinsic Special Variable $ZDATEFORM.

Example:

```
GTM>write $zdate(123456789,"DAY MON DD, YYYYYY")
FRI MAR 17, 339854
GTM>
```

This example displays year as a six-digit number.

$ZExtract()

Returns a byte sequence from a given sequence of octets (8-bit bytes).

The format for the $ZEXTRACT function is:

```
$ZE[XTRACT](expr[,intexpr1[,intexpr2]])
```

• The expression specifies a sequence of octets (8-bit bytes) from which $ZEXTRACT() derives a byte sequence.

- The first optional integer expression (second argument) specifies the starting byte position in the byte string. If the starting position is beyond the end of the expression, $ZEXTRACT() returns an empty string. If the starting position is zero (0) or negative, $ZEXTRACT() starts at the first byte position in the expression; if this argument is omitted, $ZEXTRACT() returns the first byte. $ZEXTRACT() numbers byte positions starting at one (1) (the first byte of a sequence of octets (8-bit bytes) is at position one (1)).

- The second optional integer expression (third argument) specifies the ending byte position for the result. If the ending position is beyond the end of the expression, $ZEXTRACT() stops with the last byte of the expression. If the ending position precedes the starting position, $ZEXTRACT() returns null. If this argument is omitted, $ZEXTRACT() returns one byte.

- $ZEXTRACT() provides a tool for manipulating strings based on byte positions.

- As $ZEXTRACT() operates on bytes, it can produce a string that is not well-formed according to the UTF-8 character set.

Examples of $ZEXTRACT()

Example:

```
GTM>Set A="主要雨在西班牙停留在平原"

GTM>For i=0:1:$zlength(A)
GTM>write !,$zascii($zextract(A,i)),"|"
GTM>
```

This example displays the numeric byte sequence of the sequence of octets ("主要雨在西班牙停留在平原").

$ZFind()

Returns an integer byte position that locates the occurrence of a byte sequence within a sequence of octets(8-bit bytes).

The format of the $ZFIND function is:

```
$ZF[IND](expr1,expr2[,intexpr])
```

- The first expression specifies the sequence of octets (8-bit bytes) in which $ZFIND() searches for the byte sequence.

- The second expression specifies the byte sequence for which $ZFIND() searches.

- The optional integer expression identifies the starting byte position for the $ZFIND() search. If this argument is missing, zero (0), or negative, $ZFIND() begins to search from the first position of the sequence of octets (8-bite bytes).

- If $ZFIND() locates the byte sequence, it returns the position after its last byte. If the end of the byte sequence coincides with the end of the sequence of octets (expr1), it returns an integer equal to the byte length of the expr1 plus one ($L(expr1)+1).

- If $ZFIND() does not locate the byte sequence, it returns zero (0).

- $ZFIND() provides a tool to locate byte sequences. The ([ ) operator and the two-argument $ZLENGTH() are other tools that provide related functionality.

Examples

Example:

```
GTM>write $zfind("主要雨",$zchar(187))
4
```

```
GTM>
```

This example uses $ZFIND() to WRITE the position of the first occurrence of the numeric byte code 150. The return of 3 gives the position after the "found" byte.

Example:

```
GTM>write $zfind("新年好",$zchar(229),5)
8
GTM>
```

This example uses $ZFIND() to WRITE the position of the next occurrence of the byte code 229 starting in byte position five.

Example:

```
GTM>set t=1 for  set t=$zfind("新年好",$zchar(230,150,176),t) quit:'t  write !,t
4
GTM>
```

This example uses a loop with $ZFIND() to locate all the occurrences of the byte sequence $ZCHAR(230,150,176) in the sequence of octets ("新年好"). The $ZFIND() returns 4 giving the position after the occurrence of byte sequence $ZCHAR(230,150,176).

$ZGetjpi()

Returns job or process information of the specified process. The format for the $ZGETJPI function is:

```
$ZGETJPI(expr1,expr2)
```

- expr1 identifies the PID of the target job. If expr1 is an empty string (""), $ZGETJPI() returns information about the current process.

- expr2 specifies the item keyword identifying the type of information returned; keywords may be upper, lower, or mixed-case. The keywords are as follows:

ZGETJPI()	
Keywords	Data returned
ISPROCALIVE	Determines whether the specified process is alive.
CPUTIM	Total process and child CPU time used in hundredths of a second.
CSTIME	System time of child processes
CUTIME	User time of child processes
STIME	Process system time
UTIME	Process user time

- Note that the $ZGETJPI() retrieves process time measurements (CPUTIM, CSTIME, CUTIME, STIME, and UTIME) only of the current process ($JOB). The "child" process time includes ZSYSTEM and PIPE device sub-processes (only after the PIPE CLOSEs), but excludes processes created by the JOB command.

- $ZGETJPI() provides a tool for examining the characteristics of a UNIX process. Accessing information about processes belonging to other users requires certain UNIX privileges. Consult your system manager if you require additional privileges.

Examples

Example:

```
GTM>write $zgetjpi(1975,"isprocalive")
1
GTM>
```

This uses $ZGETJPI() to determine whether process 1975 is alive.

Example:

```
GTM>set t=$zgetjpi("","cputim")
GTM>do ^bench write $zgetjpi("","cputim")-t
1738
GTM>
```

This uses $ZGETJPI() to measure the actual CPU time, measured in hundredths of a second, consumed by running the ^bench routine.

$ZJOBEXAM()

Places ZSHOW <code> output into a file and returns the absolute file name. The format for the $ZJOBEXAM function is:

```
$ZJOBEXAM([expr1][,expr2])
```

- The optional first expression is the output specification. It can be a file directory or a file name. $ZJOBEXAM() pre-processes the first expression to create a file specification as the target for the ZSHOW command output. The preprocessing is equivalent to $ZPARSE(), as illustrated by the following M code:

```
GTM>set cntr=1,deffn="GTM_JOBEXAM.ZSHOW_DMP_"_$JOB_"_"_cntr
GTM>w $zparse(deffn)
/path/to/GTM_JOBEXAM.ZSHOW_DMP_49878_1
GTM>w $zjobexam()
/path/to/GTM_JOBEXAM.ZSHOW_DMP_49878_1
GTM>
```

- The optional second expression evaluates to a string containing a list of one or more ZSHOW information codes that determine the type of information. If the second expression is missing or empty, GT.M operates as it was a "*" and produces all context. Note that using an explicit list may be useful in limiting the exposure of sensitive information. For more information, refer to to "ZSHOW Information Codes" (page 193).

The return value serves as a way to save, to notify others of the exact location of the output or to open the file for further processing. GT.M logs the JOBEXAMDONE message to the operator log for each $ZJOBEXAM() which includes the full file specification.

The $ZJOBEXAM()does not trigger error processing except when there is a problem storing its return value, so no error is reported to the process until after any dump is complete. In the event of any error encountered during the $ZJOBEXAM(), GT.M sends an appropriate message to operator log facility and returns control to the caller. Note that this special error handling applies only to the $ZJOBEXAM(), and is not a property of the $ZINTERRUPT interrupt handler, which uses $ZJOBEXAM() by default.

$ZJOBEXAM() dump files contain the context of a process at the time the function executes. Placement and management of these files should consider their potential size and security implications.

Examples of $ZJOBEXAM()

Example:

```
GTM>set x=$zjobexam()
GTM>write x
/tmp/GTM_JOBEXAM.ZSHOW_DMP_383974_1
GTM>set x=$zjobexam("jobexam.out")
GTM>write x
/tmp/jobexam.out
GTM>set x=$zjobexam("jobexam_ISV.out","I")
GTM>write x
/tmp/jobexam_ISV.out
GTM>
```

In the first two examples, the output file contains zshow "*" information. The 3rd example saves zshow "I" (ISVs only) information in the output file.

$ZJustify()

Returns a formatted and fixed length byte sequence.

The format for the $ZJUSTIFY() function is:

```
$ZJ[USTIFY](expr,intexpr1[,intexpr2])
```

- The expression specifies the sequence of octets formatted by $ZJUSTIFY().

- The first integer expression (second argument) specifies the minimum size of the resulting byte sequence.

- If the first integer expression is larger than the length of the expression, $ZJUSTIFY() right justifies the expression to a byte sequence of the specified length by adding leading spaces. Otherwise, $ZJUSTIFY() returns the expression unmodified unless specified by the second integer argument.

- The behavior of the optional second expression (third argument) for $ZJUSTIFY() is the same at $JUSTIFY(). For more information, refer to "$Justify()" (page 224).

- When the second argument is specified and the first argument evaluates to a fraction between -1 and 1, $ZJUSTIFY() returns a number with a leading zero (0) before the decimal point (.).

- $ZJUSTIFY() fills a sequence of octets to create a fixed length byte sequence. However, if the length of the specified expression exceeds the specified byte size, $ZJUSTIFY() does not truncate the result (although it may still round based on the third argument). When required, $ZEXTRACT() performs truncation.

- $ZJUSTIFY() optionally rounds the portion of the result after the decimal point. In the absence of the third argument, $ZJUSTIFY() does not restrict the evaluation of the expression. In the presence of the third (rounding) argument, $JUSTIFY() evaluates the expression as a numeric value. The rounding algorithm can be understood as follows:

  - If necessary, the rounding algorithm extends the expression to the right with 0s (zeros) to have at least one more digit than specified by the rounding argument.

  - Then, it adds 5 (five) to the digit position after the digit specified by the rounding argument.

  - Finally, it truncates the result to the specified number of digits. The algorithm rounds up when excess digits specify a half or more of the last retained digit and rounds down when they specify less than a half.

Examples of $ZJUSTIFY()

Example:

```
GTM>write "123456789012345",! write $zjustify("新年好",15),!,$zjustify("新年好",5)
123456789012345
      新年好
新年好
GTM>
```

This example uses $ZJUSTIFY() to display the sequence of octets represented by "新年好" in fields of 15 space octets and 5 space octets. Because the byte length of "新年好" is 9, it exceeds 5 spaces, the result overflows the specification.

$ZLength()

Returns the length of a sequence of octets measured in bytes, or in "pieces" separated by a delimiter specified by one of its arguments.

The format for the $ZLENGTH() function is:

```
$ZL[ENGTH](expr1[,expr2])
```

- The first expression specifies the sequence of octets that $ZLENGTH() "measures".

- The optional second expression specifies the delimiter that defines the measure; if this argument is missing, $ZLENGTH() returns the number of bytes in the sequence of octets.

- If the second argument is present and not null, $ZLENGTH() returns one more than the count of the number of occurrences of the second byte sequence in the first byte sequence; if the second argument is null , the M standard for the analogous $LENGTH() dictates that $ZLENGTH() returns a zero (0).

- $ZLENGTH() provides a tool for determining the lengths of a sequence of octets in two ways--bytes and pieces. The two argument $ZLENGTH() returns the number of existing pieces, while the one argument returns the number of bytes.

Examples of $ZLength()

Example:

```
GTM>write $zlength("主要雨在西班牙停留在平原")
36
GTM>
```

This uses $ZLENGTH() to WRITE the length in bytes of the sequence of octets "主要雨在西班牙停留在平原".

Example:

```
GTM>set x="主"_$zchar(63)_"要"_$zchar(63)_"雨"
GTM>write $zlength(x,$zchar(63))
3
GTM>
```

This uses $ZLENGTH() to WRITE the number of pieces in a sequence of octets, as delimited by the byte code $ZCHAR(63).

Example:

```
GTM>set x=$zchar(63)_"主"_$zchar(63)_"要"_$zchar(63)_"雨"_$zchar(63)"
```

```
GTM>write $zlength(x,$zchar(63))
5
GTM>
```

This also uses $ZLENGTH() to WRITE the number of pieces in a sequence of octets, as delimited by byte code $ZCHAR(63). Notice that GT.M counts both the empty beginning and ending pieces in the string because they are both delimited.

$ZMessage()

Returns a message string associated with a specified status code .

The format for the $ZMESSAGE function is:

```
$ZM[ESSAGE](intexpr)
```

- The integer expression specifies the status code for which $ZMESSAGE() returns error message text .

$ZMESSAGE() provides a tool for examining the message and/or mnemonic associated with a particular message code as reported in $ZSTATUS.

The $ZSTATUS Intrinsic Special Variable holds the message code and the message of the last non-Direct Mode GT.M error. For more information on $ZSTATUS, refer "Intrinsic Special Variables".

Examples of $ZMESSAGE()

Example:

```
GTM>write $zmessage(150373210)
%GTM-E-DIVZERO, Attempt to divide by zero
GTM>
```

This uses $ZMESSAGE() to display the message string corresponding to code 150373210.

$ZPARSE()

Expands a file name to a full pathname and then returns the full pathname or one of its fields (directory, name, or extension).

The format for the $ZPARSE function is:

```
$ZPARSE(expr1[,expr2[,expr3[,expr4[,expr5]]]])
```

- The first expression specifies the file name; if the file name is not valid, $ZPARSE() returns a null string; if the file name contains a wildcard (* and/or ?), $ZPARSE() returns a file name containing the wildcard(s).

- The optional second expression specifies the field of the pathname that $ZPARSE() returns; if this argument is missing or null, $ZPARSE() returns a full pathname constructed using default values in place of any fields missing for directory, file and extension.

- The optional third and fourth expressions specify default values to use during file name expansion for missing fields (directory, name, or extension), if any, in the original file name. For any field missing in the original file name specified in expr1, $ZPARSE() will attempt to substitute the corresponding field from expr3; if that field is not present in expr3, $ZPARSE() will attempt to use the corresponding field from expr4.

- If the file extension is missing from all three of expr1, expr3, and expr4, $ZPARSE() will return a null string for the corresponding field. If the file or directory is missing from all three of expr1, expr3, and expr4, $ZPARSE() will substitute the information from your current working directory.

- The optional fifth expression specifies the mode or type of parse that $ZPARSE() performs.

$ZPARSE() provides a tool for verifying that a file name is syntactically correct, for examining specific fields of a file name, and for filling in missing pieces in a partial specification based on a hierarchy of defaults. For information about determining whether a file exists, see "$ZSEARCH()" (page 282).

$ZPARSE() arguments, after the first, are optional. If you use no other arguments, a single argument is sufficient. However, if you use selected arguments $ZPARSE() requires that null strings ("") be filled in for the unspecified arguments.

The acceptable keywords for the second argument are:

"DIRECTORY": Directory name

"NAME": File name (excluding file extension)

"TYPE": File type extension

The keywords may be entered in either upper or lower case. Variables that evaluate to these strings and indirection are acceptable for argument two. When the keywords themselves appear as string literals, they must be enclosed in quotation marks (" ").

The following guidelines must be followed in constructing arguments one, three and four:

- Directory specifications must end in a slash; anything after the final slash in the directory specification is assumed to be part of the name specification.

- A file name with an extension must include at least one character to the left of the period (.). Thus, "/user/.login" refers to the file named ".login", while "/usr/taxes.c" refers to a file named "taxes" with the extension "c". If a file name includes more than one period, the extension includes all letters to the right of the rightmost period.

The keywords for the fifth argument $ZPARSE() are:

NULL (""): Returns a full file-specification or device

"SYNTAX_ONLY": Disables checking for the existence of the directory or device.

Examples of $ZPARSE()

Example:

```
GTM>write $zparse("test","","/usr/work/","dust.lis")
/usr/work/test.lis
GTM>
```

This uses $ZPARSE() to demonstrate defaulting using the third and fourth arguments. The result gets the directory field from the third expression, the name from the first expression, and the type from the fourth expression.

Example:

```
GTM>r!,"file :",f w ?20,$zparse(f,"directory")
file: test.list /usr/work/
```

```
GTM>
```

This uses $ZPARSE() to display the directory for the file name entered as input at the prompt file: , in this case, the current working directory.

Example:

```
$ cd /usr/work/me
$ $gtm
GTM>write $zparse("test","","x.list","y.c")/usr/work/me/test.lis
GTM>write $zparse("test","","/usr/work/","/dev/y.c")/usr/work/test.c
GTM>write $zparse("test","","/usr/work","/dev/y.c")/usr/test.c
GTM>
```

This example illustratest the use of the third and fourth arguments to $ZPARSE(). In the first statement, the first argument has no directory or extension field, so $ZPARSE() substitutes the extension field from the third argument. Since neither the third nor fourth argument specifies a directory, and because the fourth argument does not contain any fields that are not present in the third argument, the fourth argument is not used.

In the second statement, the first argument to $ZPARSE() is again missing both the directory and extension. In this instance, $ZPARSE() uses the directory specified in the third argument and, becuase neither the first nor third argument specifies a file extension, $ZPARSE() uses the file extension from the fourth argument.

In the third statement, because "/usr/work" does not end with a backward slash (/), $ZPARSE() interprets the substring "work" as a file name. Then, $ZPARSE() substitutes "/usr/" for the directory missing in the first argument and substitutes ".c" from the fourth argument for the extension missing from both the first and third arguments.

Example:

```
$ cd /usr/work/me
$ /usr/lib/fis-gtm/V5.4-002B_x86/gtm
GTM>For i="DIRECTORY","NAME","TYPE","" Write $ZPARSE("test.m",i),!
/usr/work/me/
test
.m
/usr/work/me/test.m
GTM>
```

This example illustrates the output produced for each of the possible values for the second argument.

$ZPIece()

Return a sequence of bytes delimited by a specified byte sequence made up of one or more bytes.

The format for the $ZPIECE function is:

```
$ZPI[ECE](expr1,expr2[,intexpr1[,intexpr2]])
```

- The first expression specifies the sequence of octets from which $ZPIECE() takes its result.

- The second expression specifies the delimiting byte sequence that determines the piece "boundaries"; if this argument is a null string, $ZPIECE() returns a null string.

- If the second expression does not appear anywhere in the first expression, $ZPIECE() returns the entire first expression (unless forced to return null by the second integer expression).

- The optional first integer expression (third argument) specifies the beginning piece to return; if this argument is missing, $ZPIECE() returns the first piece.

- The optional second integer expression (fourth argument) specifies the last piece to return. If this argument is missing, $ZPIECE() returns only one piece unless the first integer expression is zero (0) or negative, in which case it returns a null string. If this argument is less than the first integer expression, $ZPIECE() returns null.

- If the second integer expression exceeds the actual number of pieces in the first expression, $ZPIECE() returns all of the expression after the delimiter selected by the first integer expression.

- The $ZPIECE() result never includes the "outside" delimiters; however, when the second integer argument specifies multiple pieces, the result contains the "inside" occurrences of the delimiter.

- $ZPIECE() provides a tool for efficiently using values that contain multiple elements or fields, each of which may be variable in length.

- Applications typically use a single byte for a $ZPIECE() delimiter (second argument) to minimize storage overhead, and increase efficiency at run-time. The delimiter must be chosen so the data values never contain the delimiter. Failure to enforce this convention with edit checks may result in unanticipated changes in the position of pieces within the data value. The caret symbol (^), backward slash (\), and asterisk (*) characters are examples of popular visible delimiters. Multiple byte delimiters may reduce the likelihood of conflict with field contents. However, they decrease storage efficiency, and are processed with less efficiency than single byte delimiters. Some applications use control characters, which reduce the chances of the delimiter appearing in the data but sacrifice the readability provided by visible delimiters.

- A SET command argument can have something that has the format of a $ZPIECE() on the left-hand side of its equal sign (=). This construct permits easy maintenance of individual pieces within a sequence of octets. It also can be used to generate a byte sequence of delimiters. For more information on SET $ZPIECE(), refer to SET in the "Commands" chapter.

Examples of $ZPIECE()

Example:

```
GTM>for i=0:1:3 write !,$zpiece("主"_$zchar(64)_"要",$zchar(64),i),"|"
|
主|
要|
|
GTM>
```

This loop displays the result of $ZPIECE(), specifying $ZCHAR(64) as a delimiter, a piece position "before," first and second, and "after" the sequence of octets.

Example:

```
GTM>for i=-1:1:3 write !,$zpiece("主"_$zchar(64)_"要",$zchar(64),i,i+1),"|"
|
主|
主@要|
要|
|
GTM>
```

This example is similar to the previous example except that it displays two pieces on each iteration. Notice the delimiter () in the middle of the output for the third iteration, which displays both pieces.

Example:

```
For p=1:1:$ZLength(x,"/") Write ?p-1*10,$ZPIece(x,"/",p)
```

This loop uses $ZLENGTH() and $ZPIECE() to display all the pieces of x in columnar format.

Example:

```
GTM>Set $piece(x,$zchar(64),25)="" write x
@@@@@@@@@@@@@@@@@@@@@@@@
```

This SETs the 25th piece of the variable x to null, with delimiter $ZCHAR(64). This produces a byte sequence of 24 at-signs (@) preceding the null.

$ZPEEK()

Provides a way to examine memory in the current process address space. Use of this function requires information ▶ about GT.M internals, which may change from release to release. Contact FIS support for information on techniques for using $ZPEEK() in largely release independent ways.

The $ZPEEK() function returns the contents of the memory requested as a string depending on the requested (or defaulted) formatting.

The format of the $ZPEEK() function is:

```
$ZPEEK("mnemonic[:argument]",offset,length[,format])
```

▼

- **mnemonic** specifies the memory area $ZPEEK() is to access. Some mnemonics have arguments separated from the mnemonic by a colon (":"). The mnemonics are case independent. Possible mnemonics, their possible abbreviations and their arguments are:

  - *CSA[REG]* - returns a value from the sgmnt_addrs (process private) control block. Takes a case independent region name as an argument.

  - *FH[REG]* - returns a value from the sgmnt_data (shared file header) control block. Takes a case independent region name as an argument..

  - *GDR[REG]* - returns a value from the gd_region (process private) control block. Takes a case independent region name as an argument.

  - *GLF[REPL]* - returns a value from the jnlpool.gtmsrc_lcl_array[n] control block. Takes a numeric index (n) as an argument.

  - *GRL[REPL]* - returns a value from the recvpool.gtmrecv_local control block. No argument allowed. Only available when run on a non-primary instance.

  - *GSL[REPL]* - returns a value from the jnlpool.gtmsource_local_array[n] control block. Takes a numeric index (n) as an argument.

  - *JBF[REG]:region[* -obtains fields in shared jnl_buffer structure.

  - *JNL[REG]:region[* - obtains fields in the jnl_private_control structure.

- *JPC[REPL]* - returns a value from the jnlpool.jnlpool_ctl control block. No argument allowed.

- *NL[REG]* - returns a value from the node_local (shared) control block. Takes a case independent region name as an argument.

- *NLREPL* - returns a value from the node_local (shared) control block associated with replication. No argument allowed.

- *PEEK* - returns a value based on the supplied argument. Argument is the base address of the value to obtain in 0xhhhhhhh format where the h's are hex digits.

- *RIH[REPL]* - returns a value from the jnlpool.repl_inst_filehdr control block. No argument allowed.

- *RPC[REPL]* - returns a value from the recvpool.recvpool_ctl control block. No argument allowed. Only available when run on a non-primary instance.

- *UHC[REPL]* - returns a value from the recvpool.upd_helper_ctl control block. No argument allowed. Only available when run on a non-primary instance.

- *UPL[REPL]* - returns a value from the recvpool.upd_proc_local control block. No argument allowed. Only available when run on a non-primary instance.

- **offset** (first integer expression) is a numeric value that specifies the offset from the address supplied or implied by the the mnemonic and argument. Specifying a negative offset results in a BADZPEEKARG error. Specifying too large an offset such that unavailable memory is specified results in a BADZPEEKRANGE error.

- **length** (second integer expression) is a numeric value that specifies the length of the field to obtain. Specifying a negative legnth results in a BADZPEEKARG error. Specifying a length that exceeds the maximum string length results in a MAXSTRLEN error. Specifying too large a length such that unavailable memory is specified results in a BADZPEEKRANGE error.

- **format** is an optional single case independent character formatting code for the retrieved data. The formatting codes are:

  - *C* : returns a character representations of the memory locations; this is the DEFAULT if the fourth argument is not specified.

  - *I* : returns a signed integer value - negative values have a preceding minus sign (-); the length can be 1, 2, 4, or 8 bytes.

  - *U* : returns an unsigned integer value - all bits are part of the numeric value; the length can be 1, 2, 4, or 8 bytes.

  - *S* : returns a character representation of the memory locations and the first NULL character found terminates the returned string; that is: the specified length is a maximum.

  - *T*: Selects a $HOROLOG format for a field of 4 or 8 bytes which is intended for use on fields in UNIX time format (seconds since 01/01/1970)

  - *X* : returns a hexadecimal value as 0xXXXXXX where XXXXXX is twice the specified length in bytes, so requested length 1 returns 0xXX and length 4 returns 0xXXXXXXXX; the length can be 1, 2, 4, or 8 bytes.

  - *Z* : returns a hexadecimal representation of the memory locations as 'X' does, without regard to endianness, and with no length restriction other than max string length.

  - $ZPEEK() function generates an UNDEF error when VIEW UNDEF is not set and a format parameter is specified but is undefined.

Notes

- $ZPEEK() has no UTF-8 checking. It is possible for values returned by the 'C' and 'S' codes to have invalid UTF-8 values in them. Take care when processing values obtained by these codes to either use "VIEW NOBADCHAR" when dealing with such values and/or use the $Zxxx() flavors of functions like $ZPIECE(), $ZEXTRACT(),etc which also do not raise BADCHAR errors when encountering invalid UTF-8 encoded strings.

- Note that $ZPEEK() with 8 byte numeric formatting can return numeric string values that exceed GT.M's current limit of 18 digits of precision. If the values are used as strings, the extra digits are preserved, but if used arithmetically, the lower precision digits can be lost.

- When values from replication structures are requested and the structures are not available due to replication not running or, in the case of the gtmrecv.* control block base options, if not running on a non-primary instance where the gtmrecv.* control are available, a ZPEEKNOREPLINFO error is raised.

- The JNL[REG] and JBL[REG] mnemonics and characteristics are defined by the running the GTMDefinedTypesInit.m utility, which produces a cross-index in the form:

  ```
  gtmtypfldindx(<structure-name>.<field-mnemonic>)=<n>
  ```

  where gtmtypes(<structure-name>,<n>,*) nodes contain the field characteristics

- When $ZGBLDIR is set to a global directory specifying a replication instance and replication has started, the Replication Journal Pool for that instance becomes the source of data reported by $ZPEEK(). If a global directory does not specify a Replication Instance, the gtm_repl_instance environment variable determines the Instance for its replicated regions. The mnemonics GLF, GSL, JPC, NLREPL, and RIH require a jnlpool. $VIEW("JNLPOOL") returns the Instance for the current Replication Jounal Pool.

$ZPrevious()

The $ZPREVIOUS function returns the subscript of the previous local or global variable name in collation sequence within the array level specified by its argument. When $ZPREVIOUS() has an unsubscripted argument, it returns the previous unsubscripted local or global variable name in collating sequence.

The $ZPREVIOUS function provides compatibility with some other M implementations. The M Development Committee chose to implement this functionality with the optional second -1 argument of $ORDER(). Therefore, when a design requires this functionality $ORDER() has the advantage over $ZPREVIOUS of being part of the M standard.

The format for the $ZPREVIOUS function is:

```
$ZP[REVIOUS](glvn)
```

- The subscripted or unsubscripted global or local variable name specifies the node prior to which $ZPREVIOUS() searches backwards for a defined node with data and/or descendants. The number of subscripts contained in the argument implicitly defines the array level.

- If $ZPREVIOUS() finds no node at the specified level before the specified global or local variable, it returns a null string.

- If the last subscript in the subscripted global or local variable name is null, $ZPREVIOUS() returns the last node at the specified level.

$ZPREVIOUS() is equivalent to $ORDER() with a second argument of -1.

$ZSOCKET()

Returns information about a SOCKET device and its attached sockets. The format of the $ZSOCKET() function is:

```
$ZSOCKET(expr1,expr2[,[expr3][,expr4]])
```

- The first expression specifies the SOCKET device name; an empty string returns the same result as the current device ($IO). If the first expression is not specified, $ZSOCKET() returns information about sockets in the socketpool. Specifying a device other than a SOCKET device for the $ZSOCKET() function produces a ZSOCKETNOTSOCK error. When a GT.M process starts with different sockets for input and output on $PRINCIPAL, $ZSOCKET() accepts $ZPIN or $ZPOUT as its first argument and supplies information on the input or output side, respectively. The following is an example of getting the handles for the $PRINCIPAL input and output socket devices.

```
set handlein=$ZSOCKET($ZPIN,"SOCKETHANDLE",0)
set handleout=$ZSOCKET($ZPOUT,"SOCKETHANDLE",0)
```

For more information, refer to "$ZPIN" (page 317) and "$ZPOUT" (page 317).

- The second expression specifies a keyword identifying the type of information returned and the optional third expression usually specifies the index (starting at zero) of a socket attached to the device; if the index is outside the range of attached sockets, $ZSOCKET() returns an empty string. If the third expression is not specified, $ZSOCKET() returns information about the current socket. Using an invalid keyword produces a ZSOCKETATTR error. The fourth expression specifies an individual delimiter when the second expression specifies DELIMITER. For more information, see the following table. Note that changes to the socket collection for a SOCKET device using OPEN, CLOSE, USE :ATTACH, or USE :DETACH may change the index for a socket.

Keyword	Arguments	Returns
BLOCKING	index	1 (TRUE) for blocking otherwise 0 (FALSE) for non-blocking WRITEs
CURRENTINDEX		The index (starting at zero) of the current socket for the SOCKET device.
DELIMITER	index[, delimiter]	If only index is specified, the number of delimiters. If delimiter is also specified, selects which delimiter to return. The first delimiter is zero.
DESCRIPTOR	index	The OS socket descriptor for the socket.
HOWCREATED	index	LISTEN, CONNECT, ACCEPTED, PRINCIPAL, or PASSED ACCEPTED indicates a connection created from a LISTENing socket. PRINCIPAL indicates that the socket is the $PRINCIPAL of the process.

Keyword	Arguments	Returns
		PASSED indicates a socket passed by WRITE /ACCEPT.
INDEX	handle	The current index of the socket named by handle.
IOERROR	index	1 (TRUE) if IOERROR=TRAP otherwise 0 (FALSE).
KEEPALIVE	index	A non zero value if SO_KEEPALIVE is enabled.
KEEPCNT	index	The value of TCP_KEEPCNT.
KEEPIDLE	index	The value of TCP_KEEPIDLE in seconds.
KEEPINTVL	index	The value of TCP_KEEPINTVL in seconds.
LOCALADDRESS	index	The address of the local side of the socket. For TCP sockets: the IPv6 or IPv4 numeric address. For LOCAL sockets: the path.
LOCALPORT	index	The numeric port of the local side of a TCP socket.
MOREREADTIME	index	The value of the MOREREADTIME device parameter if it was specified, otherwise an empty string.
NUMBER		The number of sockets in the SOCKET device.
OPTIONS	index	a string of the "OPTIONS" (page 405) previously specified for the selected socket. The string may not exactly match the string originally specified but has the same meaning.
PARENT	index	If the socket was created from a LISTENing socket: the handle of the LISTENing socket.
PROTOCOL	index	TCP, TCP6, or LOCAL
REMOTEADDRESS	index	The address of the remote side of the socket. For TCP sockets: the IPv6 or IPv4 numeric address. For LOCAL sockets: the path.
REMOTEPORT	index	The numeric port of the remote side of a TCP socket.
SNDBUF	index	Size of the OS send buffer in bytes (SO_SNDBUF).
SOCKETHANDLE	index	The handle for the selected socket.
STATE	index	One of LISTENING, CONNECTED, BOUND, or CONNECTINPROGRESS

Keyword	Arguments	Returns
TLS	index[,expr4]	If the selected socket is using TLS, a string of the form: 1,{SERVER|CLIENT}[,tlsid], where the optional tlsid comes from the WRITE /TLS which enabled TLS on the socket; otherwise an empty string. See the following table for a description of all options for the fourth expression for the TLS keyword.
ZBFSIZE	index	Size of the GT.M buffer in bytes.
ZFF	index	The value of the ZFF device parameter.
ZIBFSIZE	index	Size of the OS buffer in bytes (SO_RCVBUF).
ZDELAY	index	1 if Nagle algorithm enabled, otherwise 0.

Note

The getsockopt() keywords (KEEPALIVE, KEEPCNT, KEEPIDLE, KEEPINTVL, SNDBUF, and ZIBFSIZE return two values if the value previously specified with the "OPTIONS" (page 405) or "ZIBFSIZE" device parameter doesn't match the system's current value. The two values are separated by a semicolon (";"):"uservalue;systemvalue".

The following table describes the values for the fourth expression for the TLS keyword.

expr4 (TLS)	Description
SESSION	Returns information related to SSL sessions including information about renegotiations. Here is an example: **|S:RENSEC:1,RENTOT:1,SESSID:<SESSID>, SESEXP:Thu Jun 4 21:07:11 2015** **"|S:"** denotes this piece contains session information, **"RENSEC:"** indicates whether secure renegotiation is available (1) or not (0), **"RENTOT:"** gives the current total number of renegotiations done on this socket, **"SESSID:"** shows the session id in hexadecimal, and **"SESEXP:"** indicates when the session expires respresented as time in the local time zone.
OPTIONS	the hexadecimal representation of the ssl-options selected by the combination of the OpenSSL defaults, options set by the GT.M TLS plugin, and options specified in the gtmcrypt_config configuration file prefixed by "O:", a comma, and the verify mode as two hexadecimal digits. Here is an example: **|O:0000000001520004,01** The values for the SSL_OP options and verify modes are defined in the include/openssl/ssl.h file provided by the OpenSSL development package. Note: the TLS reference implementation plug-in disables protocols prior to TLSv1.2.
CIPHER	The SSL protocol version prefixed by "P:" and the algorithm negotiated between the server and client prefixed by "C:". Here is an example: **|P:TLSv1.2|C:DHE-RSA-AES256-SHA**

expr4 (TLS)	Description
ALL	returns all available information. Here is an example: **|P:TLSv1.2|C:AES256-GCM-SHA384|O:0000000001020004,01| S:RENSEC:1,RENTOT:0,SESEXP:Mon Jun 22 23:58:09 2015**

$ZSYSLOG()

Sends its string parameter to the system log and always returns TRUE (1). The text appears in the syslog with the same format as any other GT.M syslog message (that is, in the user.info log with GTM-MUMPS[pid]" or "GTM-MUPIP[pid]" prefix along with instance information where appropriate). The format of the $ZSYSLOG function is:

```
$ZSYSLOG(expr)
```

$ZQGBLMOD()

The $ZQGBLMOD function enables an application to determine whether it can safely apply a lost transaction to the database. A lost transaction is a transaction that must be rolled off a database to maintain logical multisite consistency. $ZQGBLMOD() always applies to data-level (level-0) nodes.

The format for the $ZQGBLMOD function is:

```
$ZQGBLMOD(gvn)
```

- The subscripted or non-subscripted global variable name (gvn) specifies the target node.

- A return value of zero (0) means the value of the global variable has not changed since the last synchronization of the originating and replicating instances.

- A return value of one (1) means the value of the global variable may have changed since the last synchronization of the originating and replicating instance.

$ZQGBLMOD function produces an error if you submit an argument that is not a global variable name.

Internally, $ZQGBLMOD (gvn) compares the GT.M transaction number in the database block in which the global variable name is (or would be) stored with the value in the Zqgblmod_Trans field stored in the database file header.

For example, if x is the transaction number of the level-0 database block in which gvn resides, and y is the value of Zqgblmod_Trans of region reg containing gvn, then the following is true:

- If x <= y, no transaction modified the level-0 database block z in which gvn resides since the originating and replicating instances synchronized with each other. $ZQGBLMOD() returns a zero (0).

- If x > y, some transaction modified z, but not necessarily gvn, after the originating and replicating instances synchronized with each other. $ZQGBLMOD() returns a one (1).

If a transaction is a lost transaction that has been rolled back and it is determined that for all the M globals set and killed in the transaction $ZQGBLMOD() is zero (0), it is probably safe to apply the updates automatically. However, this determination of safety can only be made by the application designer and not by GT.M. If the $ZQGBLMOD() is one (1) for any set or kill in the transaction, it is not safe to apply the update.

> **Note**
>
> The test of $ZQGBLMOD() and applying the updates must be encapsulated inside a GT.M transaction.

Another approach to handling lost transactions would be to store in the database the initial message sent by a client, as well as the outcome and the response, and to reprocess the message with normal business logic. If the outcome is the same, the transaction can be safely applied.

> **Note**
>
> If restartable batch operations are implemented, lost batch transactions can be ignored since a subsequent batch restart will process them correctly.

$ZSEARCH()

The $ZSEARCH function attempts to locate a file matching the specified file name. If the file exists, it returns the file name; if the file does not exist, it returns the null string.

The format for the $ZSEARCH function is:

```
$ZSEARCH(expr[,intexpr])
```

- The expression contains a file name, with or without wildcards, for which $ZSEARCH() attempts to locate a matching file. Repeating $ZSEARCH with the same filename uses the same context and return a sequence of matching files when they exist; when the sequence is exhausted, $ZSEARCH() returns an empty string (""). Any change to the file name starts a new context.

- $ZSEARCH() uses the process current working directory, if the expression does not specify a directory.

- The optional integer expression specifies a "stream" number from 0 to 255 for each search; streams provide a means of having up to 256 $ZSEARCH() contexts simultaneously in progress.

- If a $ZSEARCH() stream has never been used or if the expression differs from the argument to the last $ZSEARCH() of the stream, the function resets the context and returns the first pathname matching the expression; otherwise, it returns the next matching file in collating sequence; if the last prior pathname returned for the same expression and same stream was the last one matching the argument, $ZSEARCH() returns a null string.

$ZSEARCH() provides a tool for verifying that a file exists. For information to help determine the validity of a file name, see "$ZPARSE()" (page 271).

> **Note**
>
> You can call the POSIX stat() function to access metadata. The optional GT.M POSIX plug-in packages the stat() function for easy access from M application code.

Examples of $ZSEARCH()

Example:

```
GTM>write $zsearch("data.dat")
```

```
/usr/staff/ccc/data.dat
GTM>
```

This uses $ZSEARCH() to display the full file path name of "data.dat" in the process current default directory.

Example:

```
GTM>set x=$zsearch("*.c")
GTM>for  set x=$zsearch("*.m") quit:x=""  write !,$zparse(x,"NAME")
```

This FOR loop uses $ZSEARCH() and $ZPARSE() to display M source file names in the process current working directory. To ensure that the search starts at the beginning, the example resets the context by first searching with a different argument.

$ZSIGPROC()

Sends a signal to a process. The format for the $ZSIGPROC function is:

```
$ZSIGPROC(expr1,expr2)
```

- The first expression is the pid of the process to which the signal is to be sent.

- The second expression is the system signal number. Because a signal number of a signal name can be different for various platforms, FIS recommends using signal names to maintain code portability across different platforms. For example, the signal number for SIGUSR1 is 10 on Linux, 30 on AIX, and 16 for some other platforms. Use the $&gtmposix.signalval(signame,.sigval) function available in the gtmposix plugin to determine the signal number of a signal name.

- If the second expression is 0, $ZSIGPROC() checks the validity of the pid specified in the first expression.

- There are four possible return values from $ZSIGPROC():

Return codes/POSIX Error Definitions	Description
0	The specified signal number was successfully sent to the specified pid. Any return value other than 0 indicates an error.
EPERM	The process has insufficient permissions to send the signal to the specified pid.
ESRCH	The specified pid does not exist.
EINVAL	Invalid expression(s).

⚠️ **Caution**

Although $ZSIGPROC() may work today as a way to invoke the asynchronous interrupt mechanism of GT.M processes to XECUTE $ZINTERRUPT because the underlying mechanism uses the POSIX USR1 signal, FIS reserves the right to change the underlying mechanism to suit its convenience and sending a POSIX USR1 may cease to work as a way to invoke the asynchronous interrupt mechanism. Use MUPIP INTRPT as the supported and stable API to invoke the asynchronous interrupt mechanism.

Examples of $ZSIGPROC()

Example:

```
GTM>job ^Somejob
GTM>set ret=$&gtmposix.signalval("SIGUSR1",.sigusr1) zwrite
ret=0
sigusr1=10
GTM>write $zsigproc($zjob,sigusr1)
0
GTM>
```

This example sends the SIGUSR1 signal to the pid specified by $zjob.

$ZSUBstr()

Returns a properly encoded string from a sequence of bytes.

```
$ZSUB[STR] (expr ,intexpr1 [,intexpr2])
```

- The first expression is an expression of the byte string from which $ZSUBSTR() derives the character sequence.

- The second expression is the starting byte position (counting from 1 for the first position) in the first expression from where $ZSUBSTR() begins to derive the character sequence.

- The optional third expression specifies the number of bytes from the starting byte position specified by the second expression that contribute to the result. If the third expression is not specified, the $ZSUBSTR() function returns the sequence of characters starting from the byte position specified by the second expression up to the end of the byte string.

- The $ZSUBSTR() function never returns a string with illegal or invalid characters. With VIEW "NOBADCHAR", the $ZSUBSTR() function ignores all byte sequences within the specified range that do not correspond to valid UTF-8 code-points. With VIEW "BADCHAR", the $ZSUBSTR() function triggers a run-time error if the specified byte sequence contains a code-point value that is not in the character set.

- The $ZSUBSTR() is similar to the $ZEXTRACT() byte equivalent function but differs from that function in restricting its result to conform to the valid characters in the current encoding.

Examples of $ZSUBSTR()

Example:

```
GTM>write $ZCHSET
M
GTM>set char1="a" ; one byte character
GTM>set char2="ç"; two-byte character
GTM>set char3="新"; three-byte character
GTM>set y=char1_char2_char3
GTM>write $zsubstr(y,1,3)=$zsubstr(y,1,5)
0
```

With character set M specified, the expression $ZSUBSTR(y,1,3)=$ZSUBSTR(y,1,5) evaluates to 0 or "false" because the expression $ZSUBSTR(y,1,5) returns more characters than $ZSUBSTR(y,1,3).

Example:

```
GTM>write $zchset
UTF-8
GTM>set char1="a" ; one byte character
GTM>set char2="ç"; two-byte character
GTM>set char3="新"; three-byte character
GTM>set y=char1_char2_char3
GTM>write $zsubstr(y,1,3)=$zsubstr(y,1,5)
1
```

For a process started in UTF-8 mode, the expression $ZSUBSTR(y,1,3)=$ZSUBSTR(y,1,5) evaluates to 1 or "true" because the expression $ZSUBSTR(y,1,5) returns a string made up of char1 and char2 excluding the three-byte char3 because it was not completely included in the specified byte-length.

In many ways, the $ZSUBSTR() function is similar to the $ZEXTRACT() function. For example, $ZSUBSTR(expr,intexpr1) is equivalent to $ZEXTRACT(expr,intexpr1,$L(expr)). Note that this means when using the M character set, $ZSUBSTR() behaves identically to $EXTRACT() and $ZEXTRACT(). The differences are as follows:

- $ZSUBSTR() cannot appear on the left of the equal sign in the SET command where as $ZEXTRACT() can.

- In both the modes, the third expression of $ZSUBSTR() is a byte, rather than character, position within the first expression.

- $EXTRACT() operates on characters, irrespective of byte length.

- $ZEXTRACT() operates on bytes, irrespective of multi-byte character boundaries.

- $ZSUBSTR() is the only way to extract as valid UTF-8 encoded characters from a byte string containing mixed UTF-8 and non UTF-8 data. It operates on Unicode® characters so that its result does not exceed the given byte length.

$ZTRanslate()

Returns a byte sequence that results from replacing or dropping bytes in the first of its arguments as specified by the patterns of its other arguments.

The format for the $ZTRANSLATE() function is:

```
$ZTR[ANSLATE](expr1[,expr2[,expr3]])
```

- The first expression specifies the sequence of octets on which $ZTRANSLATE() operates. If the other arguments are omitted, $ZTRANSLATE() returns this expression.

- The optional second expression specifies the byte for $TRANSLATE() to replace. If a byte occurs more than once in the second expression, the first occurrence controls the translation, and $ZTRANSLATE() ignores subsequent occurrences. If this argument is omitted, $ZTRANSLATE() returns the first expression without modification.

- The optional third expression specifies the replacement bytes for the positionally corresponding bytes in the second expression. If this argument is omitted or shorter than the second expression, $ZTRANSLATE() drops all occurrences of the bytes in the second expression that have no replacement in the corresponding position of the third expression.

- $ZTRANSLATE() provides a tool for tasks such as encryption.

The $ZTRANSLATE() algorithm can be understood as follows:

- $ZTRANSLATE() evaluates each byte in the first expression, comparing it byte by byte to the second expression looking for a match. If there is no match in the second expression, the resulting expression contains the byte without modification.

- When it locates a byte match, $ZTRANSLATE() uses the position of the match in the second expression to identify the appropriate replacement for the original expression. If the second expression has more bytes than the third expression, $ZTRANSLATE() replaces the original byte with a null, thereby deleting it from the result. By extension of this principle, if the third expression is missing, $ZTRANSLATE() deletes all bytes from the first expression that occur in the second expression.

Examples of $ZTRANSLATE()

Example:

```
GTM>set hiraganaA=$char(12354) ; $zchar(227,129,130)
GTM>set temp1=$zchar(130)
GTM>set temp2=$zchar(140)
GTM>set tr=$ztranslate(hiraganaA,temp1,temp2)
GTM>w $ascii(tr)
12364
GTM>
```

In the above example, $ZTRANSLATE() replaces byte $ZCHAR(130) in first expression and matching the first (and only) byte in the second expression with byte $ZCHAR(140) - the corresponding byte in the third expression.

$ZTRIgger()

Examine or load trigger definition. The format of the $ZTRIGGER() function is:

```
$ZTRIgger(expr1[,expr2])
```

- $ZTRIGGER() returns a truth value (1 or 0) depending on the success of the specified action.

- $ZTRIGGER() performs trigger maintenance actions similar those performed by MUPIP TRIGGER.

- If *expr1* evaluates to case-insensitive "FILE", $ZTRIGGER() evaluates *expr2* as the location of the trigger definition file. Then, it applies the trigger definitions in the file specified by *expr2*. If a file contains a delete all (-*), that action produces no user confirmation.

- If *expr1* evaluates to case-insensitive "ITEM", $ZTRIGGER() evaluates *expr2* as a single line or multi-line trigger definition entry. A multi-line trigger definition or a multi-line XECUTE string starts with << and uses $char(10) to denote the newline separator. *expr2* with ITEM does not permit a multi-line XECUTE string to end with the >> terminator. It does not require trigger logic to appear immediately after the -xecute=<<, but a $char(10) must prefix the logic as shown in the following examples:

```
set trigstr="+^a -xecute=<< -commands=S"_$char(10)_" do ^twork1"_$char(10)_" do ^twork2"_$char(10) write
▶ $ztrigger("item",trigstr)
set trigstr="+^a -xecute=<< -commands=S "_$c(10)_" do ^twork1"_$c(10)_" do ^twork2"_$c(10) write
▶ $ztrigger("item",trigstr)
```

▼

- If *expr1* evaluates to case-insensitive "SELECT", $ZTRIGGER() evaluates the optional *expr2* as a trigger name or name wildcard, and directs its output to $IO. A FALSE result (0) indicates there are no matching triggers.

- $ZTRIGGER() always operates within a TP transaction even if it needs to implicitly create one to encapsulate its work. Trigger maintenance operations reserve their output until the transaction commits (TCOMMIT where $TLEVEL goes to zero), at which time they deliver their entire output to the current $IO containing consistent information for all $ZTRIGGER() invocations within the successful processing of a larger transaction at that ultimate TCOMMIT. If an explicit transaction ends with a TROLLBACK, it does not produce any $ZTRIGGER() output.

- $ZTRIGGER() can appear within a transaction as long as it does not update any triggers for globals which have had triggers invoked earlier in the same transaction.

- An attempt by a $ZTRIGGER() within a transaction to remove or replace a trigger on a global after the transaction has activated any trigger defined within the named global generates a TRIGMODINTP error.

- $ZTRIGGER() treats the deletion of a non-existent trigger as a success; if that is the only operation, or one of a set of successful operations, it return success (TRUE/1) to the GT.M process. $ZTRIGGER() returns failure in case of trigger selection using trigger names where the number after the pound-sign (#) starts with a 0 (which is an impossible auto-generated trigger name).

- GT.M maps trigger definitions to the region to which they apply.

- GT.M allows defining triggers with the same name and signature in different regions, but does not allow defining triggers with the same name, but different signatures, in different regions within the same global directory. When loading a trigger definition, if a user-defined name conflicts with a name in any region to which the trigger applies, $ZTRIGGER() reports an error. However, when the name is auto-generated, it generates a name in every region, so if there are multiple (spanning) regions, the same trigger might have a differing auto-generated name in each region.

> **Note**
>
> A $ZTRIGGER() action (delete or select) applies to all triggers in all regions matching the specified signature. If the argument specifies an incomplete trigger signature, for example, only the name, the specification may match multiple triggers and apply the delete or select to all of them. FIS recommends you run a select and analyze the scope of the signature before any signature limited delete.

Examples of $ZTRIGGER()

Example:

```
GTM>set X=$ztrigger("S")
GTM>
```

This example displays the current trigger definitions stored in the database.

```
GTM>set X=$ztrigger("i","+^Acct(sub=:) -command=set -xecute=""set ^X($ztvalue)=sub""")
GTM>
```

This example adds a trigger definition for the first level node of ^**Acct.**

Example:

```
GTM>set trigstr="+^a -commands=S -xecute=<<"_$c(10)_" do ^twork1"_$c(10)_" do ^twork2"_$c(10) write $ztrigger("item",trigstr)
```

This example demonstrates the usage of the $ztrigger("ITEM",<multi-line-trigger-definition>> where <<denotes the definition of a multi-line -XECUTE string and $c(10) to denote the newline separator. Unlike the $ztrigger("FILE") form, $ztrigger("ITEM",<multi-line-trigger-definition>> does not require the trigger definition to terminate with >>.

Example:

```
GTM>write $ztrigger("file","agbl.trg")
1
GTM>
```

This example is equivalent to the previous $ztrigger("ITEM") example. In this example, agbl.trg contains the following multi-line trigger definition:

```
+^a -commands=S -xecute=<<
 do ^twork1
 do ^twork2
>>
```

Unlike $ztrigger("ITEM"), $ztrigger("FILE") usages require the trigger definition to terminate with >>

$ZTRNLNM()

The $ZTRNLNM function returns the value of an environment variable.The $ZTRNLNM function is analogous to the DCL Lexical function F$TRNLNM on OpenVMS.

> **Note**
>
> $ZTRNLNM() does not perform iterative translation.

The format for the $ZTRNLNM function is:

```
$ZTRNLNM(expr1[,expr2[,expr3[,expr4[,expr5[,expr6]]]]])
```

expr1 specifies the environment variable whose value needs to be returned.

expr2 to expr5 are OpenVMS-related expressions that specify logical name table(s), index (numbered from 0), initial mode of the look-up, and a value indicating whether the look-up is case sensitive. To ensure interoperability between UNIX and OpenVMS versions, $ZTRNLNM() on UNIX accepts these expressions and ignores them.

The optional expr6 specifies any one of the following keywords:

ITEM KEYWORD	DATA RETURNED
FULL	Returns the translation.
LENGTH	Length of the return value in bytes.
VALUE	Returns the translation.

Examples of $ZTRNLNM()

Example:

```
GTM>write $ztrnlnm("gtm_dist","","","","","VALUE")
```

```
/usr/lib/fis-gtm/V6.3-004_x86_64/utf8
GTM>write $ztrnlnm("gtm_dist")
/usr/lib/fis-gtm/V6.3-004_x86_64/utf8
GTM>
```

These example use $ZTRNLNM() to display the translation value for gtm_dist.

$ZWidth()

Returns the numbers of columns required to display a given string on the screen or printer. The format of the $ZWIDTH() function is:

```
$ZW[IDTH] (expr)
```

- The expression is the string which $ZWIDTH() evaluates for display length. If the expression contains a code-point value that is not a valid UTF-8 character, $ZWIDTH() generates a run-time error.

- If the expression contains any non-graphic characters, the $ZWIDTH() function does count not those characters.

- If the string contains any escape sequences containing graphical characters (which they typically do), $ZWIDTH() includes those characters in calculating its result, as it does not do escape processing. In such a case, the result many be larger than the actual display width.

With character set UTF-8 specified, the $ZWIDTH() function uses the ICU's glyph-related conventions to calculate the number of columns required to represent the expression.

> **Note**
>
> When in "NOBADCHAR" mode, $ZWIDTH() returns give any bad characters a length of zero (0), which may or may not match the behavior of any device used to display the string.

Examples of $ZWIDTH()

Example:

```
GTM>set NG=$char($$FUNC^%HD("200B"))
GTM>set S=$char(26032)_NG_$CHAR(26033)
GTM>W $ZWidth(S)
4
GTM>
```

In the above example, the local variable NG contains a non-graphic character which does not display between two double-width UTF-8 characters.

Example:

```
GTM>write $zwidth("The rain in Spain stays mainly in the plain.")
44
GTM>set A="主要雨在西班牙停留在平原"
GTM>write $length(A)
12
GTM>write $zwidth(A)
24
```

In the above example, the $ZWIDTH() function returns 24 because each character in local variable A occupies 2 columns when they are displayed on the screen or printer.

$ZWRite()

Converts its first string argument to or from ZWRITE format (quoted graphics characters concatenated with $CHAR() representations of any non-graphic characters). The second integer expression controls the direction of conversion. The format of the $ZWRITE() function is:

```
$ZWRITE(expr[,intexpr])
```

- The first argument specifies the string to convert to or from the ZWRITE format.

- The second argument specifies the direction of conversion. When intexpr is not specified or evaluates to zero, $zwrite() converts the first argument to the ZWRITE format. When intexpr evaluates to a non-zero value, $ZWRITE() treats the first argument as being in ZWRITE format and attempts to convert it to a string with embedded non-graphic characters; if it is not in ZWRITE format, it returns an empty string.

- Converting to zwrite format tends to produce a string that is longer than the input and therefore a $ZWRITE() result may exceed the maximum string length - the maximum input length that is guaranteed not to do so is a 116,510 byte string.

- If all its arguments are literals, $ZWRITE() evaluates to a literal constant at compile time.

- Note that non-graphic characters differ between M mode and UTF-8 mode.

Example:

```
GTM>set temp="X"_$char(10)_"X" ; $CHAR(10) is the linefeed character
GTM>write temp
X
X
GTM>write $zwrite(temp)
"X"_$C(10)_"X"
GTM>write $zwrite($zwrite(temp),1)
X
X
GTM>
```

Chapter 8. Intrinsic Special Variables

Revision History		
Revision V7.1-004	27 June 2024	• In "$ZMAXTPTIme" (page 314), add precision and limits
Revision V7.1-003	23 November 2023	• In "$Storage" (page 299), Revise $STORAGE description
Revision V7.1-002	19 September 2023	• In "$ZTDAta" (page 334), add clarification
		• In "$ZICUver" (page 309), Added $ZICUVER Intrinsic Special Variable provides the current International Character Utilities (ICU)
		• In "$ZTIMeout" (page 329), microsecond accuracy
Revision V7.1-001	26 June 2023	• In "$ZCOmpile" (page 305), improve clarity and correctness
		• In "$ZCstatus" (page 306), mention ZLINK and auto-ZLINK
		• In "$ZTIMeout" (page 329), add TP behavior
Revision V7.0-002	23 March 2022	• In "$ZINTerrupt" (page 309), add more details about ISV handling in the interrupt state
		• In "$ZMALLoclim" (page 313), add $ZMALLOCLIM information
Revision V7.0-001	24 November 2021	• In "$Key" (page 296), add $KEY information for non-blocking WRITEs
		• In "$SYstem" (page 300), add that SET can modify the tail of the value
		• In "$ZCstatus" (page 306), correct typo in ISV name
		• In "$ZINTerrupt" (page 309), add more information on $ZINTERRUPT
		• In "$ZKey" (page 312), add information about the WRITE values
		• In "$ZTIMeout" (page 329), Add clarifications on behavior with different trap modes and vector contents.
Revision V6.3-012	08 April 2020	• In "$Horolog" (page 295), add reference to the system clock accuracy
		• In "$ZHorolog" (page 308), add reference to system clock accuracy

Revision V6.3-009	27 June 2019	• In "Interrupt Handling" (page 88), clean up and consolidate redundant text and insert an additional caution.
Revision V6.3-008	24 April 2019	• In "$ZGbldir" (page 307), specify that NEWing $ZGBLDIR is the same as SET $ZGBLDIR=""; remove redundant information and the caution note.
Revision V6.3-007	04 February 2019	• In "Interrupt Handling" (page 88), change Typing to READing; modifications in the service of clarity and reduce passive voice. • In "$ZAUDit" (page 303), add new section. • In "$ZCstatus" (page 306), update the description for V6.3-007
Revision V6.3-006	26 October 2018	• In "$Key" (page 296), changed "bad" to "malformed". • In "$ZPATNumeric" (page 315), UTF-8 mode tweaks. • In "$ZTIMeout" (page 329), add new section
Revision V6.3-005	29 June 2018	• In "$ZQuit" (page 318), add the description of $ZQUIT
Revision V6.3-004	23 March 2018	• In "$ZRELdate" (page 319), add ISV $ZRELDATE
Revision V6.3-003	12 December 2017	• In "$ZSTep" (page 327), Add section for $ZSTRPLLIM ISV. • In "$ZSTRPllim" (page 328), Add the new $ZSTRPLLIM ISV.
Revision V6.3-002	22 August 2017	• In "$ZROUTINES settings for auto-relink" (page 325), update for technical correctness and add more information about re-linked routines. • In "Establishing the Value from $gtmroutines" (page 320), revise the initial value from an undefined $gtmroutines • In "$ZROutines" (page 319), Fix $ZROUTINES references
Revision V6.3-001	20 March 2017	• In "Create a shared library from object (.o) files" (page 324), removed HP-UX, Soalris, Tru64 and z/OS references • In "$Reference" (page 299), added error case to the description. • In "Triggers ISVs" (page 333), added the description of $ZTDElim • In "$ZCHset" (page 303), removed the platform restriction wording and add the compilation warning.

		• In "$ZHorolog" (page 308), added missing $ to ISV name, deleted an extraneous space, and changed expect -> except. • In "$ZPATNumeric" (page 315), added a compile time warning. • In "$ZPOUT" (page 317), added zpin and zpout sections • In "$ZSTep" (page 327), added the role of $gtm_zstep • In "$ZUT" (page 332), $ZUT has a line about $ZH, which exists as a first line in $ZH; added links • In "$ZVersion" (page 332), added compile time warning. • In "$ZWRite()" (page 290), added information about the second argument and information about string inflation going to zwrite format. Modified description to use "argument" as a user found the use of expression unclear.
Revision V6.2-001	27 February 2015	• Updated "$ETrap" (page 295), "$ZTrap" (page 331), "$ETRAP Behavior" (page 568), and "Nesting $ETRAP and using $ESTACK" (page 569) for V6.2-000 enhancements. • Updated "$Principal" (page 297) for read and write behavior empty SOCKET and an empty non-$PRINCIPAL SOCKET device and added a note on creating a SOCKET device for $PRINCIPAL.
Revision V6.1-000	28 August 2014	• In "$Key" (page 296), added information about using $KEY with SOCKET devices. • Added two new sections called "$ZCLose" (page 304) and "$ZKey" (page 312).
Revision V6.0-003	24 February 2014	• Corrected the description of "$ZVersion" (page 332). • In "$Key" (page 296), specified that GT.M maintains $KEY for terminals.
Revision V6.0-001	21 March 2013	In "$ZCOmpile" (page 305), added a point about $ZCOMPILE returning a status of 1 after any error in compilation.

This chapter describes the M Intrinsic Special Variables implemented in GT.M. All entries starting with the letter Z are GT.M additions to the ANSI standard Intrinsic Special Variables. None of the Intrinsic Special Variables are case sensitive.

M Intrinsic Special Variables start with a single dollar sign ($). GT.M provides such variables for program examination. In some cases, the Intrinsic Special Variables may be set to modify the corresponding part of the environment.

Note

None of the Intrinsic Special Variables can be KILLed. SETting or NEWing is generally not allowed, but is specifically noted in the descriptions of those that do.

$Device

$D[EVICE] reflects the status of the current device. If the status of the device does not reflect any error-condition, the value of $DEVICE, when interpreted as a truth-value is 0 (FALSE). If the status of the device reflect any error-condition, the value of $DEVICE, when interpreted as a truth-value is 1 (TRUE).

Note

The initial value of $DEVICE is implementation dependant. However, if the initial value of $IO is the empty string, then the initial value of $DEVICE is also empty string.

$DEVICE gives status code and meaning, in one access:

Example:

```
1,Connection reset by peer
```

The above message is displayed on the server side when the socket device is closed on the client side.

$ECode

$EC[ODE] contains a list of error codes for "active" errors -the error conditions which are not yet resolved. If there are no active errors, $ECODE contains the empty string. Whenever an error occurs, a code for that error is appended to the value of $ECODE in such a way that the value of $ECODE always starts and ends with a comma.

The value of $ECODE can be SET, and when it is set to a non-NULL value, error processing starts.

Note

See Chapter 13: "Error Processing" (page 562) to learn about $ECODE's role in error processing.

List of codes for $ECODE start with comma seperated by commas. A code starts with "M", "U", or "Z", with rest numeric. "M" codes are assigned by MDC (MUMPS Development Committee), "U" by application (programmers), and "Z" codes by MUMPS implementors (in this case GT.M).

An error always has a GT.M specified code and many errors also have an ANSI Standard code. The complete list of standardized error codes can be referenced from GT.M Message and Recovery Procedures Reference Manual version 4.3 and onwards.

```
IF $ECODE[",M61," WRITE "Undefined local variable"
```

Note

The leftmost character of the value of $ECODE is always a comma. This means that every error code that is stored in $ECODE is surrounded by commas. If $ECODE was to contains the error code without the commas

(that is, "M61"), the variable would check for subset "M6" as well. Thus, it is recommended that you include the commas in the value to check. For example; check whether $ECODE contains ",M61,".

$ECODE can be SET but not NEW'd. When $ECODE is set to the empty string (" "), error handling becomes "inactive" and therefore QUIT does not trigger additional error handling.

When $ECODE is not set to the empty string, M error handling is active, which also affects behavior in some aspects of $STACK.

$EStack

$ES[TACK] contains an integer count of the number of M virtual machine stack levels that have been activated and not removed since the last time $ESTACK was NEW'd.

A NEW $ESTACK saves the value of current $ESTACK and then sets its value to zero (0). If $ESTACK has not been NEW'd in the current execution path, $ESTACK=$STACK.

```
SET $ETRAP="QUIT:$ESTACK GOTO LABEL^ROUTINE"
```

$ESTACK maybe used as a flag to indicate error traps invoked in particular stack levels needed to perform some different action(s). $ESTACK can be most useful in setting up a layered error trapping mechanism.

> **Note**
>
> GT.M does not permit $ESTACK to be SET, however $ESTACK can be NEWed.

$ETrap

$ET[RAP] contains a string value that GT.M invokes when an error occurs during routine execution. When a process is initiated, GT.M assigns $ETRAP the value of the gtm_etrap environment variable, if gtm_etrap is defined, and otherwise the empty string, in which case $ZTRAP="B" controls initial error handling.

The value of this variable is the M[UMPS] code that GT.M executes when it encounters an error.

```
SET $ETRAP="QUIT:$ESTACK GOTO LABEL^ROUTINE"
```

The value of $ETRAP is changed with the SET command. Changing the value of $ETRAP with the SET command initiates a new trap; it does not save the old trap.

$ETRAP may also appear as an argument to an inclusive NEW command. NEW $ETRAP causes GT.M to stack the active condition handler's ($ETRAP) old value. NEW leaves the $ETRAP unchanged regardless of the previously active condition handler. NEW $ETRAP command puts $ETRAP in control for error handling.

For more examples of the use of special variable $ETRAP, see the function $STACK().

$Horolog

$H[OROLOG] contains a string value specifying the number of days since "31 December, 1840," and the number of seconds since midnight of date in the time zone of the process, separated by a comma (,). At midnight, the piece of the string following the comma resets to zero (0), and the piece preceding the comma increments by one (1). GT.M does not permit the SET

command to modify $HOROLOG. A process takes the system time from the system clock, but can adjust the time zone by appropriately setting the TZ environment variable before invoking GT.M. $HOROLOG accuracy is subject to the precision of the system clock (use man gettimeofday from the UNIX shell for more information).

Example:

```
GTM>Write $HOROLOG
```

Produces the result 58883,55555 at 3:25:55 pm on 20 March, 2002.

For further information on formatting $HOROLOG for external use, refer to "$ZDate()" (page 262).

$IO

$I[O] contains the name of the current device specified by the last USE command. The M standard does not permit the SET command to modify $IO. USE 0 produces the same $IO as USE $P[RINCIPAL], but $P is the preferred construct.

$Job

$J[OB] the current process identifier.

GT.M uses the decimal representation of the current process identifier (PID) for the value of $JOB. $JOB is guaranteed to be unique for every concurrently operating process on a system. However, operating systems reuse PIDs over time. GT.M does not permit the SET command to modify $JOB.

Example:

```
LOOP0 for  set itm=$order(^tmp($J,itm)) quit:itm=""  do LOOP1
```

This uses $J as the first subscript in a temporary global to insure that every process uses separate data space in the global ^tmp.

$Key

$K[EY] contains the string that terminated the most recent READ command from the current device (including any introducing and terminating characters). If no READ command was issued to the current device or if no terminator is used, the value of $KEY is an empty string. However, when input is terminated by typing a function key, the value of $KEY is equal to the string of characters that is transmitted by that function key.

The effect of a READ *glvn on $KEY is unspecified.

For terminals, $KEY and $ZB both have the terminator.

> **Note**
>
> See the READ and WRITE commands in Chapter 6: "Commands" (page 108).

For SOCKET:

$KEY contains the socket handle and the state information of the current SOCKET device after certain I/O commands.

After a successful OPEN or USE with the LISTEN deviceparameter, $KEY contains for TCP sockets:

```
"LISTENING|<socket_handle>|<portnumber>"
```

and for LOCAL sockets:

```
"LISTENING|<socket_handle>|<address>"
```

After a successful OPEN or USE with the CONNECT device parameter or when GT.M was started with a socket as the $PRINCIPAL device, $KEY contains:

```
"ESTABLISHED|<socket handle>|<address>"
```

When WRITE /WAIT selects an incoming connection, $KEY contains:

```
"CONNECT|<socket_handle>|<address>"
```

▼

When WRITE /WAIT selects a socket with data ▶ available for reading, $KEY contains:

```
"READ|<socket_handle>|<address>"
```

For TCP sockets, <address> is the numeric IP address for the remote end of the connection. For LOCAL sockets it is the path to the socket.

For TCP LISTENING sockets, <portnumber> is the local port on which socket_handle is listening for incoming connections. For LOCAL LISTENING sockets, it is the path of the socket.

If the WRITE /WAIT was timed, $KEY returns an empty value if the wait timed out or there was no established connection. $KEY only has the selected handle, if any, immediately after a WRITE /WAIT. $KEY is also used by other socket I/O commands such as READ which sets it to the delimiter or malformed UTF-8 character, if any, which terminated the read.

When a socket has enabled non-blocking WRITEs, additional values will show if it is possible to write at least one byte to the socket. If the second argument of WRITE /WAIT is omitted or specifies both "READ" and "WRITE" and the socket selected by WRITE /WAIT is ready for both READ and WRITE, $KEY contains:

```
READWRITE|<socket handle>|<address>
```

If the socket selected by a WRITE /WAIT which implicitly or explicitly requests the write state would block on a READ but not block on WRITE, $KEY contains:

```
WRITE|<socket handle>|<address>
```

If the socket selected by WRITE /WAIT which implicitly or explicitly requests the read state would not block on a READ but would block on a WRITE, $KEY contains:

```
READ|<socket handle>|<address>
```

$Principal

$P[RINCIPAL] contains the absolute pathname of the principal (initial $IO) device. $PRINCIPAL is an MDC Type A enhancement to standard M.

Input and output for a process may come from separate devices, namely, the standard input and output. However, the M I/O model allows only one device to be USEd (or active) at a time. When an image starts, GT.M implicitly OPENs the standard input

and standard output device(s) and assigns the device(s) to $PRINCIPAL. For USE deviceparameters, it is the standard input that determines the device type.

For an image invoked interactively, $PRINCIPAL is the user's terminal. For an image invoked from a terminal by means of a shell script, $PRINCIPAL is the shell script's standard input (usually the terminal) and standard output (also usually the terminal) for output, unless the shell redirects the input or output.

GT.M provides a mechanism for the user to create a name for $PRINCIPAL in the shell before invoking GT.M. The environment variable gtm_principal, if defined becomes a synonym for the actual device and the value for $PRINCIPAL. $IO holds the same value as $PRINCIPAL. $ZIO in this case, holds the fully expanded name of the actual device. See "$ZIO" (page 311) for an example of its usage.

GT.M ignores a CLOSE specifying the principal device. GT.M does not permit the SET command to modify $PRINCIPAL.

GT.M fulfills READ to $PRINCIPAL when it is an empty socket device (that is, one with all sockets detached) with an empty string, and discards WRITE output to such a device.

GT.M opens /dev/null as a placeholder for a socket which used to be associated with $PRINCIPAL via stdin when it is closed.

GT.M creates a SOCKET device for $PRINCIPAL when standard input is a LOCAL domain socket and sets the default DELIMITER to "$C(10)" for sockets in the device.

When $PRINCIPAL identifies a device that supports REWIND, the REWIND or INREWIND device parameters perform a REWIND of the input and OUTREWIND performs a REWIND of the output.

When $PRINCIPAL has different input/output devices, the USE command recognizes intrinsic special variables $ZPIN or $ZPOUT and applies appropriate deviceparameters to the input or output side of $PRINCIPAL, respectively. A USE with $ZPIN or $ZPOUT sets $IO to $PRINCIPAL for READs and WRITEs from the input and output side of $PRINCIPAL. For more information refer to "$ZPIN" (page 317) or "$ZPOUT" (page 317) .

$Quit

$Q[UIT] indicates whether the current block of code was called as an extrinsic function or as a subroutine.

If $Q[UIT] contains 1 (when the current process-stack frame is invoked by an extrinsic function), the QUIT would therefore require an argument.

> **Note**
>
> When a process is initiated, but before any commands are processed, the value of $Q[UIT] is zero (0).

This special variable is mainly used in error-trapping conditions. Its value tells whether the current DO level was reached by means of a subroutine call (DO xxx) or by a function call (SET variable=$$xxx).

A typical way of exiting from an error trap is:

```
QUIT:$QUIT "" QUIT
```

> **Note**
>
> GT.M does not permit $QUIT to be SET or NEWed.

$Reference

$R[EFERENCE] contains the last global reference. Until the first global reference is made by an M program or after a global reference with an invalid key, $REFERENCE contains the empty string (""). This way it is useful in determining if the usage of a naked reference is valid.

A typical way of using this is:

```
IF $REFERENCE="" QUIT "<undefined>"
```

> **Note**
>
> $R[EFERENCE] being a read-only variable cannot be SET or NEW'd.

$STack

$ST[ACK] contains an integer value of zero (0) or greater indicating the current level of M execution stack depth.

When a process is initiated but before any command is executed, the value of $STACK is zero (0).

> **Note**
>
> The difference between $STACK and $ESTACK is that $ESTACK may appear as an argument of the NEW command. NEWing $ESTACK resets its value to zero (0), and can be useful to set up a layered error trapping mechanism.

The value of $STACK is "absolute" since the start of a GT.M. process, whereas the value of $ESTACK is "relative" to the most recent "anchoring point".

For examples on the use of special variable $STACK, see "$STack()" (page 238).

$Storage

$S[TORAGE] contains an integer value specifying difference between the current $ZREALSTOR as a subtrahend and, in order of precedence, $ZMALLOCLIM if it has a non-zero value, the process R_LIMIT if it is not unlimited, and otherwise the maximum of a 64-bit address space.

GT.M uses memory for code (instructions) and data. If the amount of virtual memory available to the process exceeds 2,147,483,647 bytes, it is reported as 2,147,483,647 bytes.

Instruction space starts out with the original executable image. However, GT.M may expand instruction space by ZLINKing additional routines.

Data space starts out with stack space that never expands, and pool space which may expand. Operations such as opening a database or creating a local variable may cause an expansion in pool space. GT.M expands pool space in fairly large increments. Therefore, SETs of local variables may not affect $STORAGE at all or may cause an apparently disproportionate drop in its value.

Once a GT.M process adds either instruction or data space, it never releases that space. However, GT.M does reuse process space made available by actions such as KILLs of local variables. $STORAGE can neither be SET or NEWed.

$SYstem

$SY[STEM] contains a string that identifies the executing M instance. The value of $SYSTEM is a string that starts with a unique numeric code that identifies the manufacturer. Codes are assigned by the MDC (MUMPS Development Committee).

$SYSTEM in GT.M starts with "47" followed by a comma and the evaluation of the environment variable gtm_sysid. If the name has no evaluation, the value after the comma is gtm_sysid.

GT.M accepts SET $SYSTEM=expr, where expr appends to the initial value up to the length permitted for an initial value; an empty string removes any current added value.

$Test

$T[EST] contains a truth value specifying the evaluation of the last IF argument or the result of the last operation with timeout. If the last timed operation timed out, $TEST contains FALSE (0); otherwise, it contains TRUE (1).

$TEST serves as the implicit argument for ELSE commands and argumentless IF commands.

M stacks $TEST when invoking an extrinsic and performing an argumentless DO. After these operations complete with an implicit or explicit QUIT, M restores the corresponding stacked value. Because, with these two exceptions, $TEST reflects the last IF argument or timeout result on a process wide basis. Use $TEST only in immediate proximity to the operation that last updated it.

Neither $SELECT() nor post-conditional expressions modify $TEST.

M routines cannot modify $TEST with the SET command.

Example:

```
IF x=+x DO ^WORK
ELSE SET x=0
```

The ELSE statement causes M to use the value of $TEST to determine whether to execute the rest of the line. Because the code in routine WORK may use IFs and timeouts, this use of $TEST is not recommended.

Example:

```
SET MYFLG=x=+x
IF MYFLG DO ^WORK
IF 'MYFLG SET x=0
```

This example introduces a local variable flag to address the problems of the prior example. Note that its behavior results in the opposite $TEST value from the prior example.

Example:

```
IF x=+x DO ^WORK IF 1
ELSE SET x=0
```

This example uses the IF 1 to ensure that the ELSE works counter to the IF.

$TLevel

$TL[EVEL] contains a count of executed TSTARTs that are currently unmatched by TCOMMITs. $TLEVEL is zero (0) when there is no TRANSACTION in progress. When $TLEVEL is greater than one (>1), it indicates that there are nested sub-

transactions in progress. Sub-transactions are always subject to the completion of the main TRANSACTION and cannot be independently acted upon by COMMIT, ROLLBACK, or RESTART.

$TLEVEL can be used to determine whether there is a TRANSACTION in progress and to determine the level of nesting of sub-transactions.

M routines cannot modify $TLEVEL with SET.

Example:

```
IF $TLEVEL TROLLBACK
```

This example performs a TROLLBACK if a transaction is in progress. A statement like this should appear in any error handler used with transaction processing. For more information on transaction processing, see Chapter 5: "General Language Features of M" (page 68).

$TRestart

$TR[ESTART] contains a count of the number of times the current TRANSACTION has been RESTARTed. A RESTART can be explicit (specified in M as a TRESTART) or implicit (initiated by GT.M as part of its internal concurrency control mechanism). $TRESTART can have values of 0 through 4. When there is no TRANSACTION in progress, $TRESTART is zero (0).

$TRESTART can be used by the application to limit the number of RESTARTs, or to cause a routine to perform different actions during a RESTART than during the initial execution.

> ### Note
>
> GT.M does not permit the SET command to modify $TRESTART.

Example:

```
TRANS TSTART ():SERIAL
IF $TRESTART>2 WRITE !;"Access Conflict" QUIT
```

This example terminates the sub-routine with a message if the number of RESTARTs exceeds 2.

$X

$X contains an integer value ranging from 0 to 65,535, specifying the horizontal position of a virtual cursor in the current output record. $X=0 represents the left-most position of a record or row.

Every OPEN device has a $X. However, M only accesses $X of the current device. Therefore, exercise care in sequencing USE commands and references to $X.

Generally, GT.M increments $X for every character written to and read from the current device. Usually, the increment is 1, but for a process in UTF-8 mode, the increment is the number of glyphs or codepoints (depends on the type of device). M format control characters, write filtering, and the device WIDTH also have an effect on $X.

$X never equals or exceeds the value of the device WIDTH. Whenever it reaches the value equal to the device WIDTH, it gets reset to zero (0).

GT.M follows the MDC Type A recommendation and permits an M routine to SET $X. However, SET $X does not automatically issue device commands or escape sequences to reposition the physical cursor.

For more information, refer to "$X" (page 347).

$Y

$Y contains an integer value ranging from 0 to 65,535 specifying the vertical position of a virtual cursor in the current output page. $Y=0 represents the top row or line.

Every OPEN device has a $Y. However, M only accesses $Y of the current device. Therefore, exercise care in sequencing USE commands and references to $Y.

When GT.M finishes the logical record in progress, it generally increments $Y. GT.M recognizes the end of a logical record when it processes certain M format control characters, or when the record reaches its maximum size, as determined by the device WIDTH, and the device is set to WRAP. The definition of "logical record" varies from device to device. For an exact definition, see the sections on each device type. Write filtering and the device LENGTH also have an effect on $Y.

$Y never equals or exceeds the value of the device LENGTH. Whenever it reaches the value equal to the device LENGTH, it gets reset to zero (0)

GT.M permits an M routine to SET $Y. However, SET $Y does not automatically issue device commands or escape sequences to reposition the physical cursor.

For more information, refer to "$Y" (page 302).

$ZA

$ZA contains a status determined by the last read on the device. The value is a decimal integer with a meaning determined by the device as follows:

For Terminal I/O:

0Indicating normal termination of a read operation

1: Indicating a parity error

2: Indicating that the terminator sequence was too long

9: Indicating a default for all other errors

For Sequential Disk and Tape Files I/O:

0: Indicating normal termination of a read operation

9: Indicating a failure of a read operation

For Fifos I/O:

Decimal representing $JOB (identifier) of the process that wrote the last message the current process read

$ZA refers to the status of the current device. Therefore, exercise care in sequencing USE commands and references to $ZA.

GT.M does not permit the SET command to modify $ZA.

For more information on $ZA, refer "Input/Output Processing".

$ZALlocstor

$ZALLOCSTOR contains the number of bytes that are (sub) allocated (including overhead) by GT.M for various activities. It provides one view (see also "$ZREalstor" (page 319) and "$ZUSedstor" (page 332)) of the process memory utilization and can help identify storage related problems. GT.M does not permit $ZALLOCSTOR to be SET or NEWed.

$ZAUDit

$ZAUDIT contains a boolean value that indicates whether Audit Principal Device is enabled. GT.M does not permit $ZAUDIT to be SET or NEWed.

$ZB

$ZB contains a string specifying the input terminator for the last terminal READ. $ZB contains null and is not maintained for devices other than terminals. $ZB may contain any legal input terminator, such as <CR> (ASCII 13) or an escape sequence starting with <ESC> (ASCII 27), from zero (0) to 15 bytes in length. $ZB contains null for any READ terminated by a timeout or any fixed-length READ terminated by input reaching the maximum length.

$ZB contains the actual character string, not a sequence of numeric ASCII codes.

Example:

```
SET zb=$ZB FOR i=1:1:$L(zb) WRITE !,i,?5,$A(zb,i)
```

This displays the series of ASCII codes for the characters in $ZB.

$ZB refers to the last READ terminator of the current device. Therefore, exercise care in sequencing USE commands and references to $ZB.

GT.M does not permit the SET command to modify $ZB.

For more information on $ZB, refer to the "Input/Output Processing" chapter.

$ZCHset

$ZCHSET is a read-only intrinsic special variable that takes its value from the environment variable gtm_chset. An application can obtain the character set used by a GT.M process by the value of $ZCHSET. $ZCHSET can have only two values --"M", or "UTF-8".

> ## Warning
>
> GT.M performs operations on literals at compile time and the character set may have an impact on such operations. Therefore, always compile with the same character set as that used at runtime.

Example:

```
$ export gtm_chset=UTF-8
$ /usr/lib/fis-gtm/V6.0-001_x86/gtm
GTM>write $zchset
UTF-8
GTM>
```

$ZCLose

Provides termination status of the last PIPE CLOSE as follows:

- -99 when the check times out

- -98 for unanticipated problems with the check

- the negative of the signal value if a signal terminated the co-process.

If positive, $ZCLOSE contains the exit status returned by the last co-process.

$ZDAteform

$ZDA[TEFORM] contains an integer value, specifying the output year format of $ZDATE(). $ZDATEFORM can be modified using the SET command. GT.M initializes $ZDATEFORM to the translation of the environment variable gtm_zdate_form. If gtm_zdate_form is not defined, GT.M initializes $ZDATEFORM to zero (0).

See "$ZDate()" (page 262) for the usage of $ZDATEFORM. $ZDATEFORM also defines the behavior of some date and time utility routines; refer "M utility Routines".

Example:

```
GTM>WRITE $ZDATEFROM
0
GTM>WRITE $ZDATE($H)
11/15/02
GTM>SET $ZDATEFORM=1
GTM>WRITE $ZDATE($H)
11/15/2002
```

$ZCMdline

$ZCM[DLINE] contains a string value specifying the "excess" portion of the command line that invoked the GT.M process. By "excess" is meant the portion of the command line that is left after GT.M has done all of its command line processing. For example, a command line mumps -direct extra1 extra2 causes GT.M to process the command line upto mumps -direct and place the "excess" of the command line, that is "extra1 extra2" in $ZCMDLINE. $ZCMDLINE gives the M routine access to the shell command line input.

Note that the actual user input command line might have been transformed by the shell (for example, removing one level of quotes, filename, and wildcard substituion, and so on.), and it is this transformed command line that GT.M processes.

Example:

```
$ cat > test.m
write " $ZCMDLINE=",$ZCMDLINE,!
quit
$ mumps -run test OTHER  information
$ZCMDLINE=OTHER information
$
```

This creates the program test.m, which writes the value of $ZCMDLINE. Note how the two spaces specified in OTHER information in the command line gets transformed to just one space in OTHER information in $ZCMDLINE due to the shell's pre-processing.

Example:

```
$ cat foo.m
foo     ; a routine to invoke an arbitrary entry with or without
 parameters
   ;
  set $etrap="" ; exit if the input isn't valid
  if $length($zcmdline) do @$zcmdline quit
  quit
$ mumps -run foo 'BAR^FOOBAR("hello")'
```

In this example, GT.M processes the shell command line up to foo and puts the rest in $ZCMDLINE. This mechanism allows mumps -run to invoke an arbitrary entryref with or without parameters. Note that this example encloses the command line argument with single quotes to prevent inappropriate expansion in Bourne-type shells. Always remember to use the escaping and quoting conventions of the shell and GT.M to prevent inappropriate expansion.

> ⚠️ **Important**
>
> Use the ^%XCMD utility to XECUTEs code from the shell command line and return any error status (truncated to a single byte on UNIX) that the code generates. For more information, refer to " %XCMD" [509].

$ZCOmpile

$ZCO[MPILE] contains a string value composed of one or more qualifiers that control the GT.M run-time compiler. ZCOMPILE, explicit ZLINKs and auto-ZLINKs use these qualifiers as defaults for any compilations they perform.

$ZCOMPILE is a read-write ISV, that is, it can appear on the left side of the equal sign (=) in the argument to the SET command. A $ZCOMPILE value has the form of a list of M command qualifiers each separated by a space ( ).

When the gtmcompile environment variable is defined, GT.M initializes $ZCOMPILE to the translation of $gtmcompile. Otherwise GT.M initializes $ZCOMPILE to null. Changes to the value of $ZCOMPILE during a GT.M invocation only last for the current invocation and do not change the value of the $gtmcompile environment variable.

When $ZCOMPILE is null, GT.M uses the default M command qualifiers -IGNORE, -LABEL=LOWER, -NOLIST, and -OBJECT. See Chapter 3: "Development Cycle" (page 32) for detailed descriptions of the M command qualifiers.

Example:

```
$ export gtmcompile="-LIST -LENGTH=56 -SPACE=2"
$ gtm
GTM>WRITE $ZCOMPILE
-LIST -LENGTH=56 -SPACE=2
GTM>SET $ZCOMPILE="-LIST -NOIGNORE"
GTM>WRITE $ZCOMPILE
-LIST -NOIGNORE
GTM>ZLINK "A.m"
GTM>HALT
$ echo $gtmcompile
-LIST -LENGTH=56 -SPACE=2
```

This example uses the environment variable gtmcompile to set up $ZCOMPILE. Then it modifies $ZCOMPILE with the SET command. The ZLINK argument specifies a file with a .m extension (type), which forces a compile. The compile produces a

listing for routine A.m and does not produce an object module if A.m contains compilation errors. After GT.M terminates, the shell command echo $gtmcompile demonstrates that the SET command did not change the environment variable.

$ZCstatus

$ZC[STATUS] holds the value of the status code for the last compilation performed by a ZCOMPILE, ZLINK or auto-ZLINK . One (1) indicates a clean compilation, a positive number greater than one is an error code you can turn into text with $ZMESSAGE(), and a negative number is a negated error code that indicates GT.M was not able to produce an object file. The error details appear in the compilation output, so $ZCSTATUS typically contains the code for ERRORSUMMARY.

GT.M does not permit the SET command to modify $ZCSTATUS.

$ZDirectory

$ZD[IRECTORY] contains the string value of the full path of the current directory. Initially $ZDIRECTORY contains the default/current directory from which the GT.M image/process was activated.

If the current directory does not exist at the time of GT.M process activation, GT.M errors out.

Example:

```
GTM>WRITE $ZDIR
/usr/tmp
GTM>SET $ZDIR=".."
GTM>WRITE $ZDIR
/usr
```

This example displays the current working directory and changes $ZDIR to the parent directory.

$ZDIRECTORY is a read-write Intrinsic Special Variable, that is, it can appear on the left side of the equal sign (=) in the argument to a SET command. If an attempt is made to set $ZDIRECTORY to a non-existent directory specification, GT.M issues an error and keeps the value of $ZDIRECTORY unchanged.

At image exit, GT.M restores the current directory to the directory that was the current directory when GT.M was invoked even if that directory does not exist.

$ZEDit

$ZED[IT] holds the value of the status code for the last edit session invoked by a ZEDIT command.

GT.M does not permit the SET or NEW command to modify $ZEDIT.

$ZEOf

$ZEO[F] contains a truth-valued expression indicating whether the last READ operation reached the end-of-file. $ZEOF equals TRUE (1) at EOF and FALSE (0) at other positions.

GT.M does not maintain $ZEOF for terminal devices.

$ZEOF refers to the end-of-file status of the current device. Therefore, exercise care in sequencing USE commands and references to $ZEOF.

GT.M does not permit the SET or NEW command to modify $ZEOF.

For more information on $ZEOF, refer to the "Input/Output Processing" chapter.

$ZError

$ZE[RROR] is supposed to hold the application-specific error-code corresponding to the GT.M error-code stored in $ECODE/ $ZSTATUS (see "$ECode" (page 294) and "$ZStatus" (page 327)).

$ZERROR contains a default value of "Unprocessed $ZERROR, see $ZSTATUS" at process startup.

$ZERROR can be SET but not NEWed.

The mapping of a GT.M error-code to the application-specific error-code is achieved as follows. Whenever GT.M encounters an error, $ECODE/$ZSTATUS gets set first. It then invokes the code that $ZYERROR points to if it is not null. It is intended that the code invoked by $ZYERROR use the value of $ZSTATUS to select or construct a value to which it SETs $ZERROR. If an error is encountered by the attempt to execute the code specified in $ZYERROR, GT.M sets $ZERROR to the error status encountered. If $ZYERROR is null, GT.M does not change the value of $ZERROR. In all cases, GT.M proceeds to return control to the code specified by $ETRAP/$ZTRAP or device EXCEPTION whichever is applicable. For details, see "$ZYERror" (page 333).

$ZGbldir

$ZG[BLDIR] contains the value of the current Global Directory filename. When $ZGBLDIR specifies an invalid or inaccessible file, GT.M cannot successfully perform database operations.

GT.M initializes $ZGBLDIR to the translation of the environment variable gtmgbldir. The value of the gtmgbldir environment variable may include a reference to another environment variable. If gtmgbldir is not defined, GT.M initializes $ZGBLDIR to null. When $ZGBLDIR is null, GT.M constructs a file name for the Global Directory using the name $gtmgbldir and the extension .gld in the current working directory.

$ZGBLDIR is a read-write Intrinsic Special Variable, (that is, it can appear on the left side of the equal sign (=) in the argument to the SET command). SET $ZGBLDIR="" causes GT.M to assign $ZGBLDIR to the translation of gtmgbldir if that environment variable is defined. If it is not defined, then SET $ZGBLDIR="" causes GT.M to construct a file name using the name $gtmgbldir.gld in the current directory. NEWing $ZGBLDIR is the same as SET $ZGBLDIR="", which as just noted may change its value. For code that immediately SETs $ZGBLDIR after NEW'ng it that behavior doesn't matter, but without an associated SET, such a change may seem counterintuitive. A $ZGBLDIR value may include an environment variable.

SETting $ZGBLDIR also causes GT.M to attempt to open the specified file. If the file name is invalid or the file is inaccessible, GT.M triggers an error without changing the value of $ZGBLDIR.

To establish a value for $ZGBLDIR outside of M, use the appropriate shell command to assign a translation to gtmgbldir. Defining gtmgbldir provides a convenient way to use the same Global Directory during a session where you repeatedly invoke and leave GT.M.

Example:

```
$ gtmgbldir=test.gld
$ export gtmgbldir
$ gtm
GTM>WRITE $zgbldir
```

```
/usr/dev/test.gld
GTM>SET $zgbldir="mumps.gld"
GTM>WRITE $zgbldir
mumps.gld
GTM>HALT
$ echo $gtmgbldir
test.gld
```

This example defines the environment variable gtmgbldir. Upon entering GT.M Direct Mode, $ZGBLDIR has the value supplied by gtmgbldir. The SET command changes the value. After the GT.M image terminates, the echo command demonstrates that gtmgbldir was not modified by the M SET command.

```
$ ls test.gld
test.gld not found
$ gtm
GTM>WRITE $zgbldir
/usr/dev/mumps.gld
GTM>set $zgbldir="test.gld"
%GTM-E-ZGBLDIRACC, Cannot access global directory
"/usr/dev/test.gld". Retaining /usr/dev/mumps.gld"
%SYSTEM-E-ENO2, No such file or directory
GTM>WRITE $zgbldir
/usr/dev/mumps.gld
GTM>halt
$
```

The SET command attempts to change the value of $ZGBLDIR to test.gld. Because the file does not exist, GT.M reports an error and does not change the value of $ZGBLDIR.

$ZHorolog

$ZH[OROLOG] returns 4 comma-separated pieces (for example, "63638,39194,258602,14400"). The first two pieces are identical to the two pieces of $HOROLOG. $ZHOROLOG is a drop-in replacement for $HOROLOG in all application code of the form $PIECE($HOROLOG,",",...). For example, $ZHOROLOG can be used as the first argument of $ZDATE(). The third piece is the number of microseconds in the current second. The accuracy of the third piece is subject to the precision of the system clock. The fourth piece is an offset in seconds to UTC. For any valid UTC time offset, the fourth piece is a number between -43200 (for UTC-12:00) and +50400 (for UTC+14:00). The value of the fourth piece remains constant all through the year except for those places that observe daylight saving time. $ZHOROLOG accuracy is subject to the precision of the system clock (use man gettimeofday from the UNIX shell for more information). To obtain the $HOROLOG representation of UTC, add the fourth piece to the second piece of $ZHOROLOG and proceed as follows:

- If the result is a negative number, subtract one from the first piece and add 86400 (number of seconds in a day) to the second piece.

- If the result is a positive number greater than 86400, add one to the first piece and subtract 86400 from the second piece.

Example:

```
GTM>zprint ^zhoro
zhoro(zone)
 set:'$data(zone) zone="Europe/London"
 new zutzh
 set zutzh=$$getzutzh(zone)
 do displaytzdetails(zutzh,zone)
```

```
 quit
getzutzh(zone)
  set shcommand="TZ="_zone_" $gtm_dist/mumps -run %XCMD 'write $zut,"" "",$zhorolog,"" "",$zdate($horolog,""MON
 DD,YYYY
▸12:60:SS AM""),!'"
  set descname="tzpipe"
  open descname:(shell="/bin/sh":command=shcommand:readonly)::"pipe"
  use descname read dateline use $principal close descname
  quit dateline
displaytzdetails(zutzh,zone)
 set zut=$piece(zutzh," ",1)    ; $ZUT
 set zh=$piece(zutzh," ",2)     ; $ZHOROLOG
 set zhfp=$piece(zh,",",1)      ; first piece of $ZH of zone
 set zhsp=$piece(zh,",",2)
 set zhtp=$piece(zh,",",3)
 set zhfop=$piece(zh,",",4)
 set tz=zhfop/3600,hours=$select(tz*tz=1:" Hour ",1:" Hours ")
 write "Time in ",zone," ",$piece(zutzh," ",3,6)," $ZUT=",zut,!,$select(tz<0:-tz_hours_"Ahead
▸of",1:tz_hours_"Behind")," UTC",!
 set zhsp=zhsp+zhfop
 if zhsp>86400 set zhfp=zhfp+1,zhsp=zhsp-86400      ; 86400 seconds in a day
 else  if zhsp<1 set zhfp=zhfp-1,zhsp=zhsp+86400
 write "Time in UTC ",$zdate(zhfp_","_zhsp,"MON DD,YYYY 12:60:SS AM")
 quit
GTM>do ^zhoro
Time in Europe/London APR 10,2015 05:20:29 PM $ZUT=1428682829213711
1 Hour Ahead of UTC
Time in UTC APR 10,2015 04:20:29 PM
GTM>
```

$ZICUver

$ZICUVER provides the current International Character Utilities (ICU) version or an empty string if ICU is not available.

> **Note**
>
> The version is blank/empty if $ZCHSET is "M".

$ZINInterrupt

$ZINI[NTERRUPT] evaluates to 1 (TRUE) when a process is executing code initiated by the interrupt mechanism, and otherwise 0 (FALSE).

GT.M does not permit the SET or NEW commands to modify $ZININTERRUPT.

$ZINTerrupt

$ZINT[ERRUPT] specifies the code that GT.M XECUTEs at the time of processing an interrupt (for example, through a MUPIP INTRPT). While a $ZINTERRUPT action is in process, GT.M defers any additional interrupt signals. At the time of invoking an

interrupt handler, GT.M saves the current values of $ECODE and $REFERENCE and restores them when the interrupt handler returns. GT.M neither saves nor restores the current device ($IO).

GT.M permits the SET command to modify the value of $ZINTERRUPT.

If an interrupt handler changes the current IO device (via USE), it is the responsibility of the interrupt handler to restore the current IO device before returning. There are sufficient legitimate possibilities why an interrupt routine would want to change the current IO device (for example; daily log switching), that this part of the process context is not saved and restored automatically.

The initial value for $ZINTERRUPT is taken from the UNIX environment variable gtm_zinterrupt if it is specified, otherwise it defaults to the following string:

```
IF $ZJOBEXAM()
```

The IF statement executes the $ZJOBEXAM function but effectively discards the return value.

> **Note**
>
> If the default value for $ZINTERRUPT is modified, no $ZJOBEXAM() occurs unless the replacement value directly or indirectly invokes that function. In other words, while $ZJOBEXAM() is part of the interrupt handling by default, it is not an implicit part of the interrupt handling.

The interrupt handler is executed by GT.M when on a statement boundary or on an appropriate boundary in a potentially long running COMMAND (in the same place as GT.M recognizes <CTRL-C>). If a GT.M process is in a long running external call (for example; waiting in a message queue) GT.M does not have sufficient control of the process state to immediately drive the interrupt handler. GT.M recognizes the interrupt request and drives the handler after the external call returns to GT.M and the process reaches an appropriate execution boundary.

> **Note**
>
> The interrupt handler does not operate "outside" the current M environment but rather within the environment of the process.

It is possible for the interrupt handler to be executed while the process executing a TP transaction holds the critical section for one or more regions. Use of this feature may create temporary hangs or pauses while the interrupt handler executes. For the default case where the interrupt handler uses $ZJOBEXAM() to create a dump, the pause duration depends on the number of local variables in the process at the time of the dump and on the speed of the secondary storage device. Such a dump is slower on a network-mounted secondary storage device than on a storage device directly connected to the local system. Design interrupt driven code to account for this issue.

When GT.M uses an interrupt handler, it saves and restores the current value of $REFERENCE. However, GT.M does not implicitly save or restore the current device ($IO). If an interrupt handler changes $IO (via USE), ensure that the interrupt handler restores the current device before returning. To restore the device which was current when the interrupt handler began, specify USE without any deviceparameters. Any attempt to do IO on a device which was actively doing IO when the interrupt was recognized may cause a ZINTERCURSEIO error.

Code in $ZINTERRUPT must use routine names in any entryref argument to a DO, GOTO, JOB, ZGOTO or any extrinsic as the arrival of the interrupt can activate the code while executing an arbitrary routine in the application.

During the execution of the interrupt handling code, $ZININTERRUPT evaluates to 1 (TRUE).

If an error occurs while compiling the $ZINTERRUPT code, the error handler is not invoked, GT.M sends the ERRWZINTR message and the compiler error message to the operator log facility. If the GT.M process is at a direct mode prompt or is executing a direct mode command (for example, a FOR loop), GT.M also sends the ERRWZINTR error message to the user console along with the compilation error. In both cases, the interrupted process resumes execution without performing any action specified by the defective $ZINTERRUPT vector.

The error handler is invoked if an error occurs while executing the $ZINTERRUPT code. If an error occurs during execution of the interrupt handler's stack frame (before it calls anything), that error is prefixed with the ERRWZINTR error. The error handler then executes normal error processing associated with the module that was interrupted. Any other errors that occur in code called by the interrupt handler are handled by normal error handling. See Chapter 13: "Error Processing" (page 562).

If a TP transaction is in progress (0<$TLEVEL), updates to globals are not safe since a TP restart can be signaled at any time prior to the transaction being committed - even after the interrupt handler returns. A TP restart reverses all global updates and unwinds the M stack so it is as if the interrupt never occurred. The interrupt handler is not redriven as part of a transaction restart. Referencing (reading) globals inside an interrupt handler can trigger a TP restart if a transaction is active. When programming interrupt handling, either discard interrupts when 0<$TLEVEL (forcing the interrupting party to try again), or use local variables that are not restored by a TRESTART to defer the interrupt action until after the final TCOMMIT.

> ⚠️ **Important**
>
> Because sending an interrupt signal requires the sender to have appropriate permissions, the use of the job interrupt facility itself does not present any inherent security exposures. Nonetheless, because the dump files created by the default action contain the values of every local variable in the context at the time they are made, inappropriate access to the dump files would constitute a security exposure. Make sure the design and implementation of any interrupt logic includes careful consideration to security issues.

Example:

```
set $zinterrupt="do ^interrupthandler($io)"
interrupthandler(currentdev)
      do ^handleinterrupt ; handle the interrupt
      use currentdev      ; restore the device which was current when the interrupt was recognized
      quit
```

For more information on interrupt handling, refer to "Interrupt Handling" (page 88).

$ZIO

$ZIO contains the translated name of the current device, in contrast to $IO, which contains the name as specified by the USE command.

GT.M does not permit the SET or NEW command to modify $ZIO.

An example where $ZIO contains a value different from $IO is if the environment variable gtm_principal is defined.

Example:

```
$ gtm_principal="foo"
$ export gtm_principal
GTM>WRITE $IO
foo
GTM>WRITE $ZIO
/dev/pts/8
```

Notice that $ZIO contains the actual terminal device name while $IO contains the string pointed to by the environment variable gtm_principal.

$ZJob

$ZJ[OB] holds the pid of the process created by the last JOB command performed by the current process.

GT.M initializes $ZJOB to zero (0) at process startup. If the JOB command fails to spawn a new job, GT.M sets $ZJOB to zero (0). Note that because of the left to right evaluation order of M, using $ZJOB in the jobparameter string results in using the value created by the last, rather than the current JOB command, which is not likely to match common coding practice.

GT.M does not permit the SET or NEW command to modify $ZJOB.

$ZKey

For Socket devices:

$ZKEY contains a list of sockets in the current SOCKET device which are ready for use. Its contents include both non selected but ready sockets from the prior WRITE /WAITs and any sockets with unread data in their GT.M buffer. $ZKEY can be used any time a SOCKET device is current. Once an incoming socket (that is, "LISTENING") has been accepted either by being selected by WRITE /WAIT or by USE socdev:socket="listeningsocket", it is removed from $ZKEY.

$ZKEY contains any one of the following values:

```
"LISTENING|<listening_socket_handle>|{<portnumber>|</path/to/LOCAL_socket>}"
```

```
"READ|<socket_handle>|<address>"
```

```
"WRITE|<socket_handle>|<address>"
```

If $ZKEY contains one or more "READ|<socket_handle>|<address>" entries, it means there are ready to READ sockets that were selected by WRITE /WAIT or were partially read and there is data left in their buffer. Each entry is delimited by a ";".

If $ZKEY contains one or more "LISTENING|<listening_socket_handle>|{<portnumber|/path/to/LOCAL_socket>}" entries, it means that there are pending connections and a USE s:socket=listening_socket_handle will accept a pending connection and remove the LISTENING|<listening_socket_handle> entry from $ZKEY.

If $ZKEY contains one or more "WRITE|<socket_handle>|<address>" entries, it means that the prior WRITE /WAIT considered the non-blocking sockets writeable. This is likely to be the case most of the time. If $ZKEY contains one WRITE and one READ entry, it means that the non-blocking socket is both readable and writeable.

$ZKEY is empty if no sockets have data in the buffer and there are no unaccepted incoming sockets from previous WRITE / WAITs.

For Sequential File Device:

$ZKEY contains the current position in the file based on the last READ. This is in bytes for STREAM and VARIABLE formats, and in a record,byte pair for FIXED format. For FIXED format, SEEKs and normal READs always produce a zero byte position; a non-zero byte position in $ZKEY for FIXED format operation indicates a partially read record, caused by a READ # or READ *. In FIXED mode, the information returned for $ZKEY is a function of record size, and, if a USE command changes record size by specifying the WIDTH deviceparameter while the file is open, $ZKEY offsets change accordingly; if record size changes, previously saved values of $ZKEY are likely inappropriate for use with SEEK.

$ZLevel

$ZL[EVEL] contains an integer value indicating the "level of nesting" caused by DO commands, XECUTE commands, and extrinsic functions in the M invocation stack.

$ZLEVEL has an initial value of one (1) and increments by one with each DO, XECUTE or extrinsic function. Any QUIT that does not terminate a FOR loop decrements $ZLEVEL. ZGOTO may also reduce $ZLEVEL. In accordance with the M standard, a FOR command does not increase $ZLEVEL. M routines cannot modify $ZLEVEL with the SET or NEW commands.

Use $ZLEVEL in debugging or in an error-handling mechanism to capture a level for later use in a ZGOTO argument.

Example:

```
GTM>zprint ^zleve
zleve;
 do B
 write X,!
 quit
B
 goto C
 quit
C
 do D
 quit
D
 set X=$ZLEVEL
 quit
GTM>do ^zleve
4
GTM>
```

This program, executed from Direct Mode, produces a value of 4 for $ZLEVEL. If you run this program from the shell, the value of $ZLEVEL is three (3).

$ZMALLoclim

$ZMALL[OCLIM] contains an integer value specifying a number of bytes of process memory, which, if exceeded, cause GT.M to issue a MALLOCCRIT error. When the value is zero (0), GT.M imposes no limit, although the OS still does. A positive value specifies a byte limit with a minimum of 2.5MB. A value of minus one (-1) provides a value of half the system imposed limit if any.

When a request for additional memory exceeds the limit, GT.M does the expansion and then produces trappable MALLOCCRIT warning. By default, some later request for memory is likely to produce a fatal MEMORY error, unless subsequent to MALLOCCRIT, a limit has been reestablished by SET ZMALLOCLIM to the same or higher limit, but one not exceeding any system limit.

Note that GT.M allocates memory from the OS in large blocks so the interaction of $ZMALLOCLIM with memory growth is not exact. MEMORY errors are fatal and terminate the process. In the case of a MEMORY error, GT.M makes an attempt to marshal available memory to enable as graceful a termination as possible. Note that independent of this mechanism, the OS may kill the process without recourse if it determines the greed of the process for memory jeopardizes the viability of the system.

When the integer byte value specified in a SET $ZMALLOCLIN=intexpr or, at process startup, by the $gtm_malloc_limit environment variable specifies a positive value, GT.M uses the smaller of that value and any OS defined amount for the value

of $ZMALLOCLIM. GT.M does not give errors or messages about its choice for $ZMALLOCLIM between a specified value and some other more appropriate value, so if the application needs to verify the result, it should examine the resulting ISV value.

$ZMAXTPTIme

$ZMAXTPTI[ME] contains a value between zero (0) and 60 in seconds with millisecond precision indicating the time duration GT.M should wait for the completion of all activities fenced by the current transaction's outermost TSTART/TCOMMIT pair. Values outside the accepted range leave the current value unchanged.

$ZMAXTPTIME can be SET but cannot be NEWed.

$ZMAXTPTIME takes its value from the environment variable gtm_zmaxtptime. If gtm_zmaxtptime is not defined, the initial value of $ZMAXTPTIME is zero (0) seconds which indicates "no timeout" (unlimited time). The value of $ZMAXTPTIME when a transaction's outermost TSTART operation executes determines the timeout setting for that transaction.

When a $ZMAXTPTIME expires, GT.M executes the $ETRAP/$ZTRAP exception handler currently in effect.

> **Note**
>
> Negative values of $ZMAXTPTIME are also treated as "no timeout". Timeouts apply only to the outermost transaction, that is, $ZMAXTPTIME has no effect when TSTART is nested within another transaction.

Example:

```
Test;testing TP timeouts
  set $ZMAXTPTIME=6,^X=0,^Y=0,^Z=0
  write "Start with $ZMAXTPTIME=",$ZMAXTPTIME,":",!
  for sleep=3:2:9 do
  . set retlvl=$zlevel
  . do longtran;ztrap on longtran
  ;continues execution
  ;on next line
  . write "(^X,^Y)=(",^X,",",^Y,")",!
  write !,"Done TP Timeout test.",!
 quit
longtran ;I/O in TP doesn't get rolled back
  set $etrap=" goto err"
  tstart ():serial
  set ^X=1+^X
  write !,"^X=",^X,",will set ^Y to ",sleep
  write " in ",sleep," seconds..."
  hang sleep
  set ^Y=sleep
  write "^Y=",^Y
  tcommit
  write "...committed.",!
  quit
err;
  write !,"In $ETRAP handler. Error was: "
  write !," ",$zstatus
  if $TLEVEL do ;test allows handler use outside of TP
  . trollback
  . write "Rolled back transaction."
```

```
  write !
  set $ecode=""
  zgoto retlvl
```

Results:

```
Start with $ZMAXTPTIME=6:
^X=1,will set ^Y to 3 in 3 seconds...^Y=3...committed.
^X=2,will set ^Y to 5 in 5 seconds...^Y=5...committed.
^X=3,will set ^Y to 7 in 7 seconds...
In $ETRAP handler. Error was:
150377322,longtran+7^tptime,%GTM-E-TPTIMEOUT, Transaction timeoutRolled back transaction.
^X=3,will set ^Y to 9 in 9 seconds...
In $ETRAP handler. Error was:
150377322,longtran+7^tptime,%GTM-E-TPTIMEOUT, Transaction timeoutRolled back transaction.
Done TP Timeout test.
```

$ZMOde

$ZMO[DE] contains a string value indicating the process execution mode.

The mode can be:

- INTERACTIVE

- OTHER

M routines cannot modify $ZMODE.

Example:

```
GTM>WRITE $ZMODE
INTERACTIVE
```

This displays the process mode.

$ZONLNrlbk

$ZONLNRLBK increments every time a process detects a concurrent MUPIP JOURNAL -ONLINE -ROLLBACK.

GT.M initializes $ZONLNRLBK to zero (0) at process startup. GT.M does not permit the SET or NEW commands to modify $ZONLNRLBK.

For more information on online rollback, refer to the -ONLINE qualifier of -ROLLBACK in **GT.M Administration and Operations Guide**.

$ZPATNumeric

$ZPATN[UMERIC] is a read-only intrinsic special variable that determines how GT.M interprets the patcode "N" used in the pattern match operator.

With $ZPATNUMERIC="UTF-8", the patcode "N" matches any numeric character as defined by UTF-8 encoding. With $ZPATNUMERIC="M", GT.M restricts the patcode "N" to match only ASCII digits 0-9 (that is, ASCII 48-57). When a process starts in UTF-8 mode, intrinsic special variable $ZPATNUMERIC takes its value from the environment variable gtm_patnumeric. GT.M initializes the intrinsic special variable $ZPATNUMERIC to "UTF-8" if the environment variable

gtm_patnumeric is defined to "UTF-8". If the environment variable gtm_patnumeric is not defined or set to a value other than "UTF-8", GT.M initializes $ZPATNUMERIC to "M".

GT.M populates $ZPATNUMERIC at process initialization from the environment variable gtm_patnumeric and does not allow the process to change the value.

> ## ⚠ Warning
>
> GT.M performs operations on literals at compile time and the pattern codes settings may have an impact on such operations. Therefore, always compile with the same pattern code settings as those used at runtime.

For UTF-8 characters, GT.M assigns patcodes based on the default classification of the Unicode® character set by the ICU library with three adjustments:

1. If $ZPATNUMERIC is not "UTF-8", non-ASCII decimal digits are classified as A.

2. Non-decimal numerics (Nl and No) are classified as A.

3. The remaining characters (those not classified by ICU functions: u_isalpha, u_isdigit, u_ispunct, u_iscntrl, 1), or 2) above) are classified into either patcode P or C. The ICU function u_isprint is used since is returns "TRUE" for non-control characters.

The following table contains the resulting general category as per the Unicode standard to M patcode mapping:

General category as per the Unicode® standard	GT.M patcode class
L* (all letters)	A
M* (all marks)	P
Nd (decimal numbers)	N (if decimal digit is ASCII or $ZPATNUMERIC is "UTF-8", otherwise A
Nl (letter numbers)	A (examples of Nl are Roman numerals)
No (other numbers)	A (examples of No are fractions)
P* (all punctuation)	P
S* (all symbols)	P
Zs (spaces)	P
Zl (line separators)	C
Zp (paragraph separators)	C
C* (all control code points)	C

For a description of the Unicode general categories, refer to http://unicode.org/charts/.

Example:

```
GTM>write $zpatnumeric
UTF-8
GTM>Write $Char($$FUNC^%HD("D67"))?.N ; This is the Malayalam decimal digit 1
1
GTM>Write 1+$Char($$FUNC^%HD("D67"))
```

```
1
GTM>Write 1+$Char($$FUNC^%HD("31")) ; This is the ASCII digit 1
2
```

$ZPIN

When $PRINCIPAL has different input/output devices, the USE command recognizes intrinsic special variable $ZPIN to apply appropriate deviceparameters to the input side of $PRINCIPAL. A USE with $ZPIN sets $IO to $PRINCIPAL for READs and WRITEs from the input and output side of $PRINCIPAL. $ZSOCKET() also accepts $ZPIN as its first argument and, if the device is a split SOCKET device, supplies information on the input SOCKET device. In any context other than USE or $ZSOCKET(), or if $PRINCIPAL is not a split device, $PRINCIPAL, $ZPIN and $ZPOUT are synonyms. In the case of a split $PRINCIPAL, $ZPIN returns the value of $PRINCIPAL followed by the string "< /" Any attempt to OPEN $ZPIN results in a DEVOPENFAIL error.

For more information refer to "$Principal" (page 297), "$ZPOUT" (page 317), and "$ZSOCKET()" (page 278).

$ZPOSition

$ZPOS[ITION] contains a string value specifying the current entryref, where entryref is [label][+offset]^routine, and the offset is evaluated from the closest preceding label.

GT.M does not permit the SET or NEW commands to modify $ZPOSITION.

Example:

```
GTM>WRITE !,$ZPOS,! ZPRINT @$ZPOS
```

This example displays the current location followed by the source code for that line.

$ZPOUT

When $PRINCIPAL has different input/output devices, the USE command recognizes intrinsic special variables $ZPOUT to apply appropriate deviceparameters to the output side of $PRINCIPAL. A USE with $ZPOUT sets $IO to $PRINCIPAL for READs and WRITEs from the input and output side of $PRINCIPAL. $ZSOCKET() also accepts $ZPOUT as its first argument and, if the device is a split SOCKET device, supplies information on the output SOCKET device. In any context other than USE or $ZSOCKET(), or if $PRINCIPAL is not a split device, $PRINCIPAL, $ZPIN and $ZPOUT are synonyms. In the case of a split $PRINCIPAL, $ZPOUT returns the value of $PRINCIPAL followed by the string "> /" Any attempt to OPEN $ZPOUT results in a DEVOPENFAIL error.

For more information refer to "$Principal" (page 297), "$ZPIN" (page 317), and "$ZSOCKET()" (page 278).

Example:

```
;zpioin
;123456789012345678901234567890123456789012345678901234567890
;A123456789012345678901234567890123456789012345678901234567890123456789
zpio
  ; mumps -r zpio < zpioin
  write "$PRINCIPAL = ",$P,!
  write "$ZPIN = ",$ZPIN,!
  write "$ZPOUT = ",$ZPOUT,!
  write "Read first line from zpioin with default settings",!
  read x
  write x,!
```

```
zshow "d"
use $ZPIN:(wrap:width=50)
write "After $ZPIN set to wrap and width set to 50",!
zshow "d"
write "Read next 50 characters from zpioin",!
read y
write y,!
use $ZPOUT:wrap
use $ZPIN:nowrap
write "After $ZPOUT set to wrap and $ZPIN set to nowrap",!
zshow "d"
use $ZPOUT:nowrap
write "After $ZPOUT set to nowrap",!
zshow "d"
use $P:wrap
write "After $P set to wrap",!
zshow "d"
use $ZPOUT:width=40
write "After $ZPOUT width set to 40",!
zshow "d"
use $ZPOUT:nowrap
write "After $ZPOUT set to nowrap",!
zshow "d"
write x,!
quit
```

$ZPROMpt

$ZPROM[PT] contains a string value specifying the current Direct Mode prompt. By default, GTM>is the Direct Mode prompt. M routines can modify $ZPROMPT by means of a SET command. $ZPROMPT cannot exceed 16 characters. If an attempt is made to assign $ZPROMPT to a longer string, only the first 16 characters will be taken.

In UTF-8 mode, if the 31st byte is not the end of a valid UTF-8 character, GT.M truncates the $ZPROMPT value at the end of last character that completely fits within the 31 byte limit.

The environment gtm_prompt initializes $ZPROMPT at process startup.

Example:

```
GTM>set $zprompt="Test01">"
Test01>set $zprompt="GTM>"
```

This example changes the GT.M prompt to Test01> and then back to GTM>.

$ZQuit

$ZQUIT indicates whether the functionality associated with the gtm_zquit_anyway environment variable is enabled. If the environment variable gtm_zquit_anyway is defined and evaluates to 1 or any case-independent string or leading substrings of "TRUE" or "YES", GT.M sets $ZQUIT to 1 and executes code of the form QUIT <expr< as if it were SET <tmp>=<expr> QUIT: $QUIT tmp QUIT, where <tmp> is a temporary local variable in the GT.M runtime system that is not visible to the application code. The value of $ZQUIT is a compile time setting, that is it can be set during run-time but it has no effect on the behavior existing object code - it only effects code generated by auto-ZLINK or the ZCOMPILE command. This setting has no effect on late bound code such as that in XECUTE arguments, $ETRAP, $ZTRAP device EXCEPTIONS.

If gtm_zquit_anyway is not defined or evaluates to 0 or any case-independent string or leading substrings of "FALSE" or "NO", GT.M set $ZQUIT to 0 executes QUIT <expr> as specified by the standard. This feature is helpful when you want to run a block of code both as a routine and as an extrinsic function. Consider the following example:

```
GTM>zprint ^rtnasfunc
rtnasfunc(expr1)
  write "Hello ",expr1,!
  quit
```

When the gtm_zquit_anyway functionality is diabled, extrinsic function invocations return an error as per the standard.

```
GTM>write $zquit
0
GTM>write $$^rtnasfunc("jdoe")
Hello jdoe
  %GTM-E-QUITARGREQD, Quit from an extrinsic must have an argument
```

However, if you enable the gtm_zquit_anyway feature, extrinsic function invocations do not return an error.

```
GTM>write $zquit
1
GTM>write $$^rtnasfunc("jdoe")
Hello jdoe
GTM>
```

$ZREalstor

$ZREALSTOR contains the total memory (in bytes) allocated by the GT.M process, which may or may not actually be in use. It provides one view (see also "$ZALlocstor" (page 303) and "$ZUSedstor" (page 332)) of the process memory utilization and can help identify storage related problems. GT.M does not permit $ZREALSTOR to be SET or NEWed.

$ZRELdate

$ZRELDATE provides the UTC date / time of the build GT.M build in the form YYYYMMDD 24:60 (using $ZDATE() notation). While $ZVERSION is probably a better identifier for most uses, $ZRELDATE may be a helpful alternative for those testing pre-release builds.

$ZROutines

$ZRO[UTINES] contains a string value specifying a directory or list of directories containing object files. Each object directory may also have an associated directory, or list of directories, containing the corresponding source files. These directory lists are used by certain GT.M functions, primarily auto-ZLINK, to locate object and source files. The order in which directories appear in a given list determines the order in which they are searched for the appropriate item.

Searches that use $ZROUTINES treat files as either object or source files. GT.M treats files with an extension of .o as object files and files with an extension of .m as source files.

> **Note**
>
> Paths used in $ZROUTINES to locate routines must not include embedded spaces, as $ZROUTINES uses spaces as delimiters.

Establishing the Value from $gtmroutines

When the environment variable gtmroutines is defined, GT.M initializes $ZROUTINES to the value of gtmroutines. Otherwise, GT.M initializes $ZROUTINES to ".". When $ZROUTINES is ".", GT.M attempts to locate all source and object files in the current working directory. $ZROUTINES="" is equivalent to $ZROUTINES=".".

Commands or functions such as DO, GOTO, ZGOTO, ZBREAK, ZPRINT, and $TEXT may auto-ZLINK and thereby indirectly use $ZROUTINES. If their argument does not specify a directory, ZEDIT and explicit ZLINK use $ZROUTINES. ZPRINT and $TEXT use $ZROUTINES to locate a source file if GT.M cannot find the source file pointed to by the object file. For more information on ZLINK and auto-ZLINK, see the "Development Cycle" [32] and "Commands" [108] chapters.

Setting a Value for $ZROutines

$ZRO[UTINES] is a read-write Intrinsic Special Variable, so M can also SET the value.

By default, each directory entry in $ZROUTINES is assumed to contain both object and source files. However, each object directory may have an associated directory or list of directories to search for the corresponding source files. This is done by specifying the source directory list, in parentheses, following the object directory specification.

If the command specifies more than one source directory for an object directory, the source directories must be separated by spaces, and the entire list must be enclosed in parentheses ( ) following the object directory-specification. If the object directory should also be searched for source, the name of that directory must be included in the parentheses, (usually as the first element in the list). Directory-specifications may also include empty parentheses, directing GT.M to proceed as if no source files exist for objects located in the qualified directory.

To set $ZROUTINES outside of M, use the appropriate shell command to set gtmroutines. Because gtmroutines is a list, enclose the value in quotation marks (" ").

Changes to the value of $ZROUTINES during a GT.M invocation only last for the current invocation, and do not change the value of gtmroutines.

Directory specifications may include an environment variable. When GT.M SETs $ZROUTINES, it translates all environment variables and verifies the syntax and the existence of all specified directories. If $ZROUTINES is set to an invalid value, GT.M generates a run-time error and does not change the value of $ZROUTINES. Because the environment variables are translated when $ZROUTINES is set, any changes to their definition have no effect until $ZROUTINES is set again.

$ZROutines Examples

Example:

```
GTM>s $zroutines=".(../src) $gtm_dist"
```

This example directs GTM to look for object modules first in your current directory, then in the distribution directory that contains the percent routines. GT.M locates sources for objects in your current directory in the sibling /src directory.

Example:

```
$ gtmroutines="/usr/jones /usr/smith"
$ export gtmroutines
$ gtm
GTM>write $zroutines
```

```
"/usr/jones /usr/smith"
GTM>set $zro="/usr/jones/utl /usr/smith/utl"
GTM>write $zroutines
"/usr/jones/utl /usr/smith/utl"
GTM>halt
$ echo $gtmroutines
/usr/jones /usr/smith
```

This example defines the environment variable gtmroutines. Upon entering GT.M Direct Mode $zroutines has the value supplied by gtmroutines. The SET command changes the value. When the GT.M image terminates, the shell echo command demonstrates that gtmroutines has not been modified by the M SET command.

Example:

```
GTM>SET $ZRO=". /usr/smith"
```

This example sets $zroutines to a list containing two directories.

Example:

```
GTM>set $zro="/usr/smith(/usr/smith/tax /usr/smith/fica)"
```

This example specifies that GT.M should search the directory /usr/smith for object files, and the directories /usr/smith/tax and /usr/smith/fica for source files. Note that in this example. GT.M does not search /usr/smith for source files.

Example:

```
GTM>set $zro="/usr/smith(/usr/smith /usr/smith/tax /usr/smith/fica)"
```

This example specifies that GT.M should search the directory /usr/smith for object files and the directories /usr/smith/tax and /usr/smith/fica for source files. Note that the difference between this example and the previous one is that GT.M searches /usr/smith for both object and source files.

Example:

```
GTM>set $zro="/usr/smith /usr/smith/tax() /usr/smith/fica"
```

This specifies that GT.M should search /usr/smith and /usr/smith/fica for object and source files. However, because the empty parentheses indicate directories searched only for object files, GT.M does not search /usr/smith/tax for source files.

Omitting the parentheses indicates that GT.M can search the directory for both source and object files. $ZROUTINES=/usr/smith is equivalent to $ZROUTINES=/usr/smith(/usr/smith).

$ZROutines Search Types

GT.M uses $ZRO[UTINES] to perform three types of searches:

- Object-only when the command or function using $ZROUTINES requires a .o file extension.

- Source-only when the command or function using $ZROUTINES requires a file extension other than .o.

- Object-source match when the command or function using $ZROUTINES does not specify a file extension.

An explicit ZLINK that specifies a non .OBJ .o extension is considered as a function that has not specified a file extension for the above searching purposes.

All searches proceed from left to right through $ZROUTINES. By default, GT.M searches directories for both source and object files. GT.M searches directories followed by empty parentheses ( ) for object files only. GT.M searches directories in parentheses only for source files.

Once an object-matching search locates an object file, the source search becomes limited. If the directory containing the object file has an attached parenthetical list of directories, GT.M only searches the directories in the attached list for matching source files. If the directory containing the object files does not have following parentheses, GT.M restricts the search for matching source files to the same directory. If the object module is in a directory qualified by empty parentheses, GT.M cannot perform any operation that refers to the source file.

The following table shows GT.M commands and functions using $ZROUTINES and the search types they support.

GT.M Commands and $ZROUTINES Search Types				
SEARCH/ FUNCTION	FILE EXTENSION SPECIFIED	SEARCH TYPE		
		OBJ-ONLY	SRC-ONLY	MATCH
EXPLICIT ZLINK	.o	X		
	Not .o			X
	None			X
AUTO-ZLINK	None			X
ZEDIT	Not .o		X	
ZPRINT	None		X	
$TEXT	None		X	

If ZPRINT or $TEXT() require a source module for a routine that is not in the current image, GT.M first performs an auto-ZLINK with a matching search.

ZPRINT or $TEXT locate the source module using a file specification for the source file located in the object module. If GT.M finds the source module in the directory where it was when it was compiled, the run-time system does not use $ZROUTINES. If GT.M cannot find the source file in the indicated location, the run-time system uses $ZROUTINES.

$ZROutines Search Examples

This section describes a model for understanding $ZROUTINES operations and the illustrating examples, which may assist you if you wish to examine the topic closely.

You may think of $ZROUTINES as supplying a two dimensional matrix of places to look for files. The matrix has one or more rows. The first row in the matrix contains places to look for object and the second and following rows contain places to look for source. Each column represents the set of places that contain information related to the object modules in the first row of the column.

Example:

```
GTM>s $zro=". /usr/smi/utl() /usr/jon/utl
```

```
(/usr/jon/utl/so /usr/smi/utl)"
```

The following table illustrates the matrix view of this $ZROUTINES.

$ZROUTINES Search Matrix			
SEARCH FOR	Column 1	Column 2	Column 3
OBJECTS	.	/usr/smi/utl	/usr/jon/utl
SOURCE	.		/usr/jon/utl/so
			/usr/smi/utl

To perform object-only searches, GT.M searches only the directories or object libraries in the top 'objects' row for each column starting at column one. If GT.M does not locate the object file in a directory or object library in the 'objects' row of a column, GT.M begins searching again in the next column. If GT.M cannot locate the file in any of the columns, it issues a run-time error.

As illustrated in the preceding table, GT.M searches for object files in the directories . ,/usr/smi/utl and /usr/jon/utl.

To perform source-only searches, GT.M looks down to the 'source' row at the bottom of each column, excluding columns headed by object-only directories (that is, those object directories, which consist of an empty list of source directories) or object libraries. If GT.M cannot locate the source file in the 'source' row of a column, it searches the next eligible column.

To perform object-source match searches, GT.M looks at each column starting at column one. GT.M does an object-only search in the 'objects' row of a column and a source-only search in the 'source' row(s) of a column. If GT.M locates either the object-file or the souce-file, the search is completed. Else, GT.M starts searching the next column. If GT.M cannot locate either the object file or the source file in any of the columns, it issues a run-time error.

As illustrated in the preceding table, GT.M searches for source-files in the directory "." in column one. If GT.M cannot locate the source file in ".", it omits column two because it is an object-only directory and instead searches column three. Since column three specifies /usr/jon/utl/so and /usr/smi/utl, GT.M searches for the source-file in these directories in column three and not in /usr/jon/utl. If GT.M cannot locate the source-file in column three, it terminates the search and issues a run-time error.

Once the object-source match search is done, GT.M now has either the object-file or source-file or both available. GT.M then recompiles the source-file based on certain conditions, before linking the object-file into the current image. See "ZLink" (page 181) for more information on those conditions.

If auto-ZLINK or ZLINK determines that the source file requires [re]compilation, GT.M places the object file in the above object directory in the same column as the source file. For example, if GT.M locates the source file in /usr/smi/utl in column three, GT.M places the resultant object file in /usr/jon/utl.

Shared Library File Specification in $ZROUTINES

The $ZROUTINES ISV allows individual UNIX shared library file names to be specified in the search path. During a search for auto-ZLINK, when a shared library is encountered, it is probed for a given routine and, if found, that routine is linked/loaded into the image. During an explicit ZLINK, all shared libraries in $ZROUTINES are ignored and are not searched for a given routine.

$ZROUTINES syntax contains a file-specification indicating shared library file path. GT.M does not require any designated extension for the shared library component of $ZROUTINES. Any file specification that does not name a directory is treated as shared library. However, it is recommended that the extension commonly used on a given platform for shared library files be given to any GT.M shared libraries. See "Linking GT.M Shared Images" (page 324). A shared library component cannot

specify source directories. GT.M reports an error at an attempt to associate any source directory with a shared library in $ZROUTINES.

The following traits of $ZROUTINES help support shared libraries:

- The $ZROUTINES search continues to find objects in the first place, processing from left to right, that holds a copy; it ignores any copies in subsequent locations. However, now for auto-ZLINK, shared libraries are accepted as object repositories with the same ability to supply objects as directories.

- Explicit ZLINK, never searches Shared Libraries. This is because explicit ZLINK is used to link a newly created routine or re-link a modified routine and there is no mechanism to load new objects into an active shared library.

- ZPRINT and $TEXT() of the routines in a shared library, read source file path from the header of the loaded routine. If the image does not contain the routine, an auto-ZLINK is initiated. If the source file path recorded in the routine header when the module was compiled cannot be located, ZPRINT and $TEXT() initiate a search from the beginning of $ZROUTINES, skipping over the shared library file specifications. If the located source does not match the code in image (checked via checksum), ZPRINT generates an object-source mismatch status and $TEXT() returns a null string.

- ZEDIT, when searching $ZROUTINES, skips shared libraries like explicit ZLINK for the same reasons. $ZSOURCE ISV is implicitly set to the appropriate source file.

For example, if libshare.so is built with foo.o compiled from ./shrsrc/foo.m, the following commands specify that GT.M should search the library ./libshare.so for symbol foo when do ^foo is encountered.

```
GTM>SET $ZROUTINES="./libshare.so ./obj(./shrsrc)"
GTM>DO ^foo;auto-ZLINK foo - shared
GTM>ZEDIT "foo";edit ./shrsrc/foo.m
GTM>W $ZSOURCE,!;prints foo
GTM>ZPRINT +0^foo;issues a source-object mismatch status TXTSRCMAT error message
GTM>ZLINK "foo";re-compile ./shrsrc/foo.m to generate ./obj/foo.o.
GTM>W $TEXT(+0^foo);prints foo
```

Note that ZPRINT reports an error, as foo.m does not match the routine already linked into image. Also note that, to recompile and re-link the ZEDITed foo.m, its source directory needs to be attached to the object directory [./obj] in $ZROUTINES. The example assumes the shared library (libshare.so) has been built using shell commands. For the procedure to build a shared library from a list of GT.M generated object (.o) files, see "Linking GT.M Shared Images" (page 324).

Linking GT.M Shared Images

Following are the steps (UNIX system commands, and GT.M commands) that need to be taken to use GT.M shared image linking with $ZROUTINES.

Compile source (.m) files to object (.o) files

In order to share M routines, GT.M generates objects (.o) with position independent code, a primary requirement for shared libraries, done automatically by GT.M V4.4-000 and later releases. No change to the compiling procedures is needed. However, any objects generated by a previous release must be recompiled.

Create a shared library from object (.o) files

To create a shared library, use the following syntax:

```
ld -brtl -G -bexpfull -bnoentry -b64 -o libshr.so file1.o file2.o (on AIX)
```

```
ld -shared -o libshr.so file1.o file2.o (on Linux)
```

Where libshr.so is replaced with name of the shared library one wishes to create. The file1.o and file2.o are replaced with one or more object files created by the GT.M compiler that the user wishes to put into the shared library. Note that the list of input files can also be put into a file and then specified on the command line with the -f option (AIX). Refer to the ld man page on specific platform for details on each option mentioned above.

> ## Notes
>
> - Source directories cannot be specified with a shared library in $ZROUTINES, as GT.M does not support additions or modifications to active shared libraries.
>
> - Searching for a routine in a shared library is a two step process:
>
>   - Load the library
>
>   - Lookup the symbol corresponding to the M entryref
>
>   Since GT.M always performs the first step (even on platforms with no shared binary support), use shared libraries in $ZROUTINES with care to keep the process footprint minimal. On all platforms, it is strongly recommended not to include unused shared libraries in $ZROUTINES.
>
> - There are some tools on AIX that can aid in mitigating the problems of shared library allocation. The /usr/bin/genkld command on AIX lists all of the shared libraries currently loaded. This command requires root privileges on AIX 4.3.3 but seems to be a general user command on AIX 5. The /usr/sbin/slibclean command requires root privileges and will purge the shared library segment of unused shared libraries making room for new libraries to be loaded.

Establish $ZROUTINES from gtmroutines

When the environment variable gtmroutines is defined, GT.M initializes $ZROUTINES to the value of gtmroutines. The $ZROUTINES ISV can also be modified using SET command.

Example:

```
$ gtmroutines="./libabc.so ./obj(./src)"
$ export gtmroutines
$ mumps -direct
GTM>w $ZROUTINES,!;writes "./libabc.so ./obj(./src)"
GTM>do ^a;runs ^a from libabc.so
GTM>do ^b;runs ^b from libabc.so
GTM>do ^c;runs ^c from libabc.so
GTM>h
$
```

$ZROUTINES settings for auto-relink

By suffixing one or more directory names in $ZROUTINES with a single asterisk (*), processes can subscribe to updates of object files published in those directories. At the invocation of DO, GOTO, or ZGOTO, extrinsic functions, $TEXT(), or ZPRINT that specify an entryref which includes a routine name (vs. a label without a routine name), mumps processes (and mupip processes executing trigger logic) automatically relink ("auto-relink") and execute published new versions of routines.

- Label references (that is, without a routine name), whether direct or through indirection, always refer to the current routine, and do not invoke auto-relink logic.

- Use shell quoting rules when appending asterisks to directory names in the gtmroutines environment variable - asterisks must be passed in to GT.M, and not expanded by the shell.

- GT.M accepts but ignores asterisk suffixes to directory names on 32-bit Linux on x86 platforms, where it does not provide auto-relinking.

- Changing $ZROUTINES causes all routines linked from auto-relink-enabled directories in the process to be re-linked.

- Note that a relink does not automatically reload a routine every time. When GT.M initiates a relink and the object file (object hash) is the same as the existing one, GT.M bypasses the relink and uses the existing object file.

The ZRUPDATE command publishes of new versions of routines to subscribers.

$ZSOurce

$ZSO[URCE] contains a string value specifying the default pathname for the ZEDIT and ZLINK commands. ZEDIT or ZLINK without an argument is equivalent to ZEDIT/ZLINK $ZSOURCE.

$ZSOURCE initially contains the null string. When ZEDIT and ZLINK commands have a pathname for an argument, they implicitly set $ZSOURCE to that argument. This ZEDIT/ZLINK argument can include a full pathname or a relative one. A relative path could include a file in the current directory, or the path to the file from the current working directory. In the latter instance, do not include the slash before the first directory name. $ZSOURCE will prefix the path to the current working directory including that slash.

The file name may contain a file extension. If the extension is .m or .o, $ZSOURCE drops it. The ZEDIT command accepts arguments with extensions other than .m or .o. $ZSOURCE retains the extension when such arguments are passed.

If $ZSOURCE contains a file with an extension other than .m or .o, ZEDIT processes it but ZLINK returns an error message

$ZSOURCE is a read-write Intrinsic Special Variable, (i.e., it can appear on the left side of the equal sign (=) in the argument to the SET command). A $ZSOURCE value may include an environment variable. GT.M handles logical names that translate to other logical names by performing iterative translations according to VMS conventions. If a logical name translates to a VMS search list, GT.M uses only the first name in the list.

Example:

```
GTM>ZEDIT "subr.m"
.
.
GTM>WRITE $ZSOURCE
subr
```

Example:

```
GTM>ZEDIT "test"
.
.
.
GTM>WRITE $ZSOURCE
"test"
```

Example:

```
GTM>ZEDIT "/usr/smith/report.txt"
.
.
.
GTM>WRITE $ZSOURCE
/usr/smith/report.txt
```

Example:

```
GTM>ZLINK "BASE.O"
.
.
.
GTM>WRITE $ZSOURCE
BASE
```

$ZStatus

$ZS[TATUS] contains a string value specifying the error condition code and location of the last exception condition that occurred during routine execution.

GT.M maintains $ZSTATUS as a string consisting of three or more substrings. The string consists of the following:

- An error message number as the first substring.

- The entryref of the line in error as the second substring; a comma (,) separates the first and second substrings.

- The message detail as the third substring. The format of this is a percent sign (%) identifying the message facility, a hyphen (-) identifying the error severity, another hyphen identifying the message identification followed by a comma (,), which is followed by the message text if any:

```
Format: %<FAC>-<SEV>-<ID>, <TEXT>
Example: %GTM-E-DIVZERO, Attempt to divide by zero
```

GT.M sets $ZSTATUS when it encounters errors during program execution, but not when it encounters errors in a Direct Mode command.

$ZSTATUS is a read-write Intrinsic Special Variable, (i.e., it can occur on the left side of the equal sign (=) in the argument to the SET command). While it will accept any string, FIS recommends setting it to null. M routines cannot modify $ZSTATUS with the NEW command.

Example:

```
GTM>WRITE $ZSTATUS
150373110,+1^MYFILE,%GTM-E-DIVZERO,
Attempt to divide by zero
```

This example displays the status generated by a divide by zero (0).

$ZSTep

$ZST[EP] contains a string value specifying the default action for the ZSTEP command. $ZSTEP provides the ZSTEP action only when the ZSTEP command does not specify an action.

$ZSTEP initially contains the the value of the $gtm_zstep environment variable or string "B" if $gtm_zstep is not defined; note that the default "B" causes the process to enter direct mode. $ZSTEP is a read-write Intrinsic Special Variable, (that is, it can appear on the left side of the equal sign (=) in the argument to the SET command).

Example:

```
GTM>WRITE $ZSTEP
B
GTM>
```

This example displays the current value of $ZSTEP, which is the default.

Example:

```
GTM>SET $ZSTEP="ZP @$ZPOS B"
```

This example sets $ZSTEP to code that displays the contents of the next line to execute, and then enters Direct Mode.

$ZSTRPllim

$ZSTRP[LLIM] provides a way for a process to limit its process private memory used for local variable and scratch storage. When the value is 0 or negative, the default, there is no limit. A positive value specifies a byte limit. When a request for additional memory exceeds the limit, GT.M does the expansion and then produces an STPCRIT error. By default, a later request for memory produces an STPOFLOW, unless subsequent to STPCRIT , $ZSTRPLLIM has been set to the same or higher limit. Note that GT.M allocates memory in large blocks so the interaction of $ZSTRPLLIM with memory growth is not exact. When the gtm_string_pool_limit environment variable specifies a positive value, GT.M uses it for the initial value of $ZSTRPLLIM.

$ZSYstem

$ZSY[STEM] holds the value of the status code for the last subprocess invoked with the ZSYSTEM command.

$ZTExit

$ZTE[XIT] contains an expression that controls the GT.M interrupt facility at the transaction commit or rollback. At each outermost TCOMMIT or TROLLBACK, If +$ZTEXIT evaluates to non-zero (TRUE), then $ZINTERRUPT is XECUTEd after completing the commit or rollback.

$ZTEXIT is a read-write ISV, that is, it can appear on the left side of the equal sign (=) in the argument to the SET command. M routines cannot NEW $ZTEXIT. GT.M initializes $ZTEXIT to null at the process startup. Note that the changes to the value of $ZTEXIT during a GT.M invocation last for the entire duration of the process, so it is the application's responsibility to reset $ZTEXIT after $ZINTERRUPT is delivered in order to turn off redelivering the interrupt every subsequent transaction commit or rollback.

Example:

```
$ export sigusrval=10
$ /usr/lib/fis-gtm/V6.1-000_x86_64/gtm
GTM>zprint ^ztran
foo;
  set $ztexit=1
  set $zinterrupt="d ^throwint"
  tstart ()
  for i=1:1:10 do
  . set ^ACN(i,"bal")=i*100
  tstart ()
```

```
  do ^throwint
  ;do ^proc
  tcommit:$tlevel=2
  for i=1:1:10 do
  . set ^ACN(i,"int")=i*0.05
  ;do ^srv
  if $tlevel trollback
  ;do ^exc
  set $ztexit="",$zinterrupt=""
  quit
bar;
  write "Begin Transaction",!
  set $ztexit=1
  tstart ()
  i '$zsigproc($j,$ztrnlnm("sigusrval")) write "interrupt sent...",!!
  for i=1:1:4 set ^B(i)=i*i
  tcommit
  write "End Transaction",!
  ;do ^srv
  quit
GTM>zprint ^throwint
throwint
  set $zinterrupt="write !,""interrupt occurred at : "",$stack($stack-1,""PLACE""),! set $ztexit=1"
  if '$zsigproc($job,$ztrnlnm("sigusrval")) write "interrupt sent to process"
  write "*************************************",!!
  quit
GTM>do foo^ztran
interrupt sent to process
interrupt occurred at : throwint+3^throwint
*************************************
interrupt occurred at : foo+13^ztran
GTM>
```

In the above call to foo^ztran, the interrupt handler is a user-defined routine, throwint. The process is sent a signal (SIGUSR1), and $ZINTERRUPT is executed. At the outermost trollback, the interrupt is rethrown, causing $ZINTERRUPT to be executed again.

Example:

```
GTM>w $zinterrupt
"IF $ZJOBEXAM()"
GTM>zsystem "ls GTM*"
ls: No match.
GTM>do bar^ztran
Begin Transaction
interrupt sent...
End Transaction
GTM>zsystem "ls GTM*"
GTM_JOBEXAM.ZSHOW_DMP_3951_1   GTM_JOBEXAM.ZSHOW_DMP_3951_2
GTM>
```

This uses the default value of $ZINTERRUPT to service interrupts issued to the process. The $ZJOBEXAM function executes a ZSHOW "*", and stores the output in each GTM_ZJOBEXAM_ZSHOW_DMP for the initial interrupt, and at tcommit when the interrupt is rethrown.

$ZTIMeout

$ZTIMEOUT controls a single process wide timer. The format of the $ZTIMEOUT ISV is:

```
$ZTIMeout=([timeout][:labelref])
```

- The optional timeout in seconds specifies with microsecond accuracy how long from the current time the timer interrupts the process. If the specified timeout is negative, GT.M cancels the timer. If the timeout is zero, GT.M treats it as it would a DO of the vector.

- The optional labelref specifies a code vector defining a fragment of M code to which GT.M transfers control as if with a DO when the timeout expires. If the timeout is missing, the assignment must start with a colon and only changes the vector, and in this case, if the timeout is the empty string, GT.M removes any current vector.

Note that GT.M only recognizes interrupts such as from $ZTIMEOUT at points where it can properly resume operation, such as the beginning of a line, when waiting on a command with a timeout, or when starting a FOR iteration. GT.M defers recognition of a $ZTIMEOUT arriving while executing a TP transaction until after a TROLLBACK or the "master" TCOMMIT. When a timeout occurs, if the last assignment specified no vector, GT.M uses the current $ETRAP or $ZTRAP. When a $ZTIMEOUT expires, it invokes one of three possible vectors:

- Its own labelref, which creates a new stack frame, from which it returns to the point at which it was recognized - similar to what happens for $ZINTERUPT when it recognizes a MUPIP INTRPT; see WARNING below

- If labelref is not specified and $ETRAP is specified, it envokes the $ETRAP code, which it acts like an error and operates as if in the current frame, implicitly dropping the current frame when the $ETRAP vector processing completes

- If labelref is not specified and $ZTRAP is specified, it invokes the $ZTRAP code, acting like an error whose behavior is subject to $gtm_ztrap_form; for additional information, refer to the $ZTRAP documentation

In all cases the invoked code can alter the behavior using the tools at its disposal - for example, GOTO, ZGOTO, $ECODE, etc.

⚠ Warning

> When none of the vectors are specified, a $ZTIMEOUT expiration results in the termination of the process
>
> Using a [Z]GOTO that stays within a $ZTIMEOUT labelref vector frame may cause undefined and undesirable behavior

GT.M rejects an attempted KILL of $ZTIMeout with the VAREXPECTED error and an attempted NEW of $ZTIMeout with the SVNONEW error.

Example:

```
GTM>zprint ^ztimeout
ztimeout
  ; Display $ztimeout
    write !,$ztimeout                  ; display $ZTIMeout - in this case the initial value -1
                                       ; set with a vector (do ^TIMEOUT)
    set $ztimeout="60:do ^TIMEOUT"   ; timeout of 1 minute. After timeout expires, XECUTEs do ^TIMEOUT
    write !,$ztimeout                  ; displays the remaining time:vector until timeout
  ; set without a vector
    set $ztimeout=120                  ; set the timeout to 2 minutes without changing the vector
    set $ztimeout="1234do ^TIMEOUT"  ; missing colon creates a timeout for 1234 seconds
    set $ztimeout="10:"                ; set the timeout to 10 seconds and vector to current etrap or ztrap
    set $ztimeout=-1                   ; set cancels the timeout
  ; Note that set to 0 triggers an immediate timeout
    set $ztimeout=0                    ; triggers the current vector
```

```
    set $ztimeout="0:DO FOO"        ; this has the same effect as DO FOO
GTM>
```

$ZTrap

$ZT[RAP] contains a string value that GT.M XECUTEs when an error occurs during routine execution.

> **Note**
>
> The following discussion assumes that $ETRAP error handling is simultaneously not in effect (that is, $ETRAP=""). See Chapter 13: "Error Processing" (page 562) for more information on the interaction between $ETRAP and $ZTRAP.

When the $ZTRAP variable is not null, GT.M executes $ZTRAP at the current level. The $ZTRAP variable has the initial value of "B," and puts the process in Direct Mode when an error condition occurs. If the value of $ZTRAP is null (""), an exception causes the image to run-down with the condition code associated with the exception. If $ZTRAP contains invalid source code, GT.M displays an error message and puts the process into Direct Mode.

$ZTRAP is a read-write Intrinsic Special Variable, (that is, it can appear on the left side of the equal sign (=) in the argument to the SET command).

$ZTRAP may also appear as an argument to an inclusive NEW command. NEW $ZTRAP causes GT.M to stack the current $ZTRAP value, and set its value to the empty string ($ZTRAP=""). The NEW command puts the $ZTRAP in control for error handling. When the program QUITs from the invocation level where the NEW occurred, GT.M restores the value previously stacked by the NEW. NEW $ZTRAP provides nesting of $ZTRAP. Because $ZTRAP="" terminates the image when an error occurs, SET $ZTRAP= generally follows immediately after NEW $ZTRAP. You may use this technique to construct error handling strategies corresponding to the nesting of your programs. If the environment variable gtm_ztrap_new evaluates to boolean TRUE (case insensitive string "TRUE", or case insensitive string "YES", or a non-zero number), $ZTRAP is NEWed when $ZTRAP is SET; otherwise $ZTRAP is not stacked when it is SET.

> **Note**
>
> QUIT from a $ZTRAP terminates the level at which the $ZTRAP was activated.

Keep $ZTRAP simple and put complicated logic in another routine. If the action specified by $ZTRAP results in another run-time error before changing the value of $ZTRAP, GT.M invokes $ZTRAP until it exhausts the process stack space, terminating the image. Carefully debug exception handling.

Example:

```
GTM>S $ZTRAP="ZP @$ZPOS B"
```

This example modifies $ZTRAP to display source code for the line where GT.M encounters an error before entering Direct Mode.

There are four settings of $ZTRAP controlled by the UNIX environment variable gtm_ztrap_form.

The four settings of gtm_ztrap_form are:

- code - If gtm_ztrap_form evaluates to "code" (or a value that is not one of the subsequently described values), then GT.M treats $ZTRAP as code and handles it as previously described in the documentation.

- entryref - If gtm_ztrap_form evaluates to "entryref" then GT.M treats it as an entryref argument to an implicit GOTO command.

- adaptive - If gtm_ztrap_form evaluates to "adaptive" then if $ZTRAP does not compile to valid M code, then $ZTRAP is treated as just described for "entryref." Since there is little ambiguity, code and entryref forms of $ZTRAP can be intermixed in the same application.

> ⚠️ **Important**
>
> GT.M attempts to compile $ZTRAP before evaluating $ZTRAP as an entryref. Because GT.M allows commands without arguments such as QUIT, ZGOTO, or HANG as valid labels, be careful not to use such keywords as labels for error handling code in "adaptive" mode.

- pope[ntryref] / popa[daptive] - If gtm_ztrap_form evaluates to "POPE[NTRYREF]" or "POPA[DAPTIVE]" (case insensitive) and $ZTRAP value is in the form of entryref, GT.M unwinds the M stack from the level at which an error occurred to (but not including) the level at which $ZTRAP was last SET. Then, GT.M transfers control to the entryref in $ZTRAP at the level where the $ZTRAP value was SET. If the UNIX environment variable gtm_zyerror is defined to a valid entryref, GT.M transfers control to the entryref specified by GTM_ZYERROR (with an implicit DO) after unwinding the stack and before transferring control to the entyref specified in $ZTRAP.

> 📌 **Note**
>
> Like $ZTRAP values, invocation of device EXCEPTION values follow the pattern specified by the current gtm_ztrap_form setting except that there is never any implicit popping with EXCEPTION action.

$ZUSedstor

$ZUSEDSTOR is the value in $ZALLOCSTOR minus storage management overhead and represents the actual memory, in bytes, requested by current activities. It provides one view (see also "$ZALlocstor" (page 303) and "$ZREalstor" (page 319)) of the process memory utilization and can help identify storage related problems. GT.M does not permit $ZUSEDSTOR to be SET or NEWed.

$ZUT

$ZUT (UNIX time or universal time) returns the number of microseconds since January 1, 1970 00:00:00 UTC, which provides a time stamp for directly comparing different timezones. $ZUT accuracy is subject to the precision of the system clock (use man gettimeofday from the UNIX shell for more information).

$ZVersion

$ZV[ERSION] contains a string value specifying the currently installed GT.M. $ZV[ERSION] is a space-delimited string with four pieces as follows:

```
<product> <release> <OS> <architecture>
```

- <product> is always "GT.M".

- <release> always begins with "V", and has the structure V<DB_Format>.<major_release>-<minor_release>[<bug_fix_level>] where:

- <DB_Format> identifies the block format of GT.M database files compatible with the release. For example, V4, V5, and V6. The <DB_Format> piece in $ZVERSION does not change even when a MUPIP UPRGRADE or MUPIP DOWNGRADE changes the DB Format element in the database fileheader.

- <major_release> identifies a release with major enhancements.

- <minor_release> identifies minor enhancements to a major release. The classification of major and minor enhancements is at the discretion of FIS.

- An optional <bug_fix_level> is an upper-case letter indicating bug fixes but no new enhancements. Note that GT.M is built monolithically and never patched. Even though a bug fix release has only bug fixes, it should be treated as a new GT.M release and installed in a separate directory.

- <OS> is the host operating system name.

- <architecture> is the hardware architecture for which the release of GT.M is compiled. Note that GT.M retains it original names for continuity even if vendor branding changes, for example, "RS6000".

M routines cannot modify $ZVERSION.

> ⚠️ **Warning**
>
> GT.M treats $ZVERSION as a literal at compile time. Therefore, always compile with the same version as that used at runtime.

Example:

```
GTM>w $zversion
GT.M V6.0-003 Linux x86_64
```

This example displays the current version identifier for GT.M.

$ZYERror

$ZYER[ROR] is a read/write ISV that contains a string value pointing to an entryref. After GT.M encounters an error, if $ZYERROR is set a non-null value, GT.M invokes the routine at the entryref specified by $ZYERROR with an implicit DO. It is intended that the code invoked by $ZYERROR use the value of $ZSTATUS to select or construct a value to which it SETs $ZERROR. If $ZYERROR is not a valid entryref or if an error occurs while executing the entryref specified by $ZYERROR, GT.M SETs $ZERROR to the error status encountered. GT.M then returns control to the M code specified by $ETRAP/$ZTRAP or device EXCEPTION.

$ZYERROR is implicitly NEWed on entry to the routine specified by $ZYERROR. However, if GT.M fails to compile, GT.M does not transfer control to the entryref specified by $ZYERROR.

GT.M permits $ZYERROR to be modified by the SET and NEW commands.

Triggers ISVs

GT.M provides nine ISVs (Intrinsic Special Variables) to facilitate trigger operations. With the exception of $ZTWORMHOLE, all numeric trigger-related ISVs return zero (0) outside of a trigger context; non-numeric ISVs return the empty string.

$ZTDAta

Within trigger context, $ZTDATA returns $DATA(@$REFERENCE)#2 for a SET or $DATA(@$REFERENCE) for a KILL at the time of trigger invocation, ZKILL or ZWITHDRAW prior to the explicit update. This provides a fast path alternative, avoiding the need for indirection in trigger code, to help trigger code determine the characteristics of the triggering node prior to the triggering update. For a SET, it shows whether the node did or did not hold data - whether a SET is modifying the contents of an existing node or creating data at a new node. For a KILL it shows whether the node had descendants and whether it had data.

$ZTDElim

Within a SET trigger context, $ZTDE[LIM] returns the piece separator, as specified by -delim in the trigger definition. This allows triggers to extract updated pieces defined in $ZTUPDATE without having the piece separator hard coded into the routine. Outside of a SET trigger context, $ZTDELIM is null.

$ZTLevel

Within trigger context, $ZTLEVEL returns the current level of trigger nesting (invocation by a trigger of an additional trigger by an update in trigger context).

$ZTLEVEL greater than one (>1) indicates that there are nested triggers in progress. When a single update invokes multiple triggers solely because of multiple trigger matches of that initial (non-trigger) update, they are not nested (they are chained) and thus all have same $ZTLEVEL.

Example:

```
+^Cycle(1) -commands=Set -xecute="Write ""$ZTLevel for ^Cycle(1) is: "",$ZTLevel Set ^Cycle(2)=1"
+^Cycle(2) -commands=Set -xecute="Write ""$ZTLevel for ^Cycle(2) is: "",$ZTLevel Set ^Cycle(1)=1"
```

These trigger definitions show different values of $ZTLEVEL when two triggers are called recursively (and pathologically).

```
+^Acct("ID") -commands=set -xecute="set ^Acct(1)=$ztvalue+1"
+^Acct(sub=:) -command=set -xecute="set ^X($ztvalue)=sub"
```

SET ^Acct("ID")=10 invokes both the above triggers in some order and $ZTLEVEL will have the same value in both because these triggers are chained rather than nested.

$ZTNAME

Within a trigger context, $ZTNAME returns the trigger name. Outside a trigger context, $ZTNAME returns an empty string.

$ZTOLdval

Within trigger context, $ZTOLDVAL returns the prior (old) value of the global node whose update caused the trigger invocation. This provides a fast path alternative to $GET(@$REFERENCE) at trigger entry (which avoids the heavyweight indirection). If there are multiple triggers matching the same node (chained), $ZTOLDVAL returns the same result for each of them.

Example:

```
+^Acct(1,"ID") -commands=Set -xecute="Write:$ZTOLdval ""The prior value of ^Acct(1,ID) was: "",$ZTOLdval"
```

This trigger gets invoked with a SET and displays the prior value (if it exists) of ^Acct(1,"ID").

```
GTM>w ^Acct(1,"ID")
1975
GTM>s ^Acct(1,"ID")=2011
The prior value of ^Acct(1,ID) was: 1975
```

$ZTRIggerop

Within trigger context, for SET (including MERGE and $INCREMENT() operations), $ZTRIGGEROP has the value "S". For KILL, $ZTRIGGEROP has the value "K" For ZKILL or ZWITHDRAW, $ZTRIGGEROP has the value "ZK".

$ZTSlate

$ZTSLATE allows you to specify a string that you want to make available in chained or nested triggers invoked for an outermost transaction (when a TSTART takes $TLEVEL from 0 to 1). You might use $ZTSLATE to accumulate transaction-related information, for example $ZTOLDVAL and $ZTVALUE, available within trigger context for use in a subsequent trigger later in the same transaction. For example, you can use $ZTSLATE to build up an application history or journal record to be written when a transaction is about to commit.

You can SET $ZTSLATE only while a database trigger is active. GT.M clears $ZTSLATE for the outermost transaction or on a TRESTART. However, GT.M retains $ZTSLATE for all sub-transactions (where $TLEVEL>1).

Example:

```
TSTART ()        ; Implicitly clears $ZTSLAT
SET ^ACC(ACN1,BAL)=AMT          ; Trigger sets $ZTSLATE=ACN_"|"
SET ^ACC(ACN2,BAL)=-AMT         ; Trigger sets $ZTSLATE=$ZTSLATE_ACN_"|"
ZTRIGGER ^ACT("TRANS")          ; Trigger uses $ZTSLATE to update transaction log
TCOMMIT
```

$ZTUPdate

Within trigger context, for SET commands where the GT.M trigger specifies a piece separator, $ZTUPDATE provides a comma separated list of piece numbers of pieces that differ between the current values of $ZTOLDVAL and $ZTVALUE. If the trigger specifies a piece separator, but does not specify any pieces of interest, $ZTUPDATE identifies all changed pieces. $ZTUPDATE is 0 in all other cases (that is: for SET commands where the GT.M trigger does not specify a piece separator or for KILLs). Note that if an update matches more than one trigger, all matching triggers see the same $ZTOLDVAL at trigger entry but potentially different values of $ZTVALUE so $ZTUPDATE could change due to the actions of each matching trigger even though all matching triggers have identical -[z]delim and -piece specifications.

Example:

```
+^trigvn -commands=Set -pieces=1;3:6 -delim="|" -xecute="Write !,$ZTUPDATE"
```

In the above trigger definition entry, $ZTUPDATE displays a comma separated list of the changed piece numbers if on of the pieces of interest: 1,3,4,5,or 6 are modified by the update.

```
GTM>write ^trigvn
```

```
Window|Table|Chair|Curtain|Cushion|Air Conditioner
GTM>set ^trigvn="Window|Dining Table|Chair|Vignette|Pillow|Air Conditioner"
4,5
```

Note that even though piece numbers 2,4 and 5 are changed, $ZTUPDATE displays only 4,5 because the trigger is not defined for updates for the second piece.

$ZTVAlue

For SET, $ZTVALUE has the value assigned to the node by the explicit SET operation. Modifying $ZTVALUE within a trigger modifies the eventual value GT.M assigns to the node. Note that changing $ZTVALUE has a small performance impact because it causes an additional update operation on the node once all trigger code completes. If a node has multiple associated triggers each trigger receives the current value of $ZTVALUE, however, because the triggers run in arbitrary order, FIS strongly recommends no more than one trigger change any given element of application data, for example, a particular piece. For KILL and its variants, $ZTVALUE returns the empty string. While GT.M accepts updates to $ZTVALUE within the trigger code invoked for a KILL or any of its variants, it ultimately discards any such value. Outside trigger context, attempting to SET $ZTVALUE produces a SETINTRIGONLY error.

$ZTWOrmhole

$ZTWORMHOLE allows you to specify a string up to 128KB of information you want to make available during trigger execution. You can use $ZTWORMHOLE to supply an application-context or process context to your trigger logic. Because GT.M makes $ZTWORMHOLE available throughout the duration of the process, you can access or update $ZTWORMHOLE both from inside and outside a trigger.

$ZTWORMHOLE provides a mechanism to access information from a process/application context that is otherwise unavailable in trigger context. GT.M records any non-empty string value of $ZTWORMHOLE in the GT.M database journal file as part of any update that invokes at least one trigger which references $ZTWORMHOLE. GT.M also transmits any non-NULL $ZTWORMHOLE value in the replication stream, thus providing the same context to triggers invoked by MUPIP processes (either as part of the replicating instance update process or as part of MUPIP journal recovery/rollback). Therefore, whenever you use $ZTWORMHOLE in a trigger, you create something like a wormhole for process context that is otherwise NEWed in the run-time or non-existent in MUPIP.

Note that if trigger code does not reference $ZTMORMHOLE, GT.M does not make it available to MUPIP (via the journal files or replication stream). Therefore, if a replicating secondary has different trigger code than the initiating primary (an unusual configuration) and the triggers on the replicating node require information from $ZTWORMHOLE, the triggers on the initiating node must reference $ZTWORMHOLE to ensure GT.M maintains the data it contains for use by the update process on the replicating node. While you can change $ZTWORMHOLE within trigger code, because of the arbitrary ordering of triggers on the same node, such an approach requires careful design and implementation. GTM allows $ZTWORMHOLE to be NEW'd. NEWing $ZTWORMHOLE is slightly different from NEWing other ISVs/variables in the sense that the former retains its original value whereas the latter does not. However, like other NEWs, GT.M restores $ZTWORMHOLE's value when the stack level pops.

The following table summarizes the read/write permissions assigned to all trigger-related ISVs within trigger context and outside trigger context.

Intrinsic Special Variable	Within Trigger Context	Notes
$ETRAP	Read / Write	Set to gtm_trigger_etrap or the empty string when entering trigger context. For

Intrinsic Special Variable	Within Trigger Context	Notes
		more information on using the $ETRAP mechanism for handling errors during trigger execution, refer to "Error Handling during Trigger Execution" (page 603).
$REFERENCE	Read only	Restored at the completion of a trigger.
$TEST	Read only	Restored at the completion of a trigger.
$TLEVEL	Read only	Always >=1 in trigger code; must be the same as the completion of processing a trigger as it was at the start.
$ZTNAME	Read only	Returns the trigger name.
$ZTDATA	Read only	Shows prior state.
$ZTLEVEL	Read only	Shows trigger nesting.
$ZTOLDVAL	Read only	Shows the pre-update value.
$ZTRAP	Read only - ""	Must use $ETRAP in trigger code.
$ZTRIGGEROP	Read only	Shows the triggering command.
$ZTUPDATE	Read only	Lists modified pieces (if requested) for SET.
$ZTVALUE	Read / Write	Can change the eventual applied value for SET.
$ZTWORMHOLE	Read / Write	Holds application context because trigger code has no access to the local variable context.
$ZTSLATE	Read/ Write	Holds outermost transaction context for chained or nested triggers.

Examples of Trigger ISVs

>

The following examples are derived from the FIS Profile application.

Nodes in ^ACN(CID,50) have TYPE in piece 1, CLS in piece 2, FEEPLN in piece 15 and EMPLNO in piece 31. Indexes are ^XACN(CLS,ACN,CID), ^XREF("EMPLCTA",EMPLNO,ACN,TYPE,CID) and ^XREF("FEEPLN",FEEPLN,CID) and use ACN from the first piece of ^ACN(CLS,99). These indexes are maintained with four triggers: one invoked by a KILL or ZKill of an ^ACN(:,50) node and three invoked by SETs to different pieces of ^ACN(:,50) nodes. Note that ACN, CID, CLS and TYPE are required, whereas EMPLNO and FEEPLN can be null, which requires (in our convention) the use of $ZC(254) in indexes. The triggerfile definitions are:

```
+^ACN(cid=:,50) -zdelim="|" -pieces=2 -commands=SET -xecute="Do ^SclsACN50"
+^ACN(cid=:,50) -zdelim="|" -pieces=1,31 -commands=SET -xecute="Do ^SemplnoTypeACN50" +^ACN(cid=:,50) -zdelim="|" -pieces=15 -
commands=SET -xecute="Do ^SfeeplnACN50"
+^ACN(cid=:,50) -commands=KILL,ZKill -xecute="Do ^KACN50"
```

The code in KACN50.m KILLs cross reference indexes when the application deletes any ^ACN(:,50).

```
KACN50 ; KILL of entire ^ACN(:,50) node, e.g., from account deletion
```

```
 ; Capture information
 Set cls=$Piece($ZTOLD,"|",2)                    ; CLS
 Set emplno=$Piece($ZTOLD,"|",31)
 Set:'$Length(emplno) emplno=$ZC(254)            ; EMPLNO
 Set feepln=$Piece($ZTOLD,"|",15)
 Set:'$L(feepln) feepln=$ZC(254)                 ; FEEPLN
 Set type=$Piece($ZTOLD,"|",1)          ; TYPE
 Set acn=$Piece(^ACN(cid,99),"|",1)     ; ACN
 Kill ^XACN(cls,acn,cid)
 Kill ^XREF("EMPLCTA",emplno,acn,type,cid)
 Kill ^XREF("FEEPLN",feepln,cid)
 Quit
```

The routine in SclsACN50.m creates cross references for a SET or a SET $PIECE() that modifies the second piece of ^ACN(:,50).

```
SClsACN50 ; Update to CLS in ^ACN(,50)
 ; Capture information
 Set oldcls=$Piece($ZTOLD,"|",2)                 ; Old CLS
 Set cls=$Piece($ZTVAL,"|",2)                    ; New CLS
 Set acn=$Piece(^ACN(cid,99),"|",1)              ; ACN
 Set processMode=$Piece($ZTWORM,"|",1)           ; Process
 If processMode<2 Kill ^XACN(oldcls,acn,cid)
 Set ^XACN(cls,acn,cid)=""
 Quit
```

Note that the example is written for clarity. Eliminating values that need not be assigned to temporary local variables produces:

```
SclsACN50
 S acn=$P(^ACN(cid,99),"|",1)
 I $P($ZTWORM,"|",1)<2 K ^XACN($P($ZTOLD,"|",2),acn,cid)
 S ^XACN($P($ZTVAL,"|",2),acn,cid)=""
 Q
```

Indeed, this index can simply be included in the (one line) triggerfile specification itself:

```
+^ACN(cid=:,50) -zdelim="|" -pieces=2 -commands=SET -xecute="S
▶ oldcls=$P($ZTOLD,""|"",2),acn=$P(^ACN(cid,99),""|"",1) K:$P($ZTWO,""|"",1)<2 ^XACN(oldcls,acn,cid) S
 ^XACN($P($ZTVAL,""|"",2),acn,cid)="""""
```

▼

In the interest of readability most triggerfile definitions in this chapter are written as complete routines. The code in SemplnoTypeACN50.m handles changes to pieces 1 and 31 of ^ACN(:,50). Note that a SET to ^ACN(:,50) that modifies either or both pieces causes this trigger to execute just once, whereas two sequential SET $Piece() commands, to first modify one piece and then the other cause it to execute twice, at different times, once for each piece.

```
EmplnoTypeACN50 ; Update to EMPLNO and/or TYPE in ^ACN(,50)
 ; Capture information
 Set oldemplno=$Piece($ZTOLD,"|",31)
 Set:'$Length(oldemplno) oldemplno=$ZC(254)
 Set emplno=$Piece($ZTVAL,"|",31)
 Set:'$L(emplno) emplno=$ZC(254)
 Set oldtype=$Piece($ZTOLD,"|",1)
 Set type=$Piece($ZTVAL,"|",1)
 Set acn=$Piece(^ACN(cid,99),"|",1)
 Set processMode=$Piece($ZTWORM,"|",1)
 If processMode<2 Do
 . Kill ^XREF("EMPLNO",oldemplno,acn,oldtype,cid)
 . Set ^XREF("EMPLNO",emplno,acn,type,cid)=""
```

```
  Quit
```

The code in SFeeplnACN50.m handles changes to piece 15.

```
SFeeplnACN50 ; Update to FEEPLN in ^ACN(,50)
 ; Capture information
 Set oldfeepln=$Piece($ZTOLD,"|",15)
 Set:'$Length(oldfeepln) oldfeepln=$ZC(254)
 Set feepln=$Piece($ZTVAL,"|",15)
 Set:'$Length(feepln) feepln=$ZC(254)
 Set processMode=$Piece($ZTWORM,"|",1)
 If processMode<2 Do
 . Kill ^XREF("FEEPLN",oldfeepln,cid) Set ^XREF("FEEPLN",feepln,cid)=""
 Quit
```

Chapter 9. Input/Output Processing

Revision History		
Revision V7.1-003	23 November 2023	• In "Deviceparameter Summary Table" (page 444), Add [NO]HUPENABLE to Deviceparameter Summary Table • In "HUPENABLE" (page 426), Expand description of [NO]HUPENABLE to include SOC devices • In "USE Deviceparameters Summary" (page 435), Add [NO]HUPENABLE to USE Deviceparameters Summary table
Revision V7.0-003	24 June 2022	• In "Device Name Variables" (page 346), move $ZPIN and $ZPOUT under "Device Name Variables" • In "OPEN Deviceparameters" (page 393), New deviceparameter OPTIONS • In "OPTIONS" (page 405), add new OPTIONS deviceparameter • In "OPTIONS" (page 429), made cosmetic changes to USE :OPTIONS • In "Socket Deviceparameter Summary" (page 388), New device parameter OPTIONS • In "USE Deviceparameters" (page 418), New deviceparameter OPTIONS • In "WRITE Command" (page 383), TLSv1.3 and renegotiation • In "$ZPIN" (page 347), Move $ZPIN and $ZPOUT under "Device Name Variables" • In "$ZPOUT" (page 347), Move $ZPIN and $ZPOUT under "Device Name Variables"
Revision V7.0-002	23 March 2022	• In "CTRAP" (page 420), clean up CTRAP behavior description • In "EDITING" (page 423), tweak the effect of CTRAP • In "HUPENABLE" (page 426), add TERMHANGUP as setting for NOHUPENABLE • In "Terminal Deviceparameter Summary" (page 356), correct the syntax of CTRAP • In "USE Deviceparameters Summary" (page 435), add clarification on CTRAP

		• In "WRITE Command" (page 383), correct WRITE /TLS description
Revision V7.0-001	24 November 2021	• In "Close" (page 440), Correct typo in CLOSE dev:SOCKET example; add clarification
		• In "CLOSE Deviceparameters Table" (page 444), GTM-9452 - Add deviceparameter REPLACE to provide the overwrite functionality
		• In "CLOSE Deviceparameters" (page 441), GTM-9452 - Add deviceparameter REPLACE to provide the overwrite functionality
		• In "DELIMITER" (page 394), Add missing "escape" in DELIMITER section
		• In "DETACH" (page 421), change socketpool to YGTMSOCKETPOOL and specify that DETACH should be the only device parameter
		• In "Deviceparameter Summary Table" (page 444), GTM-9452 - Add deviceparameter REPLACE to provide the overwrite functionality
		• In "FIFO Deviceparameter Summary" (page 366), Add deviceparameter (REPLACE on CLOSE) to provide the overwrite functionality.
		• In "IOERROR" (page 400), Correct note under IOERROR
		• In "RENAME" (page 442), clarify
		• In "REPLACE" (page 442), clarify
		• In "Sequential File Deviceparameter Summary" (page 361), GTM-9452 - deviceparameter (REPLACE) to provide the overwrite functionality
		• In "Socket Deviceparameter Summary" (page 388), SOCKET= can also be used by CLOSE
		• In "WIDTH" (page 432), Correct default WIDTH for sockets; also clean up
		• In "WRITE Command" (page 383), add information about WRITE /BLOCK("OFF")
Revision V6.3-012	08 April 2020	• In "HUPENABLE" (page 426), fixed typo
Revision V6.3-011	20 December 2019	• In "OPEN Deviceparameters" (page 393), Add [NO]FFLF device parameter info and example to OPEN text.
		• In "READ/WRITE Operations" (page 359), Fix the description of what $Y is set to. Add a brief note about [NO]FFLF and gtm_nofflf.

		Add note about HEREDOCs and minor edit to SD encryption
		• In "USE Deviceparameters" (page 418), Add [NO]FFLF deviceparameter to USE; add the [NO]HUPENABLE deviceparameter
		• In "Write" (page 439), Remove fflf information already discussed in other places.
Revision V6.3-007	04 February 2019	• In "CTRAP" (page 420), specify that GT.M interrupts process execution when a READ from a device receives a trap character. Add sentence about <CTRL> characters not reaching GT.M
		• In "Socket Device Examples" (page 390), correct the description of start^server.
		• In "Using Socket Devices" (page 378), add YGTMSOCKETPOOL description
Revision V6.3-006	26 October 2018	• In "PIPE Deviceparameter Summary" (page 377), formatting adjustment to the table header
		• In "FILTER" (page 425), UTF-8 mode tweaks.
		• In "FIXED" (page 397), minor corrections.
		• In "Line Terminators " (page 358), UTF-8 mode tweaks.
		• In "READ" (page 437), UTF-8 mode tweaks.
		• In "READ * Command" (page 438), UTF-8 mode tweaks.
		• In "RECORDSIZE" (page 408), UTF-8 mode tweaks.
		• In "TERMINATOR" (page 431), UTF-8 mode tweaks.
		• In "WIDTH" (page 432), UTF-8 mode tweaks.
		• In "Write" (page 439), UTF-8 mode tweaks.
Revision V6.3-005	29 June 2018	• In "Write" (page 439), replicate UTF-8 bullets from the command section, add information about concatenation, remove some VMS-specific text
Revision V6.3-004	23 March 2018	• In "IOERROR" (page 400), change strexpr to expr; add more information about $DEVICE.
		• In "WRITE Command" (page 383), Updated the WRITE /LISTEN entry to reflect the removal of GT.M's listen queue size limitation.
Revision V6.3-003	12 December 2017	• In "Message Management" (page 379), remove extra space

		• In "PIPE Device Examples " (page 371), move an example to the correct location and fix a typing error in the nestin example.
		• In "Socket Device Examples" (page 390), make the downloadable socket example self-explanatory and a bug fix.
		• In "Socket Deviceparameter Summary" (page 388), add a Socket Management Deviceparameter summary table
		• In "STDERR" (page 411), add information about PIPE device behavior when there is no STDERR specified.
		• In "$Device " (page 349), generalize the text and examples
Revision V6.3-001	20 March 2017	• In "ATTACH" (page 418), changed I[O]CHSET to [I|O]CHSET as the alternatives for the character set device parameter are CHSET, ICHSET and OCHSET so the corresponding text representation is [I|O]CHSET.
		• In "CENABLE" (page 419), add CHSET to USE deviceparameter list
		• In "CLOSE Deviceparameters Table" (page 444), removed the entry for the deprecated ERASETAPE and SPACE deviceparameters.
		• In "DETACH" (page 421), changed I[O]CHSET to [I|O]CHSET as the alternatives for the character set device parameter are chset, ichset and ochset so the corresponding text representation is [i|o]chset.
		• In "ICHSET" (page 399), specified that ICHSET can also be used as a USE deviceparameter.
		• In "INSERT" (page 427), added the description of the INSERT deviceparameter for terminal devices.
		• In "IOERROR" (page 400), added information about when error trapping gets enabled.
		• In "OCHSET" (page 404), specified that OCHSET can also be used as a USE deviceparameter.
		• In "OPEN Deviceparameter Table" (page 416), arranged the deviceparameters entries alphabetically.
		• In "READ/WRITE Operations" (page 359), added a point on SD WRITE behavior after setting $X to a value greater than WIDTH or reducing WIDTH to less than $X.

		• In "Socket Deviceparameter Summary" (page 388), removed CONNECT and LISTEN from Socket Establishment/Disconnect Deviceparameters.
		• In "Terminal Deviceparameter Summary" (page 356), added deviceparameter entries for [NO]EDITING, [NO]EMPTERM, and [NO]INSERT.
		• In "USE Deviceparameters Summary" (page 435), corrected the alphabetically ordering, removed the deprecated deviceparameters, and added and entry for [I|O]CHSET.
		• In "Using PIPE Devices" (page 369), pipe device stderr handling
		• In "Using Socket Devices" (page 378), added updates for TLS sockets.
		• In "WRITE Command" (page 383), removed references to HP-UX and Solaris
		• In "ZIBFSIZE" (page 415), corrected the description.
Revision V6.2-001	27 February 2015	• In "Socket Device Examples" (page 390), specified that the ZSHOW "D" command reports available information on both the local and remote sides of a TCP socket and enhanced the downloadable example to include more debugging information.
		• Updated "WRITE Command" (page 383) for WRITE /PASS, WRITE /ACCEPT, and WRITE / TLS.
		• In "OPEN Deviceparameters" (page 393) and "USE Deviceparameters" (page 418), added "IKEY" (page 400),"KEY" (page 400), "OKEY" (page 404) keywords.
		• In "REWIND" (page 408), specified that when $PRINCIPAL identifies a device that supports REWIND, the REWIND or INREWIND device parameters perform a REWIND of the input and OUTREWIND performs a REWIND of the output.
		• In "DESTROY" (page 441), added the behavior of a sequential disk device CLOSEd with the NODESTROY.
		• In "READ/WRITE Operations" (page 359), specified that WRITE works at the current file position, whether attained with APPEND, REWIND or SEEK.
		• In "TRUNCATE" (page 413), specified that the TRUNCATE device parameter on a USE $PRINCIPAL command works on a stdout device when the device supports the action.

		• In "DESTROY" (page 441), moved the reference to the behavior of sequential disk devices CLOSEd with NODESTROY to "Open" (page 391).
Revision V6.1-000	28 August 2014	• Added information about using LOCAL sockets in "ZDELAY" (page 415), "ZBFSIZE" (page 434), "LISTEN" (page 402), "Close" (page 440), "CONNECT" (page 394). • Improved the documentation of SOCKET devices in "WRITE Command" (page 383). • Added the downloadable sockexamplemulti3.m example in "Socket Device Examples" (page 390). • In "Document Conventions" (page 353), added the mnemonics for SOC(LOCAL) and SOC(TCP) for LOCAL and TCP sockets. • In FIFO Characteristics (page 364), added a note about FIFO access permissions.
Revision V6.0-003	24 February 2014	• Added the descriptions of [NO]FOLLOW and [NO]EMPT[ERM] deviceparameters of OPEN and USE. • Improved the descriptions of "STREAM" (page 412) and "FIXED" (page 397). • In "CONNECT" (page 394), added information about IPv6 support. • In "Writing Binary Files" (page 360), improved the example for performing a binary copy. • Under Using Sequential Files, added a new section called "READ/WRITE Operations" (page 359). • Improved the descriptions of "WRAP" (page 414) and "FIXED" (page 397).
Revision V6.0-001	21 March 2013	In "EXCEPTION" (page 395), added a note about the handling of non-fatal errors.
Revision V6.0-000	19 November 2012	• In "PIPE Characteristics" (page 370), improved the description of WRITE /EOF. • In "Using Null Devices" (page 367), added information about /dev/zero, /dev/random, and /dev/urandom devices.

This chapter describes the following topics which relate to input and output processing:

• Input/Output Intrinsic Special Variables, and their Maintenance.

GT.M provides several intrinsic special variables that allow processes to examine, and in some cases change, certain aspects of the input/output (I/O) processing. The focus in this chapter is how GT.M handles the standard ones, such as $IO, $X, $Y, and those that are GT.M-specific (for example, $ZA, $ZB).

- Input/Output Devices

  Each device type supported by GT.M responds to a particular subset of deviceparameters, while ignoring others. Devices may be programmed in a device-specific manner, or in a device-independent manner. This chapter discusses each device type, and provides tables of their deviceparameters.

- Input/Output Commands and their Deviceparameters

  GT.M bases its I/O processing on a simple character stream model. GT.M does not use any pre-declared formats. This chapter describes the GT.M I/O commands OPEN, USE, READ, WRITE, and CLOSE.

OPEN, USE, and CLOSE commands accept deviceparameters, which are keywords that permit a GT.M program to control the device state. Some deviceparameters require arguments. The current ANSI standard for GT.M does not define the deviceparameters for all devices. This chapter includes descriptions of the GT.M deviceparameters in the sections describing each command.

> **Note**
>
> The term "device" can refer to an entity manipulated by application code using Open, Use, Close, Read and Write commands as well as a device from the perspective of the operating system. We endeavor herein to always make it clear from the context which meaning is intended.

I/O Intrinsic Special Variables

GT.M intrinsic special variables provide a mean for application code to communicate and manage the state of a device.

Device Name Variables

GT.M provides three intrinsic special variables that identify devices.

$Io

$I[O] contains the name of the current device specified by the last USE command. A SET command cannot modify $IO. USE produces the same $IO as USE $PRINCIPAL, but $P is the preferred construct.

$Principal

A process inherits three open file descriptors from its parent - STDIN, STDOUT and STDERR - which can all map to different files or devices. GT.M provides no way for M application to access STDERR. Although STDIN and STDOUT may map to different devices, files, sockets, pipes, etc. in the operating system, M provides for only device $PRINCIPAL, to refers to both. At process startup, and when $PRINCIPAL is selected with a USE command, READ commands apply to STDIN and WRITE commands apply to STDOUT. The device type of the standard input determines which USE deviceparameters apply to $PRINCIPAL.

For an interactive process, $PRINCIPAL is the user's terminal. GT.M ignores a CLOSE of the principal device. GT.M does not permit a SET command to modify $PRINCIPAL.

0 is an alternate for $PRINCIPAL (for example, USE 0). FIS recommends that application code use $PRINCIPAL. The environment variable gtm_principal can be used to set a string reported by GT.M for $PRINCIPAL and which can be used in lieu of $PRINCIPAL for the USE command.

$ZIO

$ZIO contains the translated name of the current device, in contrast to $IO, which contains the name as specified by the USE command.

$ZPIN

When $PRINCIPAL has different input/output devices, the USE command recognizes intrinsic special variable $ZPIN to apply appropriate deviceparameters to the input side of $PRINCIPAL. A USE with $ZPIN sets $IO to $PRINCIPAL for READs and WRITEs from the input and output side of $PRINCIPAL. $ZSOCKET() also accepts $ZPIN as its first argument and, if the device is a split SOCKET device, supplies information on the input SOCKET device. In any context other than USE or $ZSOCKET(), or if $PRINCIPAL is not a split device, $PRINCIPAL, $ZPIN and $ZPOUT are synonyms. In the case of a split $PRINCIPAL, $ZPIN returns the value of $PRINCIPAL followed by the string "< /" Any attempt to OPEN $ZPIN results in a DEVOPENFAIL error.

$ZPOUT

When $PRINCIPAL has different input/output devices, the USE command recognizes intrinsic special variables $ZPOUT to apply appropriate deviceparameters to the output side of $PRINCIPAL. A USE with $ZPOUT sets $IO to $PRINCIPAL for READs and WRITEs from the input and output side of $PRINCIPAL. $ZSOCKET() also accepts $ZPOUT as its first argument and, if the device is a split SOCKET device, supplies information on the output SOCKET device. In any context other than USE or $ZSOCKET(), or if $PRINCIPAL is not a split device, $PRINCIPAL, $ZPIN and $ZPOUT are synonyms. In the case of a split $PRINCIPAL, $ZPOUT returns the value of $PRINCIPAL followed by the string "> /" Any attempt to OPEN $ZPOUT results in a DEVOPENFAIL error.

Cursor Position Variables

GT.M provides two intrinsic special variables for determining the virtual cursor position. $X refers to the current column, while $Y refers to the current row.

$X

$X contains an integer value ranging from 0 to 65,535, specifying the horizontal position of a virtual cursor in the current output record. $X=0 represents the initial position on a new record or row.

Every OPENed device has a $X. However, GT.M only has access to $X of the current device.

Generally, in M mode GT.M increments $X for every character written to and read from the current device; see below for behavior of a UTF-8 mode device. GT.M format control characters, FILTER, and the device WIDTH and WRAP also have an effect on $X.

As $X is only a counter to help a program track output, SET $X does not reposition the cursor or perform any other IO. Conversely, if a sequence of characters sent to a terminal or other device with a WRITE causes it to be repositioned except as described below, $X will not reflect this change.

$Y

$Y contains an integer value ranging from 0 to 65,535, specifying the vertical position of a virtual cursor in the current output record. $Y=0 represents the top row or line.

Every OPEN device has a $Y. However, GT.M only accesses $Y of the current device.

When GT.M finishes the logical record in progress, it generally increments $Y. GT.M recognizes the end of a logical record when it processes certain GT.M format control characters, or when the record reaches its maximum size, as determined by the device WIDTH, and the device is set to WRAP. The definition of "logical record" varies from device to device. For an exact definition, see the sections on each device type. FILTER and the device LENGTH also have an effect on $Y.

As $Y is only a counter to help a program track output, SET $Y does not reposition the cursor or perform any other IO. Conversely, if a sequence of characters sent to a terminal or other device with a WRITE causes it to be repositioned except as described below, $Y will not reflect this change.

Maintenance of $X and $Y

The following factors affect the maintenance of the virtual cursor position ($X and $Y):

- The bounds of the virtual "page"

- Format control characters

- GT.M character filtering

Each device has a WIDTH and a LENGTH that define the virtual "page." The WIDTH determines the maximum size of a record for a device, while the LENGTH determines how many records fit on a page. GT.M starts a new record when the current record size ($X) reaches the maximum WIDTH and the device has WRAP enabled. When the current line ($Y) reaches the maximum LENGTH, GT.M starts a new page.

GT.M has several format control characters (used in the context of a WRITE command) that allow the manipulation of the virtual cursor. For all I/O devices, the GT.M format control characters do the following:

- ! Sets $X to zero (0) and increments $Y, and terminates the logical record in progress. The definition of "logical record" varies from device to device, and is discussed in each device section.

- # Sets $X and $Y to zero (0), and terminates the logical record in progress.

- ?n If n is greater than $X, writes n-$X spaces to the device, bringing $X to n. If n is less than or equal to $X, ?n has no effect. When WRAP is enabled and n exceeds the WIDTH of the line, WRITE ?n increments $Y and sets $X equal to n#WIDTH, where # is the GT.M modulo operator.

In UTF-8 mode, GT.M maintains $X in the following measurement units:

Devices	Input	Output
FIFO	code points	display columns
PIPE	code points	display columns
SD	code points	display columns
SOC	code points	code points
TRM	display columns	display columns

GT.M provides two modes of character filtering. When filtering is enabled, certain <CTRL> characters and/or escape sequences have special effects on the cursor position (for example, <BS> (ASCII 8) may decrement $X, if $X is non-zero). For more information on write filtering, refer to "FILTER" (page 425).

Status Variables

GT.M provides several I/O status variables that convey information about the commands operating on the device.

$Device

If the last commanded resulted in no error-condition, the value of $DEVICE, when interpreted as a truth-value is 0 (FALSE). If the status of the device reflect any error-condition, the value of $DEVICE, when interpreted as a truth-value is 1 (TRUE). When $DEVICE starts with 1, it's followed by a comma (,) and then by the text that would be in $ZSTATUS at the time of the error.

Examples:

0 indicates for READ with a zero (0) timeout that available data has been read.

"1,Device detected EOF" indicates the device reached an end-of-file condition.

$Key

$K[EY] contains the string that terminated the most recent READ command from the current device (including any introducing and terminating characters). If no READ command is issued to the current device or if no terminator is used, the value of $KEY is an empty string.

For PIPE:

$KEY contains the UNIX process id of the created process shell which executes the command connected to the PIPE.

For more information, refer to "$Key" (page 296).

$ZA

$ZA contains the status of the last read on the device. The value is a decimal integer with a meaning as follows:

For Terminal I/O:

0: Indicates normal termination of a read operation

1: Indicates a parity error

2: Indicates the terminator sequence was too long

9: Indicates a default for all other errors

For Sequential Disk :

0: Indicates normal termination of a read operation

9: Indicates a failure of a read operation

For FIFO:

0: Indicates normal termination or time out

9: Indicates a failure of a read operation

For SOCKET:

0: Indicates normal termination or time out

9: Indicates failure of a read operation

For PIPE:

0: Indicates normal termination or time out when using READ x:n, where n >0

9: Indicates failure of a READ x or READ x:n, where n>0

9: Indicates failure of a WRITE where the pipe is full and the WRITE would block

> **Caution**
>
> $ZA refers to the status of the current device. Therefore, exercise care in sequencing USE commands and references to $ZA.

$ZB

$ZB contains a string specifying the input terminator for the last terminal READ. $ZB is null, and it is not maintained for devices other than terminals. $ZB may contain any legal input terminator, such as <CR> (ASCII 13) or an escape sequence starting with <ESC> (ASCII 27), from zero (0) to 15 bytes in length. $ZB is null for any READ terminated by a timeout or any fixed-length READ terminated by input reaching the maximum length.

$ZB contains the actual character string, not a sequence of numeric ASCII codes.

If a device is opened with CHSET set to UTF-8 or UTF-16*, $ZB contains the bad character if one is encountered. This holds true for sockets, sequential files (and thus FIFOs and PIPEs) and terminals.

Example:

```
set zb=$zb for i=1:1:$length(zb) write !,i,?5,$ascii(zb,i)
```

This example displays the series of ASCII codes for the characters in $ZB.

$ZB refers to the last READ terminator of the current device. Therefore, be careful when sequencing USE commands and references to $ZB.

$ZEOF

$ZEOF contains a truth-valued expression indicating whether the last READ operation reached the end-of-file. $ZEOF is TRUE(1) at EOF and FALSE (0) at other positions. GT.M does not maintain $ZEOF for terminal devices.

$ZEOF refers to the end-of-file status of the current device. Therefore, be careful when sequencing USE commands and references to $ZEOF.

$ZEOF is set for terminals if the connection dropped on read.

I/O Devices

Each device type supported by GT.M responds to a particular subset of deviceparameters, while ignoring others. Devices may be programmed in a device-specific manner, or in a device-independent manner. Device-specific I/O routines are intended for use with only one type of device. Device-independent I/O routines contain appropriate deviceparameters for all devices to be supported by the function, so the user can redirect to a different device output while using the same program.

GT.M supports the following I/O device types:

- Terminals and Printers

- Sequential Disk Files

- FIFOs

- Null Devices

- Socket Devices

- PIPE Devices

I/O Device Recognition

GT.M OPEN, USE, and CLOSE commands have an argument expression specifying a device name.

During an OPEN, GT.M attempts to resolve the specified device names to physical names. When GT.M successfully resolves a device name to a physical device, that device becomes the target of the OPEN. If the device name contains a dollar sign ($), GT.M attempts an environment variable translation; the result becomes the name of the device. If it does not find such an environment variable, it assumes that the dollar sign is a part of the filename, and opens a file by that name.

> **Note**
>
> Note: GT.M resolves the device name argument for menemonicspace devices (SOCKET or PIPE) to a arbitrary handle instead of a physical name.

Once a device is OPEN, GT.M establishes an internal correspondence between a name and the device or file. Therefore, while the device is OPEN, changing the translation of an environment variable in the device specification does not change the device.

The following names identify the original $IO for the process:

- $PRINCIPAL

- 0

Device Specification Defaults

GT.M uses standard filenames for device specifiers.

The complete format for a filename is:

```
/directory/file
```

If the expression specifying a device does not contain a complete filename, the expression may start with an environment variable that translates to one or more leading components of the filename. GT.M applies default values for the missing components.

If the specified file is not found, it is created unless READONLY is specified.

The GT.M filename defaults are the following:

Directory: Current working directory

File: No default (user-defined filename)

Filetype: No default (user-defined filetype)

How I/O Device parameters Work

I/O deviceparameters either perform actions that cause the device to do something (for example, CLEARSCREEN), or specify characteristics that modify the way the device subsequently behaves (for example, WIDTH). When an I/O command has multiple action deviceparameters, GT.M performs the actions in the order of the deviceparameters within the command argument. When a command has characteristic deviceparameters, the last occurrence of a repeated or conflicting deviceparameter determines the characteristic.

Deviceparameters often relate to a specific device type. GT.M ignores any deviceparameters that do not apply to the type of the device specified by the command argument. Specified device characteristics are in force for the duration of the GT.M image, or until modified by an OPEN, USE, or CLOSE command.

When reopening a device that it previously closed, a GT.M process restores all characteristics not specified on the OPEN to the values the device had when it was last CLOSEd. GT.M treats FIFO, PIPE, and SD differently and uses defaults for unspecified device characteristics on every OPEN (that is, GT.M does not retain devices characteristics on a CLOSE of SD, FIFO, and PIPE).

The ZSHOW command with an argument of "D" displays the current characteristics for all devices OPENed by the process. ZSHOW can direct its output into a GT.M variable. For more information on ZSHOW, refer to "ZSHow" (page 193).

Abbreviating Deviceparameters

> ⚠️ **Important**
>
> Most Z* deviceparameters have the same functionality as their counterparts and are supported for compatibility reasons.

GT.M deviceparameters do not have predefined abbreviations. GT.M recognizes deviceparameters using a minimum recognizable prefix technique. Most deviceparameters may be represented by four leading characters, except ERASELINE, all deviceparameters starting with WRITE, and Z* deviceparameters in a mnemonicspace (such as SOCKET). The four leading characters recognized do not include a leading NO for negation.

For compatibility with previous versions, GT.M may recognize certain deviceparameters by abbreviations shorter than the minimum. While it is convenient in Direct Mode to use shorter abbreviations, FIS may add additional deviceparameters, and therefore, recommends all programs use at least four characters. Because GT.M compiles the code, spelling out deviceparameters completely has no performance penalty, except when used with indirection or XECUTEd arguments.

Document Conventions

This chapter uses the following mnemonics to describe when a deviceparameter applies:

TRM: Valid for terminals

SD: Valid for sequential disk files

FIFO: Valid for FIFOs

NULL: Valid for null devices

SOC: Valid for both socket devices (TCP and LOCAL)

SOC(LOCAL): Valid for LOCAL sockets devices

SOC(TCP): Valid for TCP sockets devices

PIPE: Valid for PIPE devices

> **Note**
>
> Lower case "pipe" refers to a UNIX pipe and the upper case "PIPE" to the GT.M device.

Some of the deviceparameter defaults shown are the basic operating system defaults, and may be subject to modification before the invocation of GT.M.

Device-Independent Programming

When a user may choose a device for I/O, GT.M routines can take one of two basic programming approaches.

- The user selection directs the program into different code branches, each of which handles a different device type.

- The user selection identifies the device. There is a single code path written with a full complement of deviceparameters to handle all selectable device types.

The latter approach is called device-independent programming. To permit device independent programming, GT.M uses the same deviceparameter for all devices that have an equivalent facility, and ignores deviceparameters applied to a device that does not support that facility.

Example:

```
OPEN dev:(EXCE=exc:REWIND:VARIABLE:WRITEONLY)
```

This example OPENs a device with deviceparameters that affect different devices. The EXCEPTION has an effect for all device types. When dev is a terminal or a null device, GT.M ignores the other deviceparameters. When dev is a sequential file on disk, GT.M uses REWIND and VARIABLE. This command performs a valid OPEN for all the different device types.

Using Terminals

A GT.M process assigns $PRINCIPAL to the UNIX standard input of the process (for READ) and standard output (for WRITE). For a local interactive process, $PRINCIPAL identifies the "terminal" from which the user is signed on.

While all terminals support the CTRAP deviceparameter, only $PRINCIPAL supports CENABLE. While CTRAP allows terminal input to redirect program flow, CENABLE allows the terminal user to invoke the Direct Mode.

Directly connected printers often appear to GT.M as a terminal (although printers generally do not provide input) regardless of whether the printer is connected to the computer with a high speed parallel interface, or an asynchronous terminal controller.

Setting Terminal Characteristics

GT.M does not isolate its handling of terminal characteristics from the operating system environment at large. GT.M inherits the operating system terminal characteristics in effect at the time the GT.M image is invoked. When GT.M exits, the terminal characteristics known by the operating system are restored.

However, if the process temporarily leaves the GT.M environment with a ZSYSTEM command , GT.M does not recognize any changes to the terminal characteristics left by the external environment. This may cause disparities between the physical behavior of the terminal, and the perceived behavior by GT.M.

UNIX enforces standard device security for explicit OPENs of terminals other than the sign-in terminal ($PRINCIPAL). If you are unable to OPEN a terminal, contact your system manager.

USE of a terminal causes the device driver to flush the output buffer. This feature of the USE command provides routine control over the timing of output, which is occasionally required. However, it also means that redundant USE commands may induce an unnecessary performance penalty. Therefore, FIS recommends restricting USE commands to redirecting I/O, modifying deviceparameters, and initiating specifically required flushes.

The terminal input buffer size is fixed at 1024 on UNIX and a variable read terminates after 1023 characters.

Setting the environment variable TERM

The environment variable $TERM must specify a terminfo entry that accurately matches the terminal (or terminal emulator) settings. Refer to the terminfo man pages for more information on the terminal settings of the platform where GT.M needs to run.

Some terminfo entries may seem to work properly but fail to recognize function key sequences or position the cursor properly in response to escape sequences from GT.M. GT.M itself does not have any knowledge of specific terminal control characteristics. Therefore, it is important to specify the right terminfo entry to let GT.M communicate correctly with the terminal. You may need to add new terminfo entries depending on their specific platform and implementation. The terminal (emulator) vendor may also be able to help.

GT.M uses the following terminfo capabilities. The full variable name is followed by the capname in parenthesis:

```
auto_right_margin(am), clr_eos(ed), clr_eol(el), columns(cols), cursor_address(cup), cursor_down(cud1),cursor_left(cub1),
 cursor_right(cuf1), cursor_up(cuu1), eat_newline_glitch(xenl), key_backspace(kbs), key_dc(kdch1),key_down(kcud1),
 key_left(kcub1), key_right(kcuf1), key_up(kcuu1), key_insert(kich1), keypad_local(rmkx),keypad_xmit(smkx), lines(lines).
```

GT.M sends keypad_xmit before terminal reads for direct mode and READs (other than READ *) if EDITING is enabled. GT.M sends keypad_local after these terminal reads.

Logical Records for Terminals

A logical record for a terminal equates to a line on the physical screen. The WIDTH device characteristic specifies the width of the screen, while the LENGTH device characteristic specifies the number of lines on the screen.

READ * Command for Terminals

If the terminal has ESCAPE sequencing enabled, and the input contains a valid escape sequence or a terminator character, GT.M stores the entire sequence in $ZB and returns the ASCII representation of the first character.

Example:

```
GTM>kill
GTM>use $principal:escape
GTM>read *x set zb=$zb zwrite
(Press the F11 key on the VT220 terminal keyboard)
x=27
zb=$C(27)_"[23~"
```

This enters an escape sequence in response to a READ *. The READ * assigns the code for <ESC> to the variable X. GT.M places the entire escape sequence in $ZB. As some of the characters are not graphic, that is, visible on a terminal, the example transfers the contents of $ZB to the local variable ZB and uses a ZWRITE so that the non-graphic characters appear in $CHAR() format.

When escape processing is disabled, READ *x returns 27 in x for an <ESC>. If the escape introducer is also a TERMINATOR, $ZB has a string of length one (1), and a value of the $ASCII() representation of the escape introducer; otherwise, $ZB holds the empty string. GT.M stores the remaining characters of the escape sequence in the input stream. A READ command following a READ * command returns the remaining characters of the escape sequence.

Example:

```
GTM>kill
GTM>use $principal:(noescape:term=$char(13))
GTM>read *x set zb=$zb read y:0 zwrite
(Press the F11 key on the terminal keyboard)
[23~x=27
y="[23~"
zb=""
GTM>use $principal:noecho read *x set zb=$zb read y:0 use $principal:echo zwrite
x=27
y="[23~"
zb=""
GTM>read *x set zb=$zb use $principal:flush read y:0 zwrite
x=27
y=""
zb=""
```

While the first READ Y:0 picks up the sequence after the first character, notice how the graphic portion of the sequence appears on the terminal – this is because the READ *X separated the escape character from the rest of the sequence thus preventing the terminal driver logic from recognizing it as a sequence, and suppressing its echo. The explicit suppression of echo removes this visual artifact. In the case of the final READ *X, the FLUSH clears the input buffer so that it is empty by the time of the READ Y:0.

READ X#maxlen Command for Terminals

Generally, GT.M performs the same maintenance on $ZB for a READ X#maxlen as for a READ. However, if the READ X#maxlen terminates because the input has reached the maximum length, GT.M sets $ZB to null. When the terminal has ESCAPE sequencing enabled, and the input contains an escape sequence, GT.M sets $ZB to contain the escape sequence.

Terminal Deviceparameter Summary

The following tables provide a brief summary of deviceparameters for terminals, grouped into related areas. For detailed information, refer to "Open" (page 142), "Use" (page 151), and "Close" (page 440).

Error Processing Deviceparameters		
DEVICEPARAMETER	**COMMAND**	**COMMENT**
EXCEPTION=expr	O/U/C	Controls device-specific error handling.

Interaction Management Deviceparameters		
DEVICEPARAMETER	**COMMAND**	**COMMENT**
[NO]CENABLE	U	Controls whether <CTRL-C> on $PRINCIPAL causes GT.M to go to direct mode.
CTRAP=$CHAR(intexpr[,...])	U	Controls vectoring on trapped <CTRL> characters.
[NO]EDITING	U	Controls the editing mode for $PRINCIPAL.
[NO]EMPTERM	U	Control whether an "Erase" character on an empty input line should terminate a READ or READ # command.
[NO]ESCAPE	U	Controls escape sequence processing.
[NO]INSERT	U	Controls insert or overstrike on input.
[NO]PASTHRU	U	Controls interpretation by the operating system of special control characters (for example <CTRL-B>).
[NO]TERMINATOR[=expr]	U	Controls characters that end a READ

Flow Control Deviceparameters		
DEVICEPARAMETER	**COMMAND**	**COMMENT**
[NO]CONVERT	U	Controls forcing input to uppercase.
[NO]FILTER	U	Controls some $X, $Y maintenance.
FLUSH	U	Clears the typeahead buffer.
[NO]HOSTSYNC	U	Controls host's use of XON/XOFF.
[NO]READSYNC	U	Controls wrapping READs in XON/XOFF.
[NO]TTSYNC	U	Controls input response to XON/XOFF.
[NO]TYPEAHEAD	U	Controls unsolicited input handling.

Screen Management Deviceparameters		
DEVICEPARAMETER	**COMMAND**	**COMMENT**
CLEARSCREEN	U	Clears from cursor to end-of-screen.
DOWNSCROLL	U	Moves display down one line.
[NO]ECHO	U	Controls the host echo of input.
ERASELINE	U	Clears from cursor to end-of-line.
[Z]LENGTH=intexpr	U	Controls maximum number of lines on a page ($Y).
UPSCROLL	U	Moves display up one line.
[Z]WIDTH=intexpr	U	Controls the maximum width of an output line ($X).
[Z][NO]WRAP	U	Controls handling of output lines longer than the maximum width.
X=intexpr	U	Positions the cursor to column intexpr.
Y=intexpr	U	Positions the cursor to row intexpr.

O: Applies to the OPEN command

U: Applies to the USE command

C: Applies to the CLOSE command

Terminal Examples

This section contains examples of GT.M terminal handling.

Example:

```
use $principal:(exception="zg "_$zl_":C^MENU")
```

This example USEs the principal device, and sets up an EXCEPTION handler. When an error occurs, it transfers control to label C in the routine ^MENU at the process stack level where the EXCEPTION was established.

Example:

```
use $principal:(x=0:y=0:clearscreen)
```

This example positions the cursor to the upper left-hand corner and clears the entire screen.

Example:

```
use $principal:(noecho:width=132:wrap)
```

This example disables ECHOing, enables automatic WRAPping, and sets the line width to 132 characters.

Note that GT.M enables WRAP automatically when you specify the WIDTH deviceparameter.

Example:

```
use $principal:nocenable
```

This example disables <CTRL-C>.

UsingSequential Files

GT.M provides access to sequential files. These files allow linear access to records. Sequential files are used to create programs, store reports, and to communicate with facilities outside of GT.M.

Setting Sequential File Characteristics

The ANSI standard specifies that when a process CLOSEs and then reOPENs a device, GT.M restores any characteristics not explicitly specified with deviceparameters to the values they had prior to the last CLOSE. However, because it is difficult for a large menu-driven application to ensure the previous OPEN state, GT.M always sets unspecified sequential file characteristics to their default value on OPEN. This approach also reduces potential memory overhead imposed by OPENing and CLOSEing a large number of sequential files during the life of a process.

GT.M does not restrict multiple OPEN commands. However, if a file is already open, GT.M ignores attempts to modify sequential file OPEN characteristics, except for RECORDSIZE and for deviceparameters that also exist for USE.

Sequential files can be READONLY, or read/write (NOREADONLY).

Sequential files can be composed of either FIXED or VARIABLE (NOFIXED) length records. By default, records have VARIABLE length.

UNIX enforces its standard security when GT.M OPENs a sequential file. This includes any directory access required to locate or create the file. If you are unable to OPEN a file, contact your system manager.

Sequential File Pointers

Sequential file I/O operations use a construct called a file pointer. The file pointer logically identifies the next record to read or write. OPEN commands position the file pointer at the beginning of the file (REWIND) or at the end-of-file (APPEND). APPEND cannot reposition a file currently open. Because the position of each record depends on the previous record, a WRITE destroys the ability to reliably position the file pointer to subsequent records in a file. Therefore, by default (NOTRUNCATE), GT.M permits WRITEs only when the file pointer is positioned at the end of the file.

A file that has been previously created and contains data that should be retained can also be opened with the device parameter APPEND.

If a device has TRUNCATE enabled, a WRITE issued when the file pointer is not at the end of the file causes all contents after the current file pointer to be discarded. This effectively moves the end of the file to the current position and permits the WRITE.

Line Terminators

LF ($CHAR(10)) terminates the logical record for all M mode sequential files, TRM, PIPE, and FIFO. For non FIXED format sequential files and terminal devices for which character set is not M, all the standard Unicode® line terminators terminate the logical record. These are U+000A (LF), U+0000D (CR), U+000D followed by U+000A (CRLF), U+0085 (NEL), U+000C (FF), U+2028 (LS) and U+2029 (PS).

READ/WRITE Operations

The following table describes all READ and WRITE operations for STREAM, VARIABLE, and FIXED format sequential files having automatic record termination enabled (WRAP) or disabled (NOWRAP).

Command	WRAP or NOWRAP	STREAM or VARIABLE format file behavior		FIXED format file behavior
READ format or WRITE or WRITE *	WRAP	Write the entire argument, but anytime $X is about to exceed WIDTH: insert a <LF> character, set $X to 0, increment $Y		Similar to VARIABLE but no <LF>
READ format or WRITE or WRITE *	NOWRAP	Update $X based on STREAM or VARIABLE format as described below		Same as VARIABLE
		STREAM: Write all of the argument with no truncation nor with a line terminator being inserted. Add length of argument to $X.	VARIABLE ($X=WIDTH): Write up to WIDTH-$X characters. Write no more output to the device until a WRITE ! or a SET $X makes $X less than WIDTH.	
READ or WRITE !	either	Write <LF>, set $X to 0, increment $Y		Write PAD bytes to bring the current record to WIDTH
WRITE #	either	Write <FF>,<LF> or <FF>, set $X and $Y to 0. See the documentation for the USE and OPEN commands to understand how the [NO]FFLF device parameter and the gtm_nofflf environment variable affects the operation of "WRITE #". If the device parameter is unused, and the environment variable is unset, the default behavior of "WRITE #" is to write <FF>,<LF>.		Write PAD bytes to bring the current record to WIDTH, then a <FF> followed by WIDTH-1 PAD bytes
CLOSE	either	After a WRITE, if $X > 0, Write <LF>		After a WRITE, if $X >0, perform an implicit "WRITE !" adding PAD bytes to create a full record. If you need to avoid trailing PAD bytes set $X to 0 before closing a FIXED format file.
READ X	either	Return characters up to $X=WIDTH, or until encountering an <LF> or EOF. If <LF> encountered, set $X to 0, increment $Y		Return WIDTH characters; no maintenance of $X and $Y, except that EOF increments $Y
READ X#len	either	Return characters up to the first of $X=WIDTH or len characters, or encountering a <LF> or EOF; if up to len characters or EOF update $X, otherwise set $X to 0 and increment $Y		Return MIN(WIDTH, len) characters; no maintenance of $X and $Y, except that EOF increments $Y
READ *X	either	Return the code for one character and increment $X, if WIDTH=$X or <LF> encountered, set $X=0, increment $Y; if EOF return -1		Return the code for one character, if EOF return -1; no maintenance of $X and $Y, except that EOF increments $Y

> **Note**
>
> - EOF == end-of-file; <FF>== ASCII form feed; <LF> == ASCII line feed;

- In M mode, and by default in UTF-8 mode PAD == <SP> == ASCII space.

- "READ format" in this table means READ ? or READ <strlit>

- A change to WIDTH implicitly sets WRAP unless NOWRAP follows in the deviceparameter list

- In VARIABLE and STREAM mode, READ (except for READ *) never returns <LF> characters

- In M mode, the last setting of RECORDSIZE or WIDTH for the device determines WIDTH

- In M Mode, a WRITE to a sequential device after setting $X to a value greater than the device WIDTH or a reducing WIDTH to less than the current $X acts as if the first character caused $X to exceed the WIDTH induces an immediate WRAP, if WRAP is enabled

- In UTF-8 mode, RECORDSIZE is in bytes and WIDTH is in characters and the smaller acts as the WIDTH limit in the table.

- In UTF-8 mode, FIXED mode writes <SP> to the RECORDSIZE when the next character won't fit.

- In UTF-8 mode, all READ forms do not return trailing PAD characters.

- In UTF-8 mode, all characters returned by all forms of FIXED mode READ are from a single record.

- WRITE for a Sequential Disk (SD) device works at the current file position, whether attained with APPEND, REWIND or SEEK.

- GT.M manages any BOM for UTF mode files by ensuring they are at the beginning of the file and produces a BOMMISMATCH error for an attempt to change the byte-ordering on OPEN for an existing file.

- An attempt to OPEN a non-zero length file WRITEONLY without either NEWVERSION or TRUNCATE in UTF mode produces an OPENDEVFAIL due to the fact that any existing BOM information cannot be verified.

- GT.M SD encryption, because of the state information associated with encryption processing, requires encrypted files to be WRITEn or READ from the beginning rather than from an arbitrary position.

- HEREDOCs in shell scripts that drive input to GT.M present to GT.M as SD files, and by default terminate with a success - zero (0) status unless GT.M terminates with a ZHALT that supplies an alternative status.

Writing Binary Files

To write a binary data file, open it with FIXED:WRAP:CHSET="M" and set $X to zero before the WRITE to avoid filling the last record with spaces (the default PAD byte value).

Note

With CHSET not "M", FIXED has a different definition. Each record is really the same number of bytes as specified by RECORDSIZE. Padding bytes are added as needed to each record.

See Also
-

- "FIXED" (page 397)

- "WRAP" (page 414)

- "X" (page 433)

Example:

```
bincpy(inname,outname); GT.M routine to do a binary copy from file named in argument 1 to file named in argument 2
        ;
  new adj,nrec,rsize,x
  new $etrap
  set $ecode="",$etrap="goto error",$zstatus=""
  set rsize=32767                          ; max recordsize that keeps $X on track
  open inname:(readonly:fixed:recordsize=rsize:exception="goto eof")
  open outname:(newversion:stream:nowrap:chset="M")
  for nrec=1:1 use inname read x use outname write x
eof
  if $zstatus["IOEOF" do  quit
  . set $ecode=""
  . close inname
  . use outname
  . set adj=$x
  . set $x=0 close outname
  . write !,"Copied ",$select((nrec-1)<adj:adj,1:((nrec-1)*rsize)+adj)," bytes from ",inname," to ",outname
  else  use $principal write !,"Error with file ",inname,":"
error
  write !,$zstatus
  close inname,outname
  quit
```

Sequential File Deviceparameter Summary

The following tables provide a brief summary of deviceparameters for sequential files grouped into related areas. For more detailed information, refer to "Open" (page 142), "Use" (page 151), and "Close" (page 440).

Error Processing Deviceparameters		
DEVICEPARAMETER	**COMMAND**	**COMMENT**
EXCEPTION=expr	O/U/C	Controls device-specific error handling.

File Pointer Positioning Deviceparameters		
DEVICEPARAMETER	**COMMAND**	**COMMENT**
APPEND	O	Positions file pointer at EOF.
REWIND	O/U/C	Positions file pointer at start of the file.
SEEK=strexpr	O/U	Positions the current file pointer to the location specified in strexpr. The format of strexpr is a string of the form "[+|-]integer" where

File Pointer Positioning Deviceparameters		
DEVICEPARAMETER	COMMAND	COMMENT
		unsigned value specifies an offset from the beginning of the file, and an explicitly signed value specifies an offset relative to the current file position. For STREAM or VARIABLE format, the positive intexpr after any sign is a byte offset, while for a FIXED format, it is a record offset. In order to deal with the possible presence of a Byte Order Marker (BOM), SEEK for a FIXED format file written in a UTF character set must follow at least one prior READ since the device was created.

File Format Deviceparameters		
DEVICEPARAMETERS	COMMAND	COMMENT
[NO]FIXED	O	Controls whether records have fixed length.
[Z]LENGTH=intexpr	U	Controls virtual page length.
RECORDSIZE=intexpr	O	Specifies maximum record size.
STREAM	O	Specifies the STREAM format.
VARIABLE	O	Controls whether records have variable length.
[Z]WIDTH=intexpr	U	Controls maximum width of an output line.
[Z][NO]WRAP	O/U	Controls handling of records longer than device width.

File Access Deviceparameters		
DEVICEPARAMETER	COMMAND	COMMENT
DELETE	C	Specifies file be deleted by CLOSE.
GROUP=expr	O/C	Specifies file permissions for other users in the owner's group.
NEWVERSION	O	Specifies GT.M create a new version of file.
OWNER=expr	O/C	Specifies file permissions for the owner of file.
[NO]READONLY	O	Controls read-only file access.
RENAME=expr	C	Specifies CLOSE rename a disk file with name specified by expression.
REPLACE=expr	C	Specifies CLOSE replace(overwrite an existing file if necessary) the name of a disk file with the name specified by the expression.
SYSTEM=expr	O/C	Specifies file permissions for the owner of the file (same as OWNER).
[NO]TRUNCATE	O/U	Controls overwriting of existing data in file.

File Access Deviceparameters		
DEVICEPARAMETER	COMMAND	COMMENT
UIC=expr	O/C	Specifies file's owner ID.
WORLD=expr	O/C	Specifies file permissions for users not in the owner's group.

O: Applies to the OPEN command

U: Applies to the USE command

C: Applies to the CLOSE command

Sequential File Examples

This section contains a few brief examples of GT.M sequential file handling.

Example:

```
GTM>do ^FREAD
FREAD;
 zprint ^FREAD
 read "File > ",sd
 set retry=0
 set $ztrap="BADAGAIN"
 open sd:(readonly:exception="do BADOPEN")
 use sd:exception="goto EOF"
 for  use sd read x use $principal write x,!
EOF;
 if '$zeof zmessage +$zstatus
 close sd
 quit
BADOPEN;
 set retry=retry+1
 if retry=2 open sd
 if retry=4 halt
 if $piece($zstatus,",",1)=2 do
 . write !,"The file ",sd," does not exist. Retrying in about 2 seconds ..."
 . hang 2.1
 . quit
 if $piece($zstatus,",",1)=13 do
 . write !,"The file ",sd," is not accessible. Retrying in about 3 seconds ..."
 . hang 3.1
 . quit
 quit
BADAGAIN;
 w !,"BADAGAIN",!

File >
```

This example asks for the name of the file and displays its contents. It OPENs that file as READONLY and specifies an EXCEPTION. The exception handler for the OPEN deals with file-not-found and file-access errors and retries the OPEN

command on error. The first USE sets the EXCEPTION to handle end-of-file. The FOR loop reads the file one record at a time and transfers each record to the principal device. The GOTO in the EXCEPTION terminates the FOR loop. At label EOF, if $ZEOF is false, the code reissues the error that triggered the exception. Otherwise, the CLOSE releases the file.

Example:

```
GTM>do ^formatACCT
formatACCT;
 zprint ^formatACCT;
 set sd="temp.dat",acct=""
 open sd:newversion
 use sd:width=132
 for  set acct=$order(^ACCT(acct)) quit:acct=""  do
 . set rec=$$FORMAT(acct)
 . write:$y>55 #,hdr write !,rec
 close sd
 quit
```

This OPENs a NEWVERSION of file temp.dat. The FOR loop cycles through the ^ACCT global formatting (not shown in this code fragment) lines and writing them to the file. The FOR loop uses the argumentless DO construct to break a long line of code into more manageable blocks. The program writes a header record (set up in initialization and not shown in this code fragment) every 55 lines, because that is the application page length, allowing for top and bottom margins.

FIFO Characteristics

FIFOs have most of the same characteristics as other sequential files, except that READs and WRITEs can occur in any order.

The following characteristics of FIFO behavior may be helpful in using them effectively.

With READ:

- If a READ is done while there is no data in the FIFO:

The process hangs until data is put into the FIFO by another process, or the READ times out, when a timeout is specified.

The following table shows the result and the values of I/O status variables for different types of READ operations on a FIFO device.

Operation	Result	$DEVICE	$ZA	$TEST	X	$ZEOF
READ X:n	Normal Termination	0	0	1	Data Read	0
READ X:n	Timeout with no data read	0	0	0	empty string	0
READ X:n	Timeout with partial data read	0	0	0	Partial data	0
READ X:n	End of File	1,Device detected EOF	9	1	empty string	1
READ X:0	Normal Termination	0	0	1	Data Read	0
READ X:0	No data available	0	0	0	empty string	0

Operation	Result	$DEVICE	$ZA	$TEST	X	$ZEOF
READ X:0	Timeout with partial data read	0	0	0	Partial data	0
READ X:0	End of File	1,Device detected EOF	9	1	empty string	1
READ X	Error	1,<error signature>	9	n/c	empty string	0

With WRITE:

- The FIFO device does non-blocking writes. If a process tries to WRITE to a full FIFO and the WRITE would block, the device implicitly tries to complete the operation up to a default of 10 times. If the gtm_non_blocked_write_retries environment variable is defined, this overrides the default number of retries. If the retries do not succeed (remain blocked), the WRITE sets $DEVICE to "1,Resource temporarily unavailable", $ZA to 9, and produces an error. If the GT.M process has defined an EXCEPTION, $ETRAP or $ZTRAP, the error trap may choose to retry the WRITE after some action or delay that might remove data from the FIFO device.

- While it is hung, the process will not respond to <CTRL-C>.

With CLOSE:

- The FIFO is not deleted unless the DELETE qualifier is specified.

- If a process closes the FIFO with the DELETE qualifier, the FIFO becomes unavailable to new users at that time.

- All processes currently USEing the FIFO may continue to use it, until the last process attached to it CLOSES it, and is destroyed.

- Any process OPENing a FIFO with the same name as a deleted FIFO creates a new one to which subsequent OPENs attach.

The default access permissions on a FIFO are the same as the mask settings of the process that created the FIFO. Use the SYSTEM, GROUP, WORLD, and UIC deviceparameters to specify FIFO access permissions. File permissions have no affect on a process that already has the FIFO open.

Considerations in Implementing FIFOs

As you establish FIFOs for interprocess communication, consider whether, and how, the following issues will be addressed:

- Do READs occur immediately, or can the process wait?

- Are timed READs useful to avoid system hangs and provide a way to remove the process?

- Does the WRITE process need to know whether the READ data was received?

- Will there be multiple processes READing and WRITEing into a single FIFO?

Error Handling for FIFOs

Deleting devices (or files) created by an OPEN which has an error has deeper implications when that device, especially a FIFO, serves as a means of communications between a two processes. If one process OPENs a FIFO device for WRITE, there is an interval during which another process can OPEN the same device for READ. During that interval the writer process can

encounter an error (for example, an invalid parameter) causing GT.M to delete the device, but the reader process can complete its OPEN successfully. This sequence results in a process with an orphaned device open for READ. Any other process that OPENs the same device for WRITE creates a new instance of it, so the reader can never find data to READ from the orphaned device. Since GT.M has insufficient context to enforce process synchronization between reader and writer, the application must use appropriate communication protocols and error handling techniques to provide synchronization between processes using files and FIFOs for communication.

GT.M Recognition of FIFOs

Like a sequential file, the path of a FIFO is specified as an argument expression to the OPEN, USE, and CLOSE commands. A device OPENed with a FIFO deviceparameter becomes a FIFO unless another device of that name is already OPEN. In that case, OPENing a device that has previously been OPENed by another process as a FIFO causes the process (the process here is the process trying to open the FIFO) to attach to the existing FIFO.

Note

If an existing named pipe (aka fifo special file) is OPENed even without specifying the FIFO deviceparameter, it is treated as if FIFO had been specified.

FIFO Device Examples

The following two examples represent a master/slave arrangement where the slave waits in a read state on the FIFO until the master sends it some data that it then processes.

Example:

```
set x="named.pipe"
open x:fifo
do getres
use x write res,!
```

This routine opens the FIFO, performs its own processing which includes starting the slave process (not shown in this code fragment).

Example:

```
set x="named.pipe"
open x:fifo
use x read res
do process(res)
```

This routine waits for information from the master process, then begins processing.

FIFO Deviceparameter Summary

The following table summarizes the deviceparameters that can be used with FIFOs.

File Format Deviceparameters		
DEVICEPARAMETER	**CMD**	**DESCRIPTION**
[NO]FIXED	O	Controls whether records have fixed length.

File Format Deviceparameters		
DEVICEPARAMETER	**CMD**	**DESCRIPTION**
[Z]LENGTH=intexpr	U	Controls the virtual page length.
RECORDSIZE=intexpr	O	Specifies the maximum record size.
VARIABLE	O	Controls whether records have variable length.
[Z]WIDTH=intexpr	U	Sets the device's logical record size and enables WRAP.
[Z][NO]WRAP	O/U	Controls the handling of records longer than the device width.

File Access Deviceparameters		
DEVICEPARAMETER	**CMD**	**COMMENT**
DELETE	C	Specifies that the FIFO should be deleted when the last user closes it. If specified on an OPEN, DELETE is activated only at the time of the close. No new attachements are allowed to a deleted FIFO and any new attempt to use a FIFO with the name of the deleted device creates a new device.
GROUP=expr	O/C	Specifies file permissions for other users in owner's group.
[NO]READONLY	O	OPENs a device for reading only (READONLY) or reading and writing (NOREADONLY).
OWNER=expr	O/C	Specifies file permissions for owner of file.
RENAME=expr	C	Specifies that CLOSE rename a disk file with the name specified by the expression.
REPLACE=expr	C	Specifies that CLOSE replace(overwrite an existing file if necessary) the name of a disk file with the name specified by the expression.
SYSTEM=expr	O/C	Specifies file permissions for owner of file (same as OWNER).
UIC=expr	O/C	Specifies the file's owner ID.
WORLD=expr	O/C	Specifies file permissions for users not in the owner's group.

Using Null Devices

Null devices comprise of a collection of system purpose devices that include /dev/null, /dev/zero, /dev/random, and /dev/urandom.

- /dev/null returns a null string on READ and sets $ZEOF

- /dev/random and /dev/urandom return a random value on READ and set $ZEOF

- /dev/zero returns 0's on READ and does not set $ZEOF

A null device discards all output. GT.M maintains a virtual cursor position for null devices as it does for terminals on output. Use null devices for program testing and debugging, or for jobs that permit I/O to be discarded under certain circumstances.

For example, JOB processes must have input and output devices associated with them, even though they do not use them. Null devices are low overhead never-fail alternatives for certain classes of I/O.

Null Deviceparameter Summary

The following table provides a brief summary of deviceparameters for null devices. For more detailed information, refer to "Open" (page 142), "Use" (page 151), and "Close" (page 440).

Null Deviceparameters		
DEVICEPARAMETER	**COMMAND**	**COMMENT**
EXCEPTION=expr	O/U/C	Controls device-specified error handling. For the null device this is only EOF handling and therefore exceptions can never be invoked except by a READ.
[NO]FILTER[=expr]	U	Controls some $X,$Y maintenance.
[Z]LENGTH=intexpr	U	Controls the length of the virtual page.
[Z]WIDTH=intexpr	U	Controls maximum size of a record.
[Z][NO]WRAP	O/U	Controls handling of records longer than the maximum width.
X=intexpr	U	Sets $X to intexpr.
Y=intexpr	U	Sets $Y to intexpr.
O: Applies to the OPEN command		
U: Applies to the USE command		
C: Applies to the CLOSE command		

Null Device Examples

This section contains examples of null device usage.

Example:

```
GTM>do ^runrep
runrep;
 zprint ^runrep
 set dev="/dev/null"
 set hdr="******** REPORT HEADER ************"
 open dev use dev
 set x="" write hdr,!,$zdate($horolog),?30,$job,!
 for  set x=$order(^tmp($job,x)) quit:x=""  do REPORT
 quit
REPORT;
 ;large amount of code
 quit;
```

This program produces a report derived from the information in the global variable ^tmp. The unspecified routine REPORT may potentially contain a large amount of code. To see that the basic program functions without error, the programmer may

discard the output involved in favor of watching the function. To run the program normally, the programmer simply has to change the variable dev to name another device and the routine REPORT writes to the dev device.

Example:

```
job ^X:(in="/dev/null":out="/dev/null":err="error.log")
JOB ^X:(IN="/dev/null":OUT="/dev/null":ERR="error.log")
```

This example issues a GT.M JOB command to execute the routine ^X in another process. This routine processes a large number of global variables and produces no output. In the example, the JOBbed process takes its input from a null device, and sends its output to a null device. If the JOBbed process encounters an error, it directs the error message to error.log.

Using PIPE Devices

A PIPE device is used to access and manipulate the input and/or output of a shell command as a GT.M I/O device. GT.M maintains I/O status variables for a PIPE device just as it does for other devices. An OPEN of the device starts a sub-process. Data written to the device by the M program is available to the process on its STDIN. The M program can read the STDOUT and STDERR of the sub-process. This facilitates output only applications, such as printing directly from a GT.M program to an lp command; input only applications, such as reading the output of a command such as ps; and co-processing applications, such as using iconv to convert data from one encoding to another.

A PIPE is akin to a FIFO device. Both FIFO and PIPE map GT.M devices to UNIX pipes, the conceptual difference being that whereas a FIFO device specifies a named pipe, but does not specify the process on the other end of the pipe, a PIPE device specifies a process to communicate with, but the pipes are unnamed. Specifically, an OPEN of a PIPE creates a subprocess with which the GT.M process communicates.

A PIPE device is specified with a "PIPE" value for mnemonicspace on an OPEN command.

> ### Note
>
> GT.M ignores the mnemonicspace specification on an OPEN of a previously OPEN device and leaves the existing device with its original characteristics.

Modes of PIPE Operation

The OPEN command for a PIPE provides a number of variations in the use of UNIX pipes shown below as Examples 1-4.

Example:

```
set p="Printer"
open p:(command="lpr":writeonly)::"PIPE"
```

This shows the use of a PIPE device to spool data to the default printer by spooling to the lpr command, opened via the default shell (the shell specified by the SHELL environment variable, and the shell used to start GT.M if SHELL is unspecified). The WRITEONLY device parameter specifies that the GT.M process not read data back from the lpr command. Use WRITEONLY when no errors are expected from the application(s) in the pipe. WRITEONLY tends not to serve most applications well.

Example:

```
set p="MyProcs"
open p:(command="ps -ef|grep $USER":readonly)::"PIPE"
```

This shows the use of a PIPE device to identify processes belonging to the current userid. The READONLY device parameter specifies that the GT.M process only read the output of the pipe, and not provide it with any input. This example illustrates the fact that the command can be any shell command, can include environment variables and pipes within the command.

> **Note**
>
> Flags to the ps command vary for different UNIX platforms.

Example:

```
set p="Convert"
open p:(shell="/bin/csh":command="iconv -f ISO_8859-1 -t WINDOWS-1252")::"PIPE"
```

This shows the use of a process to whose input the GT.M process writes to and whose output the GT.M process reads back in, in this example converting data from an ISO 8859-1 encoding to the Windows 1252 encoding. This example also shows the use of a different shell from the default. If the OPEN deviceparameters don't specify a SHELL, the PIPE device uses the shell specified by the environment variable SHELL; if it does not find a definition for SHELL, the device uses the system default /bin/sh.

Example:

```
set p="Files"
set e="Errors"
open p:(command="find /var/log -type d -print":readonly:stderr=e)::"PIPE"
```

GT.M uses the standard system utility find to obtain a list of subdirectories of /var/log, which are read back via the device with handle "Files" with any errors (for example, "Permission denied" messages for sub-directories that the find command cannot process) read back via the device with handle "Errors".

PIPE Characteristics

The following characteristics of PIPE may be helpful in using them effectively.

With Read:

A READ with no timeout reads whatever data is available to be read; if there is no data to be read, the process hangs until some data becomes available.

A READ with a timeout reads whatever data is available to be read, and returns; if there is no data to be read, the process waits for a maximum of the timeout period, an integer number of seconds, for data to become available (if the timeout is zero, it returns immediately, whether or not any data was read). If the READ returns before the timeout expires, it sets $TEST to TRUE(1); if the timeout expires, it sets $TEST to FALSE (0). When the READ command does not specify a timeout, it does not change $TEST. READ specifying a maximum length (for example, READ X#10 for ten characters) reads until either the PIPE has supplied the specified number of characters, or a terminating delimiter.

The following table shows the result and values of I/O status variables for various READ operations on a PIPE device.

Operation	Result	$DEVICE	$ZA	$TEST	X	$ZEOF
READ X:n	Normal Termination	0	0	1	Data Read	0
READ X:n	Timeout with no data read	0	0	0	empty string	0

Operation	Result	$DEVICE	$ZA	$TEST	X	$ZEOF
READ X:n	Timeout with partial data read	0	0	0	Partial data	0
READ X:n	End of File	1,Device detected EOF	9	1	empty string	1
READ X:0	Normal Termination	0	0	1	Data Read	0
READ X:0	No data available	0	0	0	empty string	0
READ X:0	Timeout with partial data read	0	0	0	Partial data	0
READ X:0	End of File	1,Device detected EOF	9	1	empty string	1
READ X	Error	1,<error signature>	9	n/c	empty string	0

With WRITE:

The PIPE device does non-blocking writes. If a process tries to WRITE to a full PIPE and the WRITE would block, the device implicitly tries to complete the operation up to a default of 10 times. If the gtm_non_blocked_write_retries environment variable is defined, this overrides the default number of retries. If the retries do not succeed (remain blocked), the WRITE sets $DEVICE to "1,Resource temporarily unavailable", $ZA to 9, and produces an error. If the GT.M process has defined an EXCEPTION, $ETRAP or $ZTRAP, the error trap may choose to retry the WRITE after some action or delay that might remove data from the PIPE device.

With WRITE /EOF:

WRITE /EOF to a PIPE device flushes, sets $X to zero (0) and terminates output to the created process, but does not CLOSE the PIPE device. After a WRITE /EOF, any additional WRITE to the device discards the content, but READs continue to work as before. A WRITE /EOF signals the receiving process to expect no further input, which may cause it to flush any output it has buffered and terminate. You should explicitly CLOSE the PIPE device after finishing all READs. If you do not want WRITE /EOF to flush any pending output including padding in FIXED mode or a terminating EOL in NOFIXED mode, SET $X=0 prior to the WRITE /EOF.

To avoid an indefinite hang doing a READ from a created process that buffers its output to the input of the PIPE device, READ with timeout (typically 0).

With CLOSE:

The CLOSE of a PIPE device prevents all subsequent access to the pipes associated with the device. Unless the OPEN that created the device specified INDEPENDENT, the process terminates. Note that any subsequent attempt by the created process to read from its stdin (which would be a closed pipe) returns an EOF and typical UNIX behavior would be to terminate on such an event.

PIPE Device Examples

The following examples show the use of deviceparameters and status variables with PIPE devices.

Example:

```
pipe1;
  set p1="test1"
  open p1:(shell="/bin/sh":comm="cat")::"PIPE"
  for i=1:1:10 do
  . use p1
  . write i,":abcdefghijklmnopqrstuvwxyz abcdefghijklmnopqrstuvwxyz ",!
  . read x
  . use $P
  . write x,!
  close p1
  quit
```

This WRITEs 10 lines of output to the cat command and reads the cat output back into the local variable x. The GT.M process WRITEs each line READ from the PIPE to the principal device. This example works because "cat" is not a buffering command. The example above would not work for a command such as tr that buffers its input.

Example :

```
pipe3;
  set p1="test1"
  open p1:(shell="/bin/sh":command="tr -d e")::"PIPE"
  for i=1:1:1000 do
  . use p1
  . write i,":abcdefghijklmnopqrstuvwxyz abcdefghijklmnopqrstuvwxyz ",!
  . read x:0
  . if '+$device use $principal write x,!
  use p1
  write /EOF
  for  read x quit:$zeof  use $principal write x,! use p1
  close p1
  quit
```

This shows the use of tr (a buffering command) in the created process for the PIPE device. To see the buffering effect the GT.M process WRITEs 1000 lines to the PIPE device. Different operating systems may have different buffer sizes. Notice the use of the r x:0 and the check on $DEVICE in the loop. If $DEVICE is 0, WRITE x writes the data read to the principal device. No actual READs complete, however, until tr reaches its buffer size and writes to its stdout. The final few lines remain buffered by tr after the process finishes the first loop. The GT.M process then issues a WRITE /EOF to the PIPE causing tr to flush its buffered lines. In the final for loop the GT.M process uses the simple form of READ x from the PIPE followed by a WRITE of each line to the principal device until $zeof becomes TRUE.

Example :

```
pipe4;
  set a="test"
  open a:(command="nestin":independent)::"PIPE"
  use a
  set key=$KEY
  write "Show nestin still running after CLOSE of a",!
  write "The parent process of 1 shows the parent shell has exited after CLOSE of a"
  read line1,line2
  use $principal
  write !,line1,!,line2,!,!
  set k="ps -ef | grep -v grep | grep -v sh | grep -w '"_key_"' | awk '{print $2}'"
  set b="getpid"
```

```
open b:(command=k:readonly)::"PIPE"
use b
read pid
close a
close b
set k2="ps -ef | grep -v grep | grep -v sh | grep -w '"_pid_"'"
set c="psout"
open c:(command=k2:writeonly)::"PIPE"
close c
quit
```

This demonstrates that the created sub process nestin keeps running as an INDEPENDENT process after the GT.M process CLOSEs the pipe. This GT.M process uses another PIPE device to return the process id of nestin and READ it into pid so that it may be killed by this or another process, should that be appropriate.

Note

"nestin.c" is a program which reads from standard input and writes to standard output until it see and EOF. It then loops for 300 1sec sleeps doing nothing. The purpose of using independent is as a server process which continues until it receives some other signal for termination.

Example:

```
GTM>kill ^a
GTM>zprint ^indepserver
indepserver;
  read x
  write "received = ",x,!
  set ^quit=0
  for  do  quit:^quit
  . if $data(^a) write "^a = ",^a,!
  . Hang 5
GTM>set a="test"
GTM>open a:(command="mumps -run ^indepserver>indout":independent)::"pipe"
GTM>use a
GTM>write "instructions",!
GTM>close a
GTM>zsystem "cat indout"
received = instructions
GTM>set ^a=1
GTM>zsystem "cat indout"
received = instructions
^a = 1
^a = 1
^a = 1
GTM>s ^quit=1
GTM>zsystem "cat indout"
received = instructions
^a = 1
^a = 1
^a = 1
^a = 1
GTM>
```

This is a simple example using a mumps process as a server.

Example:

```
pipe5;
  set p1="test1"
  set a=0
  open p1:(shell="/bin/sh":command="cat":exception="goto cont1")::"PIPE"
  set c=":abcdefghijklmnopqrstuvwxyz abcdefghijklmnopqrstuvwxyz"
  for i=1:1:10000  do
  . use p1
  . write i_c,!
  . use $principal write i,!
  use p1
  write /EOF
  for  read x quit:$zeof  use $principal write x,! use p1
  close p1
  quit
cont1
  if $zeof quit
  if a=0 set a=i/2
  set z=$za
  ; use $device to make sure ztrap is caused by blocked write to pipe
  set d=$device
  if "1,Resource temporarily unavailable"=d DO
  . use $p
  . write "pipe full, i= ",i," $ZA = ",z,!
  . set i=i-1
  . use p1
  . for j=1:1:a  read x use $principal write j,"-",x,! use p1
  quit
```

This demonstrates how to deal with write blocking of a PIPE device. The loop doing the WRITE does not READ from the PIPE. Eventually causing the output of cat to block on its output and stop reading input from the pipe. When the process takes the $ZTRAP to cont1 it tests $DEVICE to determine if the trap is caused by the full pipe. If so, it uses the for loop to read half the number of lines output by the main loop. It decrements i and returns to the original WRITE loop to retry the failed line and continue with the WRITEs to the pipe. Depending upon the configuration of the environment, it may trap several times before processing all lines.

Example:

```
  ; Example program that starts another program in a pipe and traps the errors. The called
; programs intentionally induce errors
pipexample
  set $etrap="do readfrompipe(.pipe,.piperr) use $p zwrite $zstatus zhalt 99"
  set pipe="pipe"
  set piperr="piperr"
  set writesize=1024
  set cmd=$piece($zcmdline," ") set:'$length(cmd) cmd="induceEPIPE"
  open pipe:(shell="/bin/bash":command="$gtm_dist/mumps -run "_cmd_"^pipexample":stderr=piperr)::"pipe"
  zshow "D":devicelist write "The active device is ",devicelist("D",2),!
  use pipe
  for i=1:1:1024 write $tr($justify(i,writesize)," ","X"),!
  close pipe
  quit
; Same as above, but without defining the PIPE's standard error
nostderr
```

```
  set $etrap="do readfrompipe(.pipe) use $p zshow ""*"" zhalt 99"
  set pipe="pipe"
  set writesize=1024
  set cmd=$piece($zcmdline," ",2) set:'$length(cmd) cmd="induceEAGAIN"
  open pipe:(shell="/bin/bash":command="$gtm_dist/mumps -run "_cmd_"^pipexample")::"pipe"
  zshow "D":devicelist write "The active device is ",devicelist("D",2),!
  write !,!
  use pipe
  for i=1:1:1024 write $tr($justify(i,writesize)," ","X"),!
  close pipe
  quit
; This routine intentionally delays reading from the pipe to induce an EAGAIN
induceEAGAIN
  set $etrap="use $p zwrite $zstatus zhalt 99"
  set hangtime=+$zcmdline set:'hangtime hangtime=5 set add=1
  for i=1:1:1024 read x(i) quit:$zeof  do
  . set delay(i)=1/(add+$random(hangtime))
  . hang delay(i)
  . set:i=30 add=10
  halt
; This routine intentionally induces an EPIPE by immediately sending a SIGTERM to itself causing
; a FORCEDHALT error which goes to STDERR. Subsequently, a random DIVZERO error occurs, but this
; error goes to STDOUT since it is not a fatal error
induceEPIPE
  set $etrap="use $p zwrite $zstatus zhalt 99"
  set divzero=150373210 ; DIVZERO goes to stdout
  write "My PID is ",$job,!
  zsystem:'$zcmdline "kill -15 "_$job  ; FORCEDHALT error goes to stderr
  for i=1:1 read x(i) quit:$zeof  zmessage:'$random(1000) divzero
  halt
; Read the contents of the pipe on failure. Messages from the programs inside the pipe aid
; in undestanding the underlying problem(s)
readfrompipe(pipe,piperr)
  new i
  new $etrap
  set $etrap="set x=$zjobexam() zhalt 88"
  use pipe
  for i=1:1  read pipe(i):0 quit:'$test!$zeof
  zkill pipe(i)
  do:$data(piperr)
  . use piperr
  . for i=1:1  read piperr(i):0 quit:'$test!$zeof
  . zkill piperr(i)
  close pipe
  use $p
  for i=1:1  quit:'$data(pipe(i))  write ?4,"stdout:",pipe(i),!
  for i=1:1  quit:'$data(piperr(i))  write ?4,"stderr:",piperr(i),!
  quit
; Example of trapping an error and retrying the operation as necessary. Error conditions
; used are EPIPE, aka "Broken pipe" or ENO32, and EAGAIN, aka ENO11.
retry
  set $etrap="use $p zshow ""*"" zhalt 99"
  set pipe="pipe"
  set piperr="piperr"
  set writesize=1024
```

```
 set cmd=$piece($zcmdline," ") set:'$length(cmd) cmd="induceEPIPE"
 for try=0:1  do  quit:$get(readcomplete,0)
 . new $etrap set $etrap="goto retryEPIPE"
 . open pipe:(shell="/bin/bash":command="$gtm_dist/mumps -run "_cmd_"^pipexample "_try:stderr=piperr)::"pipe"
 . zshow "D":devicelist write "Try ",try,$char(9),devicelist("D",2),!
 . use pipe
 . for i=1:1:1024 do
 . . new $etrap set $etrap="goto retryEAGAIN^pipexample"
 . . write $tr($justify(i,writesize)," ","X"),!
 . set readcomplete=1
 close pipe
 use $p
 write ?4,"Writes completed",!
 quit
retryEPIPE
 quit:$zstatus'["ENO32"
 use $p
 write "...Caught on try ",try,", write ",i,"... ",$zstatus,!
 set $ecode=""
 do readfrompipe(.pipe,.piperr)
 quit
retryEAGAIN
 quit:$zstatus'["ENO11"
 use $p
 write "...Failed to perform non-blocked writes... Retrying write # ",$increment(i,-1),!
 set $ecode=""
 hang 1+$random(5)
 use pipe
 quit
```

This example demonstrates how to handle PIPE device errors, whether with the device itself or from programs inside the PIPE device.

Example:

```
sh> mumps -run pipexample induceEAGAIN
The active device is pipe OPEN PIPE SHELL="/bin/bash" COMMAND="$gtm_dist/mumps -run induceEAGAIN^pipexample"
▶ STDERR="piperr"
$ZSTATUS="11,pipexample+9^pipexample,%SYSTEM-E-ENO11, Resource temporarily unavailable"

sh> mumps -run retry^pipexample induceEAGAIN
Try 0    pipe OPEN PIPE SHELL="/bin/bash" COMMAND="$gtm_dist/mumps -run induceEAGAIN^pipexample 0" STDERR="piperr"
...Failed to perform non-blocked writes... Retrying write # 54
...Failed to perform non-blocked writes... Retrying write # 63
...Failed to perform non-blocked writes... Retrying write # 69
...Failed to perform non-blocked writes... Retrying write # 78
    Writes completed
```

This example demonstrates handling WRITE errors, like ENO11 or EAGAIN, that do not terminate the PIPE device. The PIPE device does non-blocking writes. If a process tries to WRITE to a full PIPE and the WRITE would block, the device implicitly tries to complete the operation up to a default of 10 times. GT.M sleeps 100 micro seconds between each retry. When dealing

with programs that can take a while to process input, it's a good idea to either schedule a delay between WRITEs or come up with a mechanism to back off the WRITEs when the buffer fills up.

```
sh> mumps -run pipexample induceEPIPE
The active device is pipe OPEN PIPE SHELL="/bin/bash" COMMAND="$gtm_dist/mumps -run induceEPIPE^pipexample"
▶ STDERR="piperr"
    stdout:My PID is 12808
    stderr:%GTM-F-FORCEDHALT, Image HALTed by MUPIP STOP
$ZSTATUS="32,pipexample+9^pipexample,%SYSTEM-E-ENO32, Broken pipe"

sh> mumps -run retry^pipexample induceEPIPE
Try 0   pipe OPEN PIPE SHELL="/bin/bash" COMMAND="$gtm_dist/mumps -run induceEPIPE^pipexample 0" STDERR="piperr"
...Caught on try 0, write 49... 32,retry+13^pipexample,%SYSTEM-E-ENO32, Broken pipe
    stdout:My PID is 16252
    stderr:%GTM-F-FORCEDHALT, Image HALTed by MUPIP STOP
Try 1   pipe OPEN PIPE SHELL="/bin/bash" COMMAND="$gtm_dist/mumps -run induceEPIPE^pipexample 1" STDERR="piperr"
...Caught on try 1, write 697... 32,retry+13^pipexample,%SYSTEM-E-ENO32, Broken pipe
    stdout:My PID is 16403
    stdout:$ZSTATUS="150373210,induceEPIPE+5^pipexample,%GTM-E-DIVZERO, Attempt to divide by zero"
Try 2   pipe OPEN PIPE SHELL="/bin/bash" COMMAND="$gtm_dist/mumps -run induceEPIPE^pipexample 2" STDERR="piperr"
    Writes completed
```

This example demonstrates how to create a separate STDERR pipe device from which to read the STDERR output of the program(s) inside the pipe. Reading the STDERR is important when dealing with failures from Unix programs. It is possible to read the errors without creating a STDERR pipe device, however the error messages are commingled with the output of the programs inside the pipe which could make diagnosis of the underlying problem harder. Notice that GT.M writes fatal errors, GTM-F types, to STDERR, but all others go to STDOUT.

Additionally, this example demonstrates handling errors that terminate the PIPE device. In this example, the PIPE device is terminated when a program inside the pipe terminates before reading all of the driving MUMPS program's output causing an EPIPE or ENO32, a broken pipe. In such a situation the MUMPS program must capture the error that caused the termination and respond accordingly. The program may need to call out to other programs to determine the status of a service it is using or to alert the operator of an error with an external program or service. To operate successfully, the program must recreate the pipe and retry the operation.

PIPE Deviceparameter Summary

The following table summarizes the PIPE format deviceparameters.

DEVICE PARAMETER	CMD	DESCRIPTION
[NO]FIXED	O	Controls whether records have fixed length
RECORDSIZE=intexpr	O	Specifies the maximum record size.
VARIABLE	O	Controls whether records have variable length.
[Z]WIDTH=intexpr	U	Sets the device's logical record size and enables WRAP.

DEVICE PARAMETER	CMD	DESCRIPTION
[Z][NO]WRAP	O/U	Controls the handling of records longer than the device width.

The following table summarizes PIPE access deviceparamters.

COMMAND=string	O	Specifies the command string to execut in a created process for the PIPE device. GT.M uses the default searching mechanism of the UNIX shell for creating the process and initiating its command(s).
SHELL=string	O	Specifies the path to a shell to be used instead of the default shell
STDERR=string	O	Specifies a device handle for a return pipe to which the created process writes any standard error output. The GT.M process can USE, READ, and CLOSE it, but cannot WRITE to it. When the GT.M process CLOSEs the PIPE device, the PIPE device CLOSEs STDERR, if still OPEN.
WRITEONLY	O	Specifies that the GT.M process may only WRITE to the created process via the PIPE device.
READONLY	O	Specifies that the GT.M process may only READ from the created process via the PIPE device. Output from both the standard output and the standard error output of the created process is available unless STDERR is specified.
PARSE	O	Specifies that GT.M parse the COMMAND and issue an OPEN exception for any invalid command.
INDEPENDENT	O	Specifies that the created process continues to execute after the PIPE device is CLOSEd.

Using Socket Devices

SOCKET devices are used to access and manipulate sockets. A SOCKET device can have unlimited associated sockets. The default limit is 64. Set the environment variable gtm_max_sockets to the number of maximum associated sockets sockets that you wish to set for a GT.M process. $VIEW("MAX_SOCKETS")returns the current value of the maximum number of associated sockets.

At any time, only one socket from the collection can be the current socket. If there is no current socket, an attempt to READ from, or WRITE to the device, generates an error.

Sockets can be attached and detached from the collection of sockets associated with a device. Detached sockets belong to a pseudo-device called the "socketpool". SOCKET devices use "YGTMSOCKETPOOL" to identify the socket pool; an attempt

OPEN a device of that name produces a DEVNAMERESERVED error. A process can detach a socket from a device and later attach it to the same device or another device.

> ⚠️ **Caution**
>
> Currently, GT.M does not produce an error if a socket is attached to a device having a different CHSET.

> 📌 **Note**
>
> Exception handler (EXCEPTION) operates at the SOCKET device level and error trapping (IOERROR) operates the socket-level. So, one EXCEPTION operates on all sockets of a SOCKET device and IOEROR can be individually turned on or off for each socket.

Message Management

From an application perspective, the transport layers used by a socket device are stream-oriented, with no provisions for implicit application messages. Therefore, the following are two common protocols used to segment application messages.

1. One method is to use a, typically small, fixed length message containing the length of the next, variable length, message. In GT.M a simplistic writer might be:

```
Write $Justify($Length(x),4),x
```

   A corresponding simplistic reader might be:

```
read len#4,x#len
```

   The advantage of this approach is that the message content (the value of x in the code fragments above) can contain any character. The disadvantage is that detecting that the protocol has become desynchronized is a problem.

2. The other common method is to place a delimiter between each application message. The protocol breaks if a message ever includes a delimiter as part of its content.

The SOCKET device provides a facility for recognizing delimiters to simplify parsing messages.

Socket Read Operation

TCP/IP is a stream-based protocol that guarantees that bytes arrive in the order in which they were sent. However, it does not guarantee that they will be grouped in the same packets.
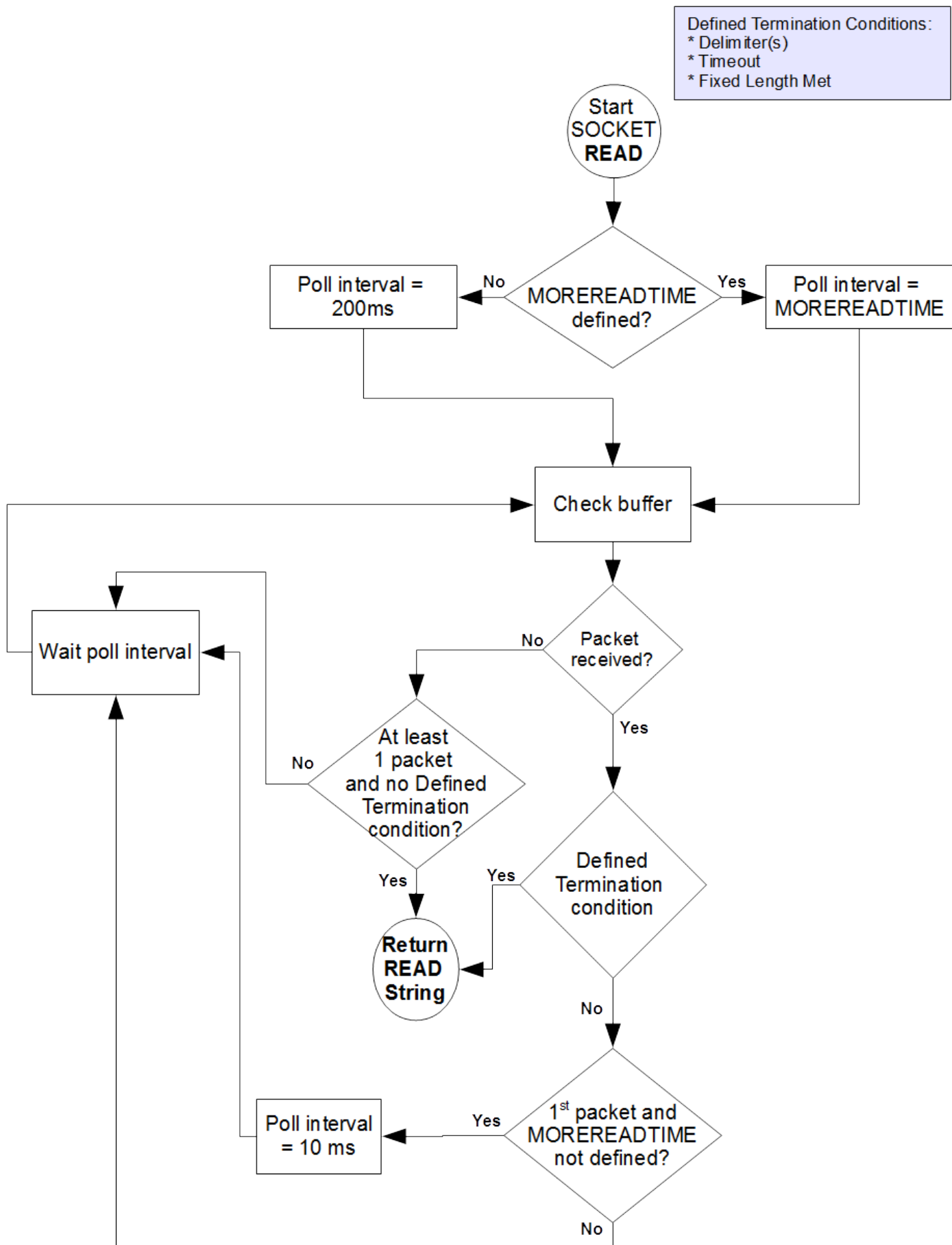
If packets arrive infrequently, or at varying rates that are sometimes slow, a short interval can waste CPU cycles checking for an unlikely event. On the other hand, if the handling of packets is time critical, a long interval can introduce an undesirable latency. If packets arrive in a rapid and constant flow (an unusual situation), the interval doesn't matter as much, as there is always something in the buffer for the READ to work with. If you do not specify MOREREADTIME, SOCKET READ implements a dynamic approach of using a longer first interval of 200 ms when it finds no data, then shortening the interval to 10 ms when data starts to arrive. If you specify an interval, the SOCKET device always uses the specified interval and does not adjust dynamically. For more information on MOREREADTIME, refer to "MOREREADTIME" [403].

Most SOCKET READ operations terminate as a result of the first condition detected from (a) receipt of delimiters, (b) receipt of the maximum number of characters, or (c) expiration of a timeout. Note that all of these conditions are optional, and a specific

READ may specify zero or more of them. This section refers to these three conditions as "defined terminating conditions". If a SOCKET READ is not subject to any of the defined terminating conditions, it terminates after it has received at least one character followed by an interval with no new characters. An error can also terminate a READ. While none of the terminating conditions is satisfied, the READ continues.

The following flowchart represents the logic of a SOCKET READ.

Defined Termination Conditions:
* Delimiter(s)
* Timeout
* Fixed Length Met

Start SOCKET **READ**

MOREREADTIME defined?

No → Poll interval = 200ms

Yes → Poll interval = MOREREADTIME

Check buffer

Packet received?

No → At least 1 packet and no Defined Termination condition?

Yes → Defined Termination condition

At least 1 packet and no Defined Termination condition?
No → Wait poll interval
Yes → **Return READ String**

Defined Termination condition
Yes → **Return READ String**
No → 1st packet and MOREREADTIME not defined?

1st packet and MOREREADTIME not defined?
Yes → Poll interval = 10 ms
No → Wait poll interval

Poll interval = 10 ms → Wait poll interval

Wait poll interval

Socket Read Termination Conditions

A SOCKET READ operation terminates if any of the following conditions are met:

Terminating Conditions	Argument Contains	$Device	$Key	$Test
Error	Empty String	Error String	Empty String	1
Timeout*	Data received before timeout	Empty String	Empty String	0
Delimiter*	Data up to, but not including the delimiter	Empty String	Delimiter String	1
Fixed Length Met*	String of Fixed Length	Empty String	Empty String	1
Width	Full width String	Empty String	Empty String	1
Buffer Emptied	One (1) to as many characters as provided by the transport interface before waiting for an interval (in milliseconds) specified by MOREREADTIME with no additional input. If MOREREADTIME is not specified, buffer is checked every 200 milliseconds for its first input and then every 10 milliseconds until no new input arrives and no other terminating conditions are met. IF MOREREADTIME is specified, READ uses that value exclusively for buffer checks.	Empty String	Empty String	1

* denotes Defined Terminating Conditions

A non-fixed-length read, with no timeout and no delimiters (the sixth row in the above table) requires a complex implementation of sequence of READs to ensure a predictable result. This is because the transport layer stream fragments delivered to the reader has only accidental correspondence with the operations performed by the writer. For example, the following:

Write "Message 1","Message 2" is presented to the reader as the stream "Message1Message2" but it can take from one (1) to 18 READ commands to retrieve the entire stream.

Messaging protocol should implement READ in any of the following ways:

1. Use a delimiter to separate messages (generic READ and possibly a larger value for MOREREADTIME).

2. Specify messages as <length, value> pairs (a pair of fixed-length READs (READ #) and possibly a larger value for MOREREADTIME).

3. Parse the bytes or characters as they come in (possibly a smaller value for MOREADTIME)

Message Delimiters

Each device can have from zero (0) to 64 delimiters associated with it. Each delimiter can be from one (1) to 64 characters. All the delimiters declared for a device are valid for any READ from any associated socket, which means, any of the defined delimiters terminate the READ. The actual terminating delimiter is available in $KEY. A WRITE to a socket associated with a device with one or more delimiters inserts the first of the delimiters for any WRITE ! format.

Read Command

The READ command may be used to obtain data from a socket. A READ operation terminates if any of the following are detected, in the order specified below:

Terminating Condition	Argument Contains	$Device	$Key (Continued)
Error	Empty string	Error string	Empty string
Timeout	Data received before timeout	Empty string	Empty string
Delimiter	Data up to, but not including the delimiter	Empty string	Delimiter string
Fixed length met	String of fixed length	Empty string	Empty string
Buffer emptied	One (1) to as many characters as happen to be provided by the transport interface	Empty string	Empty string

A non-fixed-length read, with no timeout and no delimiters requires a complex implementation of sequence of READs to ensure a predictable result. This is because the transport layer stream fragments delivered to the reader has only accidental correspondence with the operations performed by the writer. For example, the following

```
Write "Message 1","Message 2"
```

is presented to the reader as the stream "Message1Message2" but it can take from one (1) to 18 READ commands to retrieve the entire stream.

WRITE Command

The WRITE command sends data to a socket.

WRITE ! inserts the character(s) of the first I/O delimiter (if any) to the sending buffer. When the "ZFF=expr" characteristic is specified, WRITE # inserts the characters of expr. Otherwise WRITE # has no effect. WRITE ! and WRITE # always maintain $X and $Y in a fashion that emulates a terminal cursor position except when the device is OPENed with a UTF CHSET because the units for $X and $Y for terminals are in display columns while for sockets they are in codepoints/characters.

The WRITE command for SOCKET devices accepts the following controlmnemonics:

```
/L[ISTEN][(numexpr)]
```

where numexpr specifies the listen queue depth for a listening socket. The value of numexpr must be between 1 and the system-enforced maximum. By default, an OPEN or USE with LISTEN sets the listen queue size to 1. For vendor-specific information on how to change your system's maximum queue length, refer to the listen manpage.

```
WRITE /WAIT[(timeout[,[what][,handle]])]
```

where the required timeout is a numeric expression specifing how long in seconds a server waits for a connection or data to become available on one of the sockets in the current Socket Device, what may contain a string containing "READ" and/or "WRITE", and handle specifies the socket handle.

If the optional second argument only specifies "WRITE", WRITE /WAIT does not check incoming connections for listening sockets.

The optional third argument to WRITE /WAIT can be used to check only a single socket instead of all sockets in the current SOCKET device by specifying the handle name of a socket.

> ### Note
>
> In most circumstances, WRITE /WAIT(timeout[,"WRITE"]) for SOCKET devices which contain a non blocking socket returns immediately because non blocking sockets are usually ready for writing.
>
> If the current Socket Device is $PRINCIPAL and input and output are different SOCKETs, WRITE /WAIT applies to the input side of the device.

```
WRITE /PASS([targetpid],[timeout],handle[,handle]...)
```

WRITE /PASS allows a GT.M process to send DETACHed TCP or LOCAL sockets (that is, sockets in the socket pool) to another GT.M process. The receiving process must execute WRITE /ACCEPT to receive the socket.

- If a numeric targetpid is specified, GT.M matches the value against the process id ($JOB) of the process receiving the sockets. GT.M uses a system service to perform this check on platforms that support it - currently: Linux and AIX. If the pids do not match, GT.M issues a PEERPIDMISMATCH error and does not transfer the sockets.

- If a numeric timeout is specified, GT.M sets $TEST to 1 if the transfer completes within the specified time, and otherwise sets $TEST to 0 and does not transfer any of the sockets.

- Each handle specifies a socket in the socket pool.

- On a successful transfer, GT.M eliminates further access to the specified and sent sockets by the sending process. In any case where the transfer does not complete, GT.M retains all the sockets in the socket pool of the sender.

```
WRITE /ACCEPT(.lvar,[sourcepid],[timeout][,[handle]]...)
```

WRITE /ACCEPT allows a GT.M process to receive a DETACHed TCP or LOCAL sockets (that is, sockets in the socket pool) from another GT.M process . The sending process must execute WRITE /PASS to send the socket.

- lvar is an unsubscripted local variable name (lvn) which must be passed by reference indicated with a period (".") prefix. On successful completion, the specified unsubscripted lvn contains the handles of the received socket, in the order they were sent, delimited with a vertical bar ("|"). GT.M places the sockets in the socket pool, so the process can ATTACH them to an appropriate SOCKET device for subsequent use.

- If a numeric sourcepid is specified, GT.M matches the value against the process id ($JOB) of the process sending the sockets. If the pids do not match, GT.M issues a PEERPIDMISMATCH error and does not transfer the sockets.

- If a numeric timeout is specified, GT.M sets $TEST to 1 if the transfer completes within the specified time, and otherwise sets $TEST to 0 and does not transfer the sockets.

- If any handles are specified, GT.M assigns the provided handles to the received sockets in the order in which they appear in the WRITE /PASS of the sending process; empty items in the comma delimited handle list act to preserve ordering. Where the list provides no handle, the socket retains the handle provided by the sender. In either case, if there is already a socket with the transfer handle in the socket pool, GT.M generates a new handle for the transfer socket. GT.M ignores excess handles specified beyond the number of incoming sockets.

Both WRITE /PASS and WRITE /ACCEPT require the current $IO to be a SOCKET device with a CONNECTed (not LISTENing) and LOCAL domain (not TCP) current socket. GT.M issues CONNSOCKREQ or LOCALSOCKREQ errors, respectively, when those conditions are not met.

SOCKET devices do not support mixing other READs and WRITEs with socket passing on the same CONNECTED LOCAL socket and produce SOCKPASSDATAMIX errors. The application may perform multiple WRITE /PASS and WRITE /ACCEPT operations in either direction on the socket before issuing a CLOSE.

Note that the receiving process must establish desired deviceparameters (e.g., DELIMITER) for a socket either by ATTACHing it to a SOCKET device that provides the characteristic for all its sockets, or by a subsequent USE that specifies the appropriate deviceparameter(s). GT.M transfers only the socket connection itself, the socket handle, and buffered socket data (if any), but no characteristics established by the sender.

```
WRITE /TLS(option[,[timeout][,tlsid[,[obfuscatedpassword][,cfg-file-options]]]])
```

SOCKET devices support encrypted connections with TLS using an encryption plugin. GT.M ships with a reference implementation of the plugin which uses OpenSSL; the reference implementation also supports TLS for GT.M replication streams. OpenSSL options are controlled by a configuration file. The WRITE /TLS command activates this feature for connected sockets.

- option is "server", "client", or "renegotiate". "server" or "client" indicates which TLS role to assume. The server role requires a certificate specified in the configuration file section with the label matching tlsid. The client role may require a certificate depending on the OpenSSL options. If the argument specifies a timeout with options "client" or "server", GT.M sets $TEST to 1 if the command successfully completes or to 0 if it times out. $DEVICE provides status information in case of an error. ZSHOW "D" includes "TLS" in the second line of the output for an encrypted socket.

- "renegotiate" applies only to a server socket. Prior to TLSv1.3, this option allows applications to request a TLS renegotiation. Renegotiation requires the suspension of application communication and the application must read all pending data before initiating a renegotiation. This means that in the communication protocol used, both parties must be at a known state when renegotiating keys. For example, in GT.M replication, one party sends a renegotiation request and waits for an acknowledgement before initiating the renegotiation. Due to a change in the TLSv1.3 protocol which removed the concept of renegotiation, this option refreshes the TLS connection keys.

- tlsid refers to the name of a section in the configuration file specified by the gtmcrypt_config environment variable. If tlsid is not specified with the "renegotiate" option and cfg-file-options are specified, GT.M creates a virtual section by appending "-RENEGOTIATE" to the tlsid used to enable TLS on the socket. For the renegotiate option, if no section named tlsid is present in the configuration file, GT.M creates a virtual section with that name for the life of the process.

- obfuscatedpassword can represent a private key in the tlsid section of the configuration file which overrides any existing password such as the environment variable gtmtls_passwd_<tlsid>. If a password is supplied on the command, it must supply the tlsid as well unless the option is "renegotiate".

- cfg-file-options specifies configuration file options for the section labeled tlsid. If there is no section with the label tlsid, GT.M creates a new virtual section, which persists for the life the process. The specified options override those in the

configuration file except for ssl-options and verify-level GT.M combines/merges with those specified in the configuration file. The options, which are case-sensitive, must be specified in the same format as the configuration file including the terminating semi-colon (";").

- Supported cfg-file-options for the "renegotiate" command are (case-sensitive): verify-depth, verify-level, verify-mode, session-id-hex, and CAfile. WRITE /TLS ignores all other configuration file options whether given with the command or in the configuration file. For more information on the supported configuration options, refer to the Creating a Configuration File section of the Administration and Operations Guide.

Example:

```
set obspass="CD86FF2BFD1F06EE" ; maskpass output of password for private key
set cfgoptions="cert:""/path/to/certificate"";key:""/path/to/key.pem"";"
write /tls("server",,"tlsone",obspass,cfgoptions)
```

> **Note**
>
> Note that SOCKET device actions may produce the following errors: TLSDLLOPEN, TLSINIT, TLSCONVSOCK, TLSHANDSHAKE, TLSCONNINFO, TLSIOERROR, and TLSRENEGOTIATE.

The TLS plugin uses OpenSSL options in the configuration file specified under the tls: label as the default for all TLS connections and under the specific labels to override the defaults for corresponding connections.

GT.M buffers WRITEs to TLS enabled sockets until a subsequent USE :FLUSH, WRITE !, WRITE #, or an internal 400 millisecond timer expires.

> **Note**
>
> Because this functionality has a wide variety of user stories (use cases) and has substantial complexity, although the code appears robust, we are not confident that we have exercised a sufficient breadth of use cases in testing. Also we may make changes in future releases that are not entirely backwards compatible. We encourage you to use with this facility in development and testing, and to provide us with feedback. If you are an FIS customer and wish to use this in production, please contact us beforehand to discuss your use case(s).

> **Note**
>
> Owing to the range of OpenSSL versions in use across the breadth of platforms and versions supported by GT.M, on all platforms, but especially on non-Linux UNIX platforms, FIS recommends rebuilding the plugin for any production or production staging environments that use TLS from sources included with the GT.M binary distribution in order to use the specific version of OpenSSL installed on your systems. For more information on recompiling the reference implementation, refer to the Installing GT.M chapter of Administration and Operations Guide.

```
WRITE /BLOCK("OFF")
```

WRITE /BLOCK("OFF") enables non-blocking WRITEs for the current socket of the current SOCKET device. Sockets default to blocking WRITEs.

A socket must be enabled for non-blocking WRITEs before enabling it for TLS when using both features on the same socket.

For non-blocking sockets, GT.M retries a WRITE that blocks up to the number of times specified by the $gtm_non_blocked_write_retries environment variable with a 100 millisecond delay between each retry. The default retries value is 10 times if $gtm_non_blocked_write_retries is not defined.

If WRITE remains blocked after the specified retries, the WRITE sets $DEVICE to "1,Resource temporarily unavailable" and issues an error if IOERROR is "TRAP".

If IOERROR is not "TRAP", the application must check $DEVICE after each WRITE. An attempt to WRITE to a socket after it has been blocked is an error which sets $DEVICE to "1,Non blocking WRITE blocked - no further WRITEs allowed". Thus the only operation permitted on a blocked socket is a CLOSE.

> **Note**
>
> Note that multi-argument WRITEs are equivalent to a series of one argument WRITEs, and that GT.M turns unparenthesized concatenation within a write argument into multi-argument WRITEs. Format control characters such as "!" and "#" are each considered as an argument.
>
> Note that a significant delay between bytes for any reason, including blocking, especially within a multibyte character when CHSET is UTF-8, may be considered an error by the receiving end of a connection. If the application is unable to handle such a delay, it may result in an application error.

A WRITE to a non-blocking socket, which is not enabled for TLS, may terminate early on the following events:

<CTRL-C>, exceeding $ZMAXTPTIME, or $ZTIMEOUT expiring. These events result in a transfer to the interrupt vector or error handler at the next execution boundary as described in "Interrupt Handling" (page 88).

When non-blocking WRITEs are enabled for a socket, WRITE /WAIT may check if that socket would not block on WRITE in addition to READ. The optional second argument may contain a string containing "READ" and/or "WRITE".

If the second argument is omitted or specifies both "READ" and "WRITE" and the socket selected by WRITE /WAIT is ready for both READ and WRITE, $KEY contains:

```
READWRITE|<socket handle>|<address>.
```

If the second argument is omitted or contains "WRITE", WRITE /WAIT checks readiness for WRITE on non-blocking sockets, but never checks readiness to WRITE on blocking sockets, even if explicitly requested.

If the socket selected by a WRITE /WAIT implicitly or explicitly requests the state for writing would block on a READ but not block on WRITE, $KEY contains:

```
WRITE|<socket handle>|<address>
```

Note that a WRITE may still not be able to complete if it tries to write more bytes than the system is ready to accept.

If the socket selected by WRITE /WAIT which implicitly or explicitly requests the state for reading would not block on a READ but would block on a WRITE, $KEY contains:

```
READ|<socket handle>|<address>
```

$ZKEY after a prior WRITE /WAIT will contain a piece of the format "WRITE|sockethandle|ipaddress" if a non-blocking socket was considered writable, which we expect to be typical. If a socket was also readable, there will be two pieces in $ZKEY for the socket, one for WRITE and the other for READ.

An application can determine whether a socket is enabled for non-blocking WRITEs with:

```
$ZSOCKET(device,"BLOCKING",index)
```

which returns either 1 (TRUE) for blocking, or 0 (FALSE) for non-blocking.

Socket Device Operation

Each socket may be in one of the following states (observable through $KEY):

- CREATE-indicates that the socket exists.

- ESTABLISHED-After a successful OPEN or USE with the CONNECT device parameter or when GT.M was started with a socket as the $PRINCIPAL device.

- LISTENING-indicates that the OPEN or USE with the LISTEN deviceparameter was successful and a listen queue was established.

A listening socket used for accepting new connections goes through these three states in one step with a single OPEN or USE. When a server does a WRITE /WAIT, a client can establish a connection which creates a new server socket. $KEY includes information about this new socket in the form of CONNECT|handle|<address> where <address> is the IP address for TCP sockets and path for LOCAL sockets.

Each socket may have one or more sockets waiting for either an incoming connection or data available to READ (observable through $ZKEY). $ZKEY contains semi- colon (";") separated list of entries detailing any waiting sockets for a current SOCKET device.

For more information on $KEY and $ZKEY, refer to "Intrinsic Special Variables" [291].

Socket Deviceparameter Summary

The following table provides a brief summary of deviceparameters for socket devices.

Error Processing Deviceparameters		
DEVICEPARAMETER	COMMAND	COMMENT
EXCEPTION=expr	O/U/C	Controls device-specific error handling.
IOERROR=strexpr	O/U	use [NO]TRAP as strexpr If $LENGTH(strexpr)&("Tt"[$EXTRACT(strexpr)) then Error Trapping is enabled; otherwise the application must check $DEVICE for errors.

Socket Management Deviceparameters		
DEVICEPARAMETER	COMMAND	COMMENT
ATTACH=strexpr	O/U	With OPEN, ATTACH assigns expr as the handle name to the newly created socket With USE, expr specifies the handle of a socket in the socketpool.
CONNECT	O/U	Creates a client connection with a server, which is located by the information provided by expr.

Socket Management Deviceparameters		
DEVICEPARAMETER	COMMAND	COMMENT
[NO]DELIMITER[=strexpr]	O/U	Establishes or replaces the list of delimiters used by the newly created socket.
DETACH	U	Removes the socket identified by expr from the current socket device, without affecting any existing connection of that socket, and places it in the socketpool.
LISTEN=expr	O/U	Allocate a new socket to listen for a connection.
MOREREADTIME=intexpr	O/U	The polling interval (in milliseconds) that a SOCKET device uses to check for arriving packet
OPTIONS=expr	O/U	Specifies a list of options for the socket.
SOCKET=expr	U/C	With USE, makes the socket specified by the handle named in expr the current socket for the Socket device. With CLOSE, closes only the socket specified by the handle named in expr.
Z[NO]DELAY	U	Controls buffering of data packets by the system TCP stack using the TCP_NODELAY option to the setsockopt system call. ZNODELAY must be fully spelled out.

Format Deviceparameters		
DEVICEPARAMETER	COMMAND	COMMENT
[NO]FILTER[=strexpr]	U	strexpr specifies character filtering for socket output.
LENGTH=intexpr or ZLENGTH=intexpr	U	Sets virtual page length for socket device.
ICHSET=strexpr	O/U/C	strexpr specifies input character set
OCHSET=strexpr	O/U/C	strexpr specifies output character set
[NO]WRAP	O/U	Controls handling of records longer than the device width.
WIDTH=intexpr	U	Controls the maximum length of an output message.
Z[NO]FF[=strexpr]	O/U	Controls whether and what characters to send in response to a WRITE #.

Buffer Deviceparameters		
DEVICEPARAMETER	COMMAND	COMMENT
ZBFSIZE =intexpr	O/U	Allocates a buffer used by GT.M when reading from a socket.
ZIBFSIZE=intexpr	O/U	Sets the buffer size used by the network software (setsockopt SO_RCVBUF).

Socket Device Examples

The sockexamplemulti32.m routine shows the use of $KEY and $ZKEY in a basic socket I/O setup. It's functionality is atypical in order to demonstrate a number of features. It launches two jobs: a server process which opens a listening socket and a client process which makes five connections to the server. The server sends a message to each connection socket. Even-numbered client sockets read the message partially but do not send a response back to the server. Odd-numbered client sockets receive the full message and respond to the server with the message "Ok.". The server reads two characters (but the client sends three) and $ZKEY shows sockets with unread characters.Please click to download the sockexamplemulti32.m program and follow instructions in the comments near the top of the program file. You can also download sockexamplemulti32.m from http://tinco.pair.com/bhaskar/gtm/doc/books/pg/UNIX_manual/sockexamplemulti32.m.

You can start a GT.M process in response to a connection request made using inetd/xinetd. The following example uses inetd/xinetd to implement a listener which responds to connections and messages just as the prior example.

In the configuration file for xinetd, define a new service called gtmserver. Set socket_type to "stream" and wait to "no" as in the following snippet:

```
service gtmserver
{
disable = no
type = UNLISTED
port = 7777
socket_type = stream
wait = no
user = gtmuser
server = /path/to/startgtm
}
```

If you define the server in /etc/services, the type and port options are not needed. For more information, the xinetd.conf man page for more details.

If you are using inetd, add a line to /etc/inetd.conf with the sockettype "stream", protocol "tcp", and specify the "nowait" flag as in the example below, which assumes a gtmserver service is defined in /etc/services:

```
gtmserver stream tcp nowait gtmuser /path/to/startgtm
```

In both of the above examples, "gtmuser" is the name of the user to own and run the gtmserver service, and "/path/to/startgtm" is the name of a script which defines some environment variables needed before invoking GT.M. Please check the man page for inetd.conf on your system as the details may be slightly different.

The minimum variables are: $gtm_dist, which specifies the directory containing the GT.M distribution, and $gtmroutines, which specifies the paths used to locate the GT.M routines. As an example:

```
#!/bin/bash
cd /path/to/workarea
export gtm_dist=/usr/local/gtm
export gtmroutines="/var/myApp/o(/var/myApp/r) $gtm_dist"
export gtmgbldir=/var/myApp/g/mumps.dat
$gtm_dist/mumps -r start^server
```

When start^server begins, the $PRINCIPAL device is the current device which is the incoming connection and $KEY contains "ESTABLISHED|socket_handle| remote_ip_address". In most cases, a USE command near the beginning of the routine sets various device parameters such as delimiters.

The ZSHOW "D" command reports available information on both the local and remote sides of a TCP socket including local and remove addresses and ports.

```
0 OPEN SOCKET TOTAL=1 CURRENT=0
SOCKET[0]=h11135182870 DESC=0 CONNECTED ACTIVE NOTRAP
REMOTE=10.1.2.3@53731 LOCAL=10.2.3.4@7777
ZDELAY ZIBFSIZE=1024 ZIBFSIZE=0
```

I/O Commands

This section describes the following GT.M I/O commands:

- OPEN establishes a connection from a GT.M process to a device.

- USE declares a device as the current source of input and destination for output.

- READ accepts characters from the current device into a global or local variable.

- WRITE sends characters to the current device.

- CLOSE breaks the connection between a GT.M process and a device.

Open

The OPEN command establishes a connection from a GT.M process to a device.

The format of the OPEN command is:

```
O[PEN][:tvexpr] expr[:[(keyword[=expr][:...])][:numexpr][:expr]][,...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The required expression specifies the device to OPEN.

- The optional keywords specify deviceparameters that control device behavior; some deviceparameters take arguments delimited by an equal sign (=); if the argument only contains one deviceparameter, the surrounding parentheses are optional.

- The optional numeric expression specifies a time in seconds after which the command should timeout if unsuccessful; 0 provides a single attempt to open the device.

- When an OPEN command specifying a timeout contains no deviceparameters, double colons (::) separate the timeout numeric expression from the device expression.

- The optional expression specifies a mnemonicspace that selects a device binding. The only mnemonicspaces that GT.M currently accepts are SOCKET and PIPE.

- When an OPEN command specifies a mnemonicspace with no timeout, double colons separate the mnemonicspace string expression from the deviceparameters; if there are neither a timeout nor deviceparameters, triple colons separate the SOCKET mnemonicspace from the device expression.

- A triple colon for the PIPE menemonicspace produces an error.

- An indirection operator and an expression atom evaluating to a list of one or more OPEN arguments form a legal argument for an OPEN.

- For sequential files, multiple processes can open the same file for reading with the OPEN command.

By default, when a device is unavailable, GT.M retries the OPEN indefinitely at approximately one second intervals. A device is unavailable when another process is using it exclusively, or when the OPENing process does not have the resources left to open the device.

All other errors on OPEN raise an error condition and interrupt program flow. A timeout is a tool that lets a GT.M routine regain program control when a device remains unavailable. When the OPEN specifies a timeout, GT.M keeps retrying until either the OPEN succeeds or the timeout expires.

If OPEN establishes a connection with a device before the timeout expires, GT.M sets $TEST to TRUE (1). If the timeout expires, GT.M sets $TEST to FALSE (0). If an OPEN command does not specify a timeout, the execution of the command does not affect $TEST.

If a process has not previously OPENed a device, any deviceparameters not supplied on the OPEN take their default values. When reOPENing a device that it previously closed, a GT.M process restores all characteristics not specified on the OPEN to the values the device had when it was last CLOSEd, except with SD, FIFO, and PIPE. GT.M treats sequential disk files differently and uses defaults for unspecified sequential disk file characteristics on every OPEN; in other words, by default CLOSE of a sequential disk file acts as if DESTROY were specified and it requires an explicit NODESTROY to retain sequential disk file characteristics on a CLOSE.

For a sequential disk device CLOSEd with the NODESTROY deviceparameter, a subsequent OPEN of the device with no deviceparameters restores the device state including its file position; or to the file position specified by a SEEK deviceparameter. Note that when $ZCHSET specifies a UTF character set in FIXED format, the device must have done at least one READ prior to its close. An OPEN with additional deviceparameters positions the device to the beginning of the file or to the end of file if APPEND is specified. Any SEEK specified as a deviceparameter is then applied.

If you have a menu-driven application that OPENs and CLOSEs devices based on user selections, take care that every OPEN explicitly includes all deviceparameters important to the application.

If a process OPENs an already OPEN device, GT.M modifies any characteristics that accept changes when a device is OPEN to reflect any new deviceparameter specifications.

In UTF-8 mode, the OPEN command recognizes ICHSET, OCHSET, and CHSET as three additional deviceparameters to determine the encoding of the the input / output devices.

In M mode, the OPEN command ignores ICHSET, OCHSET, CHSET, and PAD device parameters.

If an I/O device uses a multi-byte character encoding, every READ and WRITE operation of that device checks for well-formed characters according to the specified character encoding with ICHSET or OCHSET. If the I/O commands encounter an illegal sequence of bytes, they always trigger a run-time error; a VIEW "NOBADCHAR" does not prevent such errors. Strings created by $ZCHAR() and other Z equivalent functions may contain illegal sequences. The only way to input or output such illegal sequences is to specify character set "M" with one of these deviceparameters.

Examples of OPEN

Example:

```
set sd="report.dat" open sd:newversion
```

This OPENs a NEWVERSION of a sequential disk file named report.dat for both read and write access.

OPEN Deviceparameters

APPEND

APPEND Applies to: SD

Positions the file pointer at the end-of-file. This deviceparameter only affects the device on the first OPEN command or OPEN command if the file is CLOSEd NODESTROY. Re-OPENing an already OPEN device with this deviceparameter has no effect. By default, OPEN sets the file pointer to the beginning-of-file.

> **Note**
>
> If an APPEND is combined with a SEEK deviceparameter the APPEND is done first - regardless of deviceparameter order.

Example:

```
set sd="foo.txt"
open sd:(append:recordsize=70:wrap)
use sd
```

This example open file foo.txt and positions the file pointer at the end of the file.

ATTACH

ATTACH=expr Applies to: SOC

ATTACH assigns expr as the handle name to the newly created socket. When ATTACH is used and one of LISTEN or CONNECT is specified on the same OPEN, the value of expr becomes the identifier of the newly created socket. If neither LISTEN nor CONNECT is specified, ATTACH is ignored.

For information on using the ATTACH with USE, refer to "ATTACH" (page 418) in the USE Deviceparameters section.

Example:

```
open tcpdev:(ichset="M":connect=hostname_":"_portno_":TCP":attach="client"):timeout:"SOCKET"
```

This example uses the ATTACH deviceparameter to specify "client" as the identifier of the newly created socket. Note that GT.M recognizes ICHSET only in UTF-8 mode.

CHSET

CHSET=expr Applies to: All devices

Establishes a common encoding for both input and output devices for the device being OPENed in UTF-8 mode. The value of the expression can be M, UTF-8, UTF-16, UTF-16LE, or UTF-16BE. For more information, refer to "ICHSET" (page 399) and "OCHSET" (page 404).

See Also	• "ICHSET" (page 399) • "OCHSET" (page 404) • "Discussion and Best Practices" (page 29)

COMMAND

COMMAND=expr Applies to: PIPE

Specifies the UNIX command the newly created shell process performs. An invalid command value triggers an error in the new process, not the process issuing the OPEN. This can make diagnosis difficult - see the "PARSE" (page 406) deviceparameter for potential assistance.

CONNECT

CONNECT=expr Applies to: SOC

Creates a client connection with a server, which is located by the information provided by expr. A new socket is allocated for the client connection and is made the current socket for the device, if the operation is successful.

expr specifies the protocol and the protocol-specific information. Currently, GT.M supports TCP/IP and LOCAL (also known as UNIX domain) socket protocols. For TCP/IP sockets, specify expr in the form of "<host>:<port>:TCP", where host is an IPv4 or IPv6 address optionally encapsulated by square-brackets ([]) like "127.0.0.1", "::1", "[127.0.0.1]", or "[::1]" or a IPv4 or IPv6 hostname like server.fis-gtm.com. When a hostname is specified, GT.M uses the IP version of the first address returned by DNS:

- that is supported by the operating system, and

- for which a network interface exists.

For LOCAL sockets, specify expr in the form of "<pathname>:LOCAL", where <pathname> is the name of the file to be used for communication. <pathname> may contain a dollar sign ($) followed by the name of an environment variable which GT.M expands in the same way as the device name for a sequential file. The maximum allowed length of the expanded path name depends on the OS.

For LOCAL sockets, CONNECT attempts to open the specified file. If it doesn't exist or there is no listener, CONNECT retries until it succeeds or a specified timeout expires.

> **Note**
>
> CONNECT is not compatible with LISTEN.

If the OPEN does not specify a timeout, a SOCKET OPEN waits for the connection to complete or an event that terminates the attempt.

Example:

```
open tcpdev:(connect=hostname_":"_portno_":TCP":attach="client":ioerror="TRAP"):timeout:"SOCKET"
```

This example establishes a client connect with the server using the connection string in the format of "hostname:port:TCP".

DELIMITER

[NO]DELIMITER=expr Applies to: SOC

DELIMITER establishes or replaces the list of delimiters used by the newly created socket. The default is NODELIMITER. The delimiter list on a preexisting device remains the same until it is explicitly replaced or deleted.

expr is a string where the following characters have special interpretation:

- ':' is used to separate delimiters (it is the delimiter for delimiters).

- '/' serves as an escape character, so use /: to use colon as part of a delimiter and // to use slash as part of a delimiter.

> **Note**
>
> expr "ab:/:://:bc" is interpreted as four delimiters, which are "ab", ":", "/", and "bc". One socket can have 0-64 delimiters and each delimiter can contain 1-64 characters.

Example:

```
open tcpdev:(connect=host_":"_portno_":TCP":delim=$c(13):attach="client"):timeout:"SOCKET"
```

This command specifies $CHAR(13)$ as the delimiter for the socket tcpdev.

EXCEPTION

EXCEPTION=expr Applies to: All devices

Defines an error handler for an I/O device. The expression must contain a fragment of GT.M code (for example, GOTO ERRFILE) that GT.M XECUTEs when GT.M detects an error, or an entryref to which GT.M transfers control, as appropriate for the current gtm_ztrap_form, setting except that there is never any implicit popping with EXCEPTION action.

A device EXCEPTION gets control after a non-fatal device error and $ETRAP/$ZTRAP get control after other non-fatal errors.

Example:

```
GTM>do ^FREAD
FREAD;
 zprint ^FREAD
 read "File > ",sd
 set retry=0
 set $ztrap="BADAGAIN"
 open sd:(readonly:exception="do BADOPEN")
 use sd:exception="goto EOF"
 for  use sd read x use $principal write x,!
EOF;
 if '$zeof zmessage +$zstatus
 close sd
 quit
BADOPEN;
 set retry=retry+1
 if retry=2 open sd
 if retry=4 halt
 if $piece($zstatus,",",1)=2 do
 . write !,"The file ",sd," does not exist. Retrying in about 2 seconds ..."
 . hang 2.1
 . quit
 if $piece($zstatus,",",1)=13 do
 . write !,"The file ",sd," is not accessible. Retrying in about 3 seconds ..."
 . hang 3.1
 . quit
 quit
BADAGAIN;
 w !,"BADAGAIN",!
```

```
File >
```

This example asks for the name of the file and displays its contents. It OPENs that file as READONLY and specifies an EXCEPTION. The exception handler for the OPEN deals with file-not-found and file-access errors and retries the OPEN command on error. The first USE sets the EXCEPTION to handle end-of-file. The FOR loop reads the file one record at a time and transfers each record to the principal device. The GOTO in the EXCEPTION terminates the FOR loop. At label EOF, if $ZEOF is false, the code reissues the error that triggered the exception. Otherwise, the CLOSE releases the file.

EMPTERM

[NO]EMPT[ERM] Applies to: TRM

Allows an "Erase" character on an empty input line to terminate a READ or READ # command. The default is NOEMPTERM. The gtm_principal_editing environment variable specifies the initial setting of [NO]EMPTERM. The TERMINFO specified by the current value of the TERM environment variable defines capnames values "kbs" and/or "kdch1" with character sequences for "Erase." If "kbs" or "kdch1" are multi-character values, you must also specify the ESCAPE or EDIT deviceparameters for EMPTERM recognition.

The erase character as set and shown by stty also terminates a READ command with an empty input line. You can set this erase character to various values using the stty shell command. Typical values of an erase character are <CTRL-H> and <CTRL-?>. Characters set and shown with stty setting must match what the terminal emulator sends.

The environment variable TERM must specify a terminfo entry that matches both what the terminal (or terminal emulator) sends and expects.

FFLF

[NO]FFLF Applies to: SD

The [NO]FFLF deviceparameter controls whether WRITE # produces only a form-feed (<FF>) or a form-feed and line-feed (<FF><LF>). Previously, GT.M used <FF><LF> which deviated from the standard, but out of concern for existing practice the default remains <FF><LF>.

Additionally, the "gtm_nofflf" environment variable controls the default WRITE # behavior of GT.M. If it is unset or set to 0, N[O] or F[ALSE], the default behavior is unchanged. If it is set to 1, Y[ES] or T[RUE], the default behavior of WRITE # is changed to produce only a form-feed (<FF>), though M programs can still control behavior by specifying the FFLF deviceparameter.

Example:

```
GTM>zprint ^fflf
fflf
  set file="/tmp/myfile"
  open file:(newversion:wrap:stream:nofflf)
  use file
  write #,"Hello"
  close file
  zsystem "od -tcd1 /tmp/myfile"
  quit
GTM>do ^fflf
0000000    \f    H    e    l    l    o    \n
           12   72  101  108  108  111   10
```

```
0000007
GTM>halt
```

This example opens the file "myfile", and sets the NOFFLF deviceparameter with OPEN. It will write the sequence "<FF>Hello<LF>" to the file.

FIFO

FIFO Applies to: FIFO

Specifies that the device for the OPEN is a FIFO name. GT.M creates the FIFO if it does not already exist and if the process has adequate privileges. However, in the event that the process does not have adequate privileges, the process generates a run-time error. A process does not require any special privileges to OPEN an existing FIFO. The FIFO needs to be readable (or writeable) just like any other file.

Example:

```
open file:(fifo:read:recordsize=1048576):100
```

FIXED

[NO]FIXED Applies to: SD FIFO PIPE

Selects a fixed-length record format for sequential disk files. FIXED does not specify the actual length of a record. Use RECORDSIZE to specify the record length.

NOFIXED specifies a variable-length record format for sequential disk files. NOFIXED is a synonym for VARIABLE. FIXED is incompatible with STREAM and VARIABLE. By default, records have VARIABLE length record format.

> **Note**
>
> FIXED length records do not implicitly use embedded record terminators such as line feeds.

In UTF-8 mode, GT.M I/O enforces a more record-oriented view of the file, treating each record as RECORDSIZE bytes long. Note that a UTF-8 code-point never splits across records. If a multi-byte character (when CHSET is UTF-8) or a surrogate pair (when CHSET is UTF-16) does not fit into the record (either logical as given by WIDTH or physical as given by RECORDSIZE), the WRITE command uses the byte values as specified by the PAD deviceparameter to fill the physical record. A combining character may end up in the subsequent record if it does not fit in the current record.

> **Note**
>
> PAD is effective only for devices opened with a CHSET related to Unicode® characters. In M mode PAD is always <SP>

Example:

```
GTM>do ^fixedex
fixedex;
  zprint ^fixedex
  set file="fix.txt"
  open file:(newversion:fixed:recordsize=4)
  use file
```

```
  write "Hello, World",!
  close file
  set file="fixnowrap.txt"
  open file:(newversion:fixed:recordsize=4:nowrap)
  use file
  write "Hel",!
  write "lo, World",! ; This writes only 'lo, '
  close file
  zsystem ("more fix*.txt")
  zsystem ("od -cb fix.txt")
  zsystem ("od -cb fixnowrap.txt")
  quit
::::::::::::::
fix.txt
::::::::::::::
Hello, World
::::::::::::::
fixnowrap.txt
::::::::::::::
Hel lo,
0000000   H   e   l   l   o   ,       W   o   r   l   d
        110 145 154 154 157 054 040 127 157 162 154 144
0000014
0000000   H   e   l       l   o   ,
        110 145 154 040 154 157 054 040
0000010
```

Example:

```
GTM>zprint ^gtmcp
gtmcp ; Copy a binary file using GT.M
  new dest,line,max,src
  if 2>$length($zcmdline," ") write "$gtm_dist/mumps -r source target",!
  set dest=$piece($zcmdline," ",2)
  set src=$piece($zcmdline," ",1)
  set max=1024*1024 ; the maximum GT.M string size
  open src:(readonly:FIXED:WRAP:CHSET="M") ;
  open dest:(newversion:FIXED:WRAP:CHSET="M") ; use FIXED format because it does not insert carriage control characters after $X
 reaches its maximum value.
  for  use src read line#max quit:$zeof  use dest write line
  close src
  use dest
  set $x=0
  close dest
  quit
```

This example copies a binary file using GT.M.

FOLLOW

[NO]FOLLOW Applies to: SD

Configures READ to return only when it has a complete record or reaches any specified timeout; it waits for more input rather than terminating on an EOF (end-of-file) condition.

The USE command can switch a device from NOFOLLOW to FOLLOW or from FOLLOW to NOFOLLOW. This provides a READ mode of operation similar to a **tail -f** in UNIX.

GROUP

GROUP=expr Applies to: SOC(LOCAL) SD FIFO

Specifies access permission on a UNIX file for other users in the file owner's group. The expression is a character string evaluating to null or to any combination of the letters RWX, indicating respectively Read, Write, and eXecute access. When permission controlling deviceparameters (OWNER,GROUP,WORLD) appears on an OPEN of a new file, any user category (OWNER, SYSTEM, WORLD), that is not explicitly specified is given the default access permissions. When any one of these deviceparameters appears on an OPEN of an existing device, any user category that is not explicitly specified remains unchanged.

In order to modify file security, the user who issues the OPEN must have ownership.

If none of GROUP, SYSTEM, OWNER, or WORLD are specified on OPEN, GT.M does not modify the permissions on an existing file and new files are created using the standard UNIX rules.

Example:

```
open "test52.txt":(append:group="rw")
```

This examples open file test52.txt in append mode with Read Write group access. Note that the user who opens file text52.txt must have ownership permissions for it.

ICHSET

ICHSET=expr

Applies to: All devices

Establishes the character encoding of an input device being OPENed and USEed in the UTF-8 mode. The value of the expression can be M, UTF-8, UTF-16, UTF-16LE, or UTF-16BE. In M mode, ICHSET has no effect.

If ICHSET is not specified, GT.M assumes UTF-8 as the default character set for input from the device.

If expr is set to a value other than M, UTF-8, UTF-16, UTF-16LE or UTF-16BE, GT.M produces a run-time error. UTF-16, UTF-LE, and UTF-16BE are not supported for $Principal and Terminal devices.

> **Note**
>
> ICHSET is a deviceparameter of both the OPEN and USE commands. As ICHSET can change the character set of an already OPENed device, it can help deal with binary data intermixed with character data.

See Also

- "CHSET" (page 393)
- "OCHSET" (page 404)
- "Discussion and Best Practices" (page 29)

INDEPENDENT

INDEPENDENT Applies to: PIPE

The INDEPENDENT deviceparameter specifies that the newly created process will not be terminated by the CLOSE of the device. The input and output of INDEPENDENT processes should be handled in such a way that it runs independently even after the CLOSE of the device. By default, CLOSE terminates the process associated with the PIPE device.

IKEY

Applies to: SD, PIPE, and FIFO

IKEY allows the use of a seperate key for READ to a device; for example, when a GT.M process is an element of a UNIX pipe. The format of the IKEY deviceparameter is:

IKEY="key_name [IV]"

key_name is case-sensitive and must match a key name in the "files" section of the gtmcrypt_config file. The optional IV specifies an initialization vector to use for encryption and decryption.

For more information, refer to the description of KEY deviceparameter of OPEN or USE.

IOERROR

IOERROR=expr Applies to: SOC

Enables exception handling in socket devices. expr specifies the I/O error trapping mode. A value equal to "TRAP" specifies that I/O errors on a device raise error conditions. A value equal to "NOTRAP", or when IOERROR is not specified, indicates that I/O error on a device does not raise error conditions.

> ## Note
>
> The IOERROR setting is associated with sockets while EXCEPTION is associated with the SOCKET device. In other words, IOERROR can be turned on or off for each of the sockets associated with a SOCKET device but there is only one EXCEPTION value which is used for all the sockets.

Example:

```
open sock:(connect=host_":"_port_":TCP":delim=$char(13,10):ioerror="TRAP")::"SOCKET"
```

This example opens a socket connection and specifies that I/O errors on the device raises error conditions.

If $LENGTH(expr)&("Tt"[$EXTRACT(expr)) then Error Trapping is enabled; otherwise the application must check $DEVICE and other ISVs for errors.

Note that an OPEN command does not change the current device. Therefore, $DEVICE does not have the status information when an error occurs on OPEN. An application should check $TEST and other ISVs for errors on OPEN.

$DEVICE holds status information for a socket device. To properly trap IOERRORs related to connection handling, it is best to create an empty SOCKET device (with something like **open tcpdev::timeout:"SOCKET"**) before opening a socket connection with the OPEN (with LISTEN or CONNECT deviceparameters) command. Then, use "USE tcpdev" to bring it to the current device. This method ensures that a device exists that would update $DEVICE with status information.

KEY

Applies to: SD, PIPE, and FIFO

Specifies information about the key file to use for reading and writing encrypted data. The syntax of the KEY deviceparameter is as follows:

KEY="key_name [IV]"

key_name is case-sensitive and must match a key name in the "files" section of the gtmcrypt_config file. The optional IV specifies an initialization vector to use for encryption and decryption.

To perform encryption and description, GT.M calls an encryption plugin using the GT.M encryption API and can use any library that conforms to the API. The encryption plugin in turn can call user-selected cryptographic libraries for cryptographic functionality. The key name and IV are passed as binary sequences of bytes to the reference implementation plugin. Because GT.M only uses the first space in the deviceparameter to delimit the end of the key, the IV can include any content, including spaces. The GT.M runtime system uses the plugin to pass the IV to the cryptographic libraries used, which use the length of the IV, to determine whether an IV less than the required size it is zero padded, and whether an IV that is longer than the required length generates an error. FIS suggests using $ZCHAR() in preference to $CHAR() when building IV byte sequences, and to make sure that IV sequences are not unintentionally subjected to numeric conversion.

A USE command with a KEY/IKEY/OKEY deviceparameter that attempts to change the cipher key or IV, including disabling encryption (by specifying an empty key), only succeeds prior to the first WRITE or READ, or after the encryption or decryption state has been reset, such as after a REWIND (only for READ) or a TRUNCATE at the start of a file (for both READ and WRITE).

Separate IKEY and OKEY deviceparameters allow different keys for READ from and WRITE to a device; for example, when a GT.M process is an element of a UNIX pipe. Because encryption ciphers use state machines (which are initialized with the IV at the beginning of the file), GT.M permits READ and WRITE operations only either starting at the beginning of a file, or at the position at which the last READ or WRITE operation completed. In particular, non-empty files cannot be opened in APPEND mode; the SEEK deviceparameter is prohibited; and the TRUNCATE is only permitted at the beginning of a file or at the end, the former deleting the contents, and the latter effectively a no-op.

> **Note**
>
> Encrypted files must be written and read sequentially from the beginning (including the Byte Order Marker for UTF files); GT.M supports READ and WRITE operations at arbitrary locations in a file only for unencrypted files.

Example:

The basic steps to use a key and IV to create an encrypted file and decrypt its data in a testing environment are as follows. These steps are solely for demonstration purposes. You must understand and appropriately adjust the steps before using them in a production environment. For example, in a production environment you should keep the key files in a secure location, protected with appropriate permissions from unauthorized access (such as 0500 for directories and 0400 for individual files). File encryption is just one of many components of a comprehensive security plan.

```
export LD_LIBRARY_PATH=/usr/local/lib
export GNUPGHOME=$PWD/mygnupg
$gtm_dist/plugin/gtmcrypt/gen_keypair.sh mykeypair@gtm Keymaster
$gtm_dist/plugin/gtmcrypt/gen_sym_key.sh 0 Sunday.key
$gtm_dist/plugin/gtmcrypt/gen_sym_key.sh 0 Monday.key
$gtm_dist/plugin/gtmcrypt/gen_sym_key.sh 0 Tuesday.key
$gtm_dist/plugin/gtmcrypt/gen_sym_key.sh 0 Wednesday.key
$gtm_dist/plugin/gtmcrypt/gen_sym_key.sh 0 Thursday.key
$gtm_dist/plugin/gtmcrypt/gen_sym_key.sh 0 Friday.key
$gtm_dist/plugin/gtmcrypt/gen_sym_key.sh 0 Saturday.key
echo -n "Enter password for gtm_passwd";export gtm_passwd="`$gtm_dist/plugin/gtmcrypt/maskpass|cut -f 3 -d " "`"
export gtmcrypt_config=mygtmcryptfile
cat mygtmcryptfile
 files: {
```

```
 CustomerReportKey1: "Sunday.key";
 CustomerReportKey2: "Monday.key";
 CustomerReportKey3: "Tuesday.key";
 CustomerReportKey4: "Wednesday.key";
 CustomerReportKey5: "Thursday.key";
 CustomerReportKey6: "Friday.key";
 CustomerReportKey7: "Saturday.key";
};
$gtm_dist/mumps -dir
GTM>zprint ^encrfile
encrfile
 set now=$horolog
 set timestamp=$zdate(now,"YYYYMMDDAM1260SS")
 set dayofweek=$zdate(now,"DAY","","1,2,3,4,5,6,7")
 set file="Customers"_timestamp_".log"
 open file:(newversion:key="CustomerReportKey"_dayofweek_" "_timestamp)
 use file
 write "Customer Report - Page 1",!
 close file
 write "IV : ",timestamp,!,"Key : CustomerReportKey"_dayofweek
GTM>do ^encrfile
IV : 20140911AM042419
Key : CustomerReportKey5
GTM>zprint ^readencrfile
readencrfile(key,iv)
 set file="Customers"_iv_".log"
 open file:(key=key_" "_iv)
 use file
 for read data use $principal write data,! use file quit:$zeof
 close file
GTM>do ^readencrfile("CustomerReportKey5","20140911AM042419")
Customer Report - Page 1
GTM>
```

In this example, the key name is CustomerReportKey followed by the number representing the day of the week, and IV is a timestamp, which is also a part of the file name. Although all reports start with the same string "Customer Report - Page 1", using a different IV for each file ensures that encrypted data begins with a different sequence of bytes, and making that IV a part of the file name ensures that the recipient of a report (who would have access to the key) can easily deduce the IV needed to decrypt the contents.

LISTEN

LISTEN=expr Applies to: SOC

A new socket is allocated to listen for a connection. It is made the current socket for the device, if the operation is successful. Upon successful completion, $KEY is set to the format of "LISTENING|<socket_handle>|{<portnumber>|</path/to/LOCAL_socket>}" otherwise, $KEY is assigned the empty string.

expr specifies the protocol and protocol specific information. Currently, GT.M supports TCP/IP and LOCAL (also known as UNIX domain) socket protocols. For TCP/IP sockets, specify expr in the form of "<port>:TCP".

If <port>=0 is specified, the system chooses the port for the TCP/IP socket.

For LOCAL sockets:

- Specify expr in the form of "<pathname>:LOCAL", where <pathname> is the name of the file to be used for communication. <pathname> may contain a dollar sign ($) followed by the name of an environment variable which GT.M expands in the same way as the device name for a sequential file. The maximum allowed length of the expanded path name depends on the OS.

- LISTEN creates the file if it doesn't exist. If the OPEN command specifies the NEWVERSION deviceparameter, the file specified by the pathname exists, and is a socket file, that file is deleted and GT.M creates a new file.

- LISTEN with an OPEN processes the GROUP, OWNER, SYSTEM, WORLD, UIC, and NEWVERSION deviceparameters the same as OPEN for sequential files.

MOREREADTIME

MOREREADTIME=intexpr Applies to: SOC

MOREREADTIME specifies the polling interval (in milliseconds) that a SOCKET device uses to check for arriving packets.

With no MOREREADTIME specified, SOCKET READ implements a dynamic approach of using a longer first interval of 200 ms when it finds no data, then shortening the interval to 10 ms when data starts to arrive.

If an interval is specified, the SOCKET device always uses the specified interval and doesn't adjust dynamically. This applies to any SOCKET READ. For more information on implementing SOCKET READ, refer to "Socket Read Operation" (page 379).

If a SOCKET READ is not subject to any of the defined terminating conditions, it terminates either after it has at least one character followed by an interval with no new packets, or reading 1,048,576 bytes.

If you use the MOREREADTIME behavior, bear in mind that:

- Usually, it is more efficient and responsive for an application to wait and process input in larger chunks. Therefore, a larger value for MOREREADTIME can bring larger chunks of input to the application. However, large values may make for sluggish response.

- A short value for MOREREADTIME may consume considerable CPU cycles, especially on a lightly loaded system.

- The maximum value of MORETREADTIME is 999 (basically 1 second). Never set MOREREADTIME to 0 as it causes excessive CPU "spinning".

Example:

```
Use tcpdev:morereadtime=200
```

This example specifies that all READs for socket device tcpdev must wait for 200 milliseconds for input.

NEWVERSION

NEWVERSION Applies to: SD FIFO SOC(LOCAL)

The NEWVERSION deviceparameter assures that when an existing file is used, it is empty upon the OPEN.

By default, if any version of the file exists, OPEN accesses the current version. If no version of the file exists, OPEN without READONLY creates a new file.

Example:

```
GTM>file1="foo.txt"
GTM>open file1:newversion:recordsize=5000
GTM>
```

This example creates a new version of sequential file foo.txtwith RECORDSIZE of 5000 bytes.

Example:

```
GTM>set delim=$c(13)
GTM>set tcpdev="server$"_$j,timeout=30
GTM>open
▶ tcpdev:(LISTEN="local.socket"_":LOCAL":delim=$c(13):attach="server":newversion):timeout:"SOCKET"
```

▼

This example deletes the old local.socket file (if it exists) and creates a new LISTENING local.socket file.

OCHSET

OCHSET=expr Applies to: All devices

Establishes the character encoding of the output device OPENed and USEed in the UTF-8 mode. The value of the expression can be M, UTF-8, UTF-16, UTF-16LE, or UTF-16BE. In M mode, OCHSET has no effect.

If *CHSET is not specified, GT.M assumes UTF-8 as the default character set for all the input / output devices.

If expr is set to a value other than M, UTF-8, UTF-16, UTF-16LE or UTF-16BE, GT.M produces a run-time error. UTF-16, UTF-LE, and UTF-16BE are not supported for $Principal and Terminal devices.

> **Note**
>
> OCHSET is a deviceparameter of both the OPEN and USE commands. As OCHSET can change the character set of an already OPENed device, it can help deal with binary data intermixed with character data.

See Also

- "CHSET" (page 393)
- "ICHSET" (page 399)
- "Discussion and Best Practices" (page 29)

Example:

```
GTM>SET file1="mydata.out"
GTM>SET expr="UTF-16LE"
GTM>OPEN file1:(ochset=expr)
GTM>SET DS=$CHAR($$FUNC^%HD("0905"))_$CHAR($$FUNC^%HD("091A"))
GTM>SET DS=DS_$CHAR($$FUNC^%HD("094D"))_$CHAR($$FUNC^%HD("091B"))_$CHAR($$FUNC^%HD("0940"))
GTM>USE file1 WRITE DS,!
GTM>CLOSE file1
```

This example opens a new file called mydata.out and writes Devanagari characters in the UTF-16LE encoding.

OKEY

Applies to: SD, PIPE, and FIFO

OKEY allows the use of a seperate key for WRITE to a device; for example, when a GT.M process is an element of a UNIX pipe. The format of the IKEY deviceparameter is:

OKEY="key_name [IV]"

key_name is case-sensitive and must match a key name in the "files" section of the gtmcrypt_config file. The optional IV specifies an initialization vector to use for encryption and decryption.

For more information, refer to the description of KEY deviceparameter of OPEN or USE.

OPTIONS

OPTIONS=expr Applies to: SOC

Specifies setsockopt() options to be set for sockets. The value of the expression is a comma separated list of option names. If the option takes a value, it is given after an equal sign (=) following the name.

The supported options are:

```
KEEPALIVE   a non zero value enables SO_KEEPALIVE.  A zero value disables SO_KEEPALIVE.
KEEPCNT     sets the TCP_KEEPCNT socket value.
KEEPIDLE    sets the TCP_KEEPIDLE socket value.
KEEPINTVL   sets the TCP_KEEPINTVL socket value.
SNDBUF      sets the size of the socket's network send buffer (SO_SNDBUF) in bytes.
```

Example:

OPEN dev:(LISTEN="1234:TCP":OPTIONS="KEEPALIVE=1,KEEPIDLE=50)::"SOCKET"

This enables SO_KEEPALIVE and sets TCP_KEEPIDLE to 50 seconds.

> **Note**
>
> Please review the man page for setsockopt for more information on the use of these options. On Linux, "man 7 socket" and "man 7 tcp" provide additional information.

OWNER

OWNER=expr Applies to: SOC(LOCAL) SD FIFO

Specifies access permission on a UNIX file for the owner of the file. The expression is a character string evaluating to null or to any combination of the letters RWX, indicating Read, Write, and eXecute access. When any one of these deviceparameters appears on an OPEN of a new file, any user category that is not explicitly specified is given the default mask. When any one of these deviceparameters (OWNER, GROUP, , WORLD) appears on an OPEN of an existing file, any user category that is not explicitly specified remains unchanged.

To modify file security, the user who issues the OPEN must have ownership.

If none of GROUP, SYSTEM, OWNER, or WORLD are specified on OPEN, GT.M does not modify the permissions on an existing file and new files are created using the standard UNIX rules.

Example:

```
open "test49.txt":(newversion:owner="rw":group="rw":world="rw")
```

This example opens a new version of test49.txt with Read Write acess for the owner.

PAD

PAD=expr Applies to: SD FIFO PIPE

For FIXED format sequential files when the character set is not M, if a multi-byte character (when CHSET is UTF-8) or a surrogate pair (when CHSET is UTF-16) does not fit into the record (either logical as given by WIDTH or physical as given by RECORDSIZE) the WRITE command uses bytes with the value specified by the PAD deviceparameter to fill out the physical record. READ ignores the pad bytes when found at the end of the record. The value for PAD is given as an integer in the range 0-127 (the ASCII characters). PAD is always a byte value and the default is $ZCHAR(32) or [SPACE].

In UTF-8 mode, there are three cases that cause GT.M to insert PAD characters when WRITEing. When READing GT.M attempts to strip any PAD characters. This stripping only works properly if the RECORDSIZE and PAD are the same for the READ as when the WRITEs occurred. WRITE inserts PAD characters when:

1. The file is closed and the last record is less than the RECORDSIZE. Records are padded (for FIXED) by WRITE ! as well as when the file is closed.

2. $X exceeds WIDTH before the RECORDSIZE is full.

3. The next character won't fit in the remaining RECORDSIZE.

> **Note**
>
> In all UTF-16 character sets, RECORDSIZE must be even and PAD bytes occupy two bytes with the high order byte zero.

Example:

```
GTM>do ^padexample
padexample
 zprint ^padexample
 set a="主要雨在西班牙停留在平原"
 set encoding="UTF-8"
 set filename="bom"_encoding_".txt"
 open filename:(newversion:fixed:record=8:pad=66:chset=encoding)
 use filename
 write a
 close filename
 halt
$ cat bomUTF-8.txt
主要BB雨在BB西班BB牙停BB留在BB平原
$ od -tcd1 bomUTF-8.txt
0000000  344  270  273  350  246  201    B    B  351  233  250  345  234  250    B    B
         -28  -72  -69  -24  -90 -127   66   66  -23 -101  -88  -27 -100  -88   66   66
0000020  350  245  277  347  217  255    B    B  347  211  231  345  201  234    B    B
         -24  -91  -65  -25 -113  -83   66   66  -25 -119 -103  -27 -127 -100   66   66
0000040  347  225  231  345  234  250    B    B  345  271  263  345  216  237
         -25 -107 -103  -27 -100  -88   66   66  -27  -71  -77  -27 -114  -97   32   32
```

In this example, the local variable a is set to a string of three-byte characters. PAD=66 sets padding byte value to $CHAR(66)

PARSE

PARSE Applies to: PIPE

The PARSE deviceparameter invokes preliminary validation of the COMMAND value. When debugging, PARSE provides more accessible diagnosis for COMMAND values. By default, OPEN does not validate command values before passing them to the newly created process. PARSE has certain limitations, which may, or may not map to, those of the shell.

- PARSE searches for the command in the environment variables PATH and gtm_dist and produces an error if it is not found.

- PARSE does not resolve aliases, so they produce an error.

- PARSE does not resolve environment variables, except $gtm_dist (as mentioned above), so they trigger an error.

- PARSE does not recognize built-in commands other than nohup and cd unless $PATH or $gtm_dist contain a version with the same name (as the built-in). In the case of nohup, PARSE looks for the next token in $PATH and $gtm_dist. "When PARSE encounters cd it ignores what follows until the next "|" token (if one appears later in the COMMAND value).

- PARSE rejects parentheses around commands.

- The following example fails:

    1. **OPEN p:(COMM="(cd; pwd)":WRITEONLY)::"PIPE"**

        which could be specified without a PARSE error as:

        OPEN p:(COMM="cd; pwd":WRITEONLY)::"pipe"

        This restriction does not include parentheses embedded in character strings as in:

    2. **OPEN p:(COMM="echo ""(test)"""::WRITEONLY)::"pipe"**

        or parameters to a command as in:

        OPEN p:(COMM="tr -d '()'":WRITEONLY)::"PIPE"

    3. The following are examples of valid OPEN commands using PARSE:

    ```
    OPEN a:(COMM="tr e j | echoback":STDERR=e:exception="g BADOPEN":PARSE)::"PIPE"
    OPEN a:(SHELL="/usr/local/bin/tcsh":COMM="/bin/cat |& nl":PARSE)::"PIPE"
    OPEN a:(COMM="mupip integ -file mumps.dat":PARSE)::"PIPE"
    OPEN a:(COMM="$gtm_dist/mupip integ -file mumps.dat":PARSE)::"PIPE"
    OPEN a:(COMM="nohup cat":PARSE)::"PIPE"
    ```

READONLY

[NO]READONLY Applies to: SD FIFO PIPE

OPENs a device for reading only (READONLY) or reading and writing (NOREADONLY).

To open a sequential file using the READONLY parameter, the file must exist on the disk. If it does not, GT.M issues a run-time error.

When GT.M encounters a WRITE directed to a file, OPENed READONLY, GT.M issues a run-time error.

By default, OPEN accesses the device or file NOREADONLY (read-write).

Example:

```
GTM>set filename="foo.txt"
GTM>open filename:(readonly:recordsize=1048576)
```

```
GTM>
```

This example open the file foo.txt with read permission

RECORDSIZE

RECORDSIZE=intexpr Applies to: SD FIFO PIPE

Overrides the default record size for a disk.

RECORDSIZE specifies an initial WIDTH. Note because RECORDSIZE is in bytes that in UTF-8 mode it produces a WIDTH that assumes one-byte characters.

The RECORDSIZE of a fixed length record for a GT.M sequential disk device is always specified in bytes, rather than characters.

For all UTF-16 CHSET values, RECORDSIZE must be even and PAD characters each occupy two bytes in the record.

The maximum size of intexpr is 1,048,576 bytes. GT.M produces an error if you specify a value greater than 1,048,576.

When a CHSET related to Unicode® characters is in use, GT.M treats RECORDSIZE as a byte limit at which to wrap or truncate output depending on [Z][NO]WRAP. For any Unicode character set, GT.M ignores RECORDSIZE for a device which is already open if any I/O has been done.

If the character set is not UTF-16, UTF-16LE, UTF-16BE, the default RECORDSIZE is 32K-1bytes.

If the character set is UTF-16, UTF-16LE or UTF16-BE, the RECORDSIZE must always be in multiples of 2. For these character sets, the default RECORDIZE is 32K-4 bytes.

For all UTF-16 CHSET values, RECORDSIZE must be even and PAD characters each occupy two bytes in the record.

REWIND

REWIND Applies to: SD

REWIND positions the file pointer of a sequential disk.

When $PRINCIPAL identifies a device that supports REWIND, the REWIND or INREWIND device parameters perform a REWIND of the input and OUTREWIND performs a REWIND of the output.

By default, OPEN does not REWIND.

Example:

```
OPEN "test40.txt":(REWIND:RECORDSIZE=70:NOWRAP)
```

This example opens file test40.txt and places the file pointer at the beginning of the file.

SEEK=strexpr

SEEK Applies to: SD

Positions the current file pointer to the location specified in strexpr. The format of strexpr is a string of the form "[+|-]integer" where unsigned value specifies an offset from the beginning of the file, and an explicitly signed value specifies an offset relative to the current file position. For STREAM or VARIABLE format, the positive intexpr after any sign is a byte offset, while for a FIXED format, it is a record offset. In order to deal with the possible presence of a Byte Order Marker (BOM), SEEK for a FIXED format file written in a UTF character set must follow at least one prior READ since the device was created.

> **Note**
>
> If an APPEND is combined with a SEEK deviceparameter the APPEND is done first - regardless of deviceparameter order.

Example:

```
GTM>zprint ^seekdemo
seekdemo
  new x,p
  set p="seekfixed"
  open p:(newversion:fixed:recordsize=60)
  use p
  ; create file with 9 records of length 60 bytes each
  ; number from 0 to correspond to record offset

  for i=0:1:8 write $justify(i_" - [-05-|-10-|-15-|-20-|-25-|-30-|-35-|-40-|-45-|-50-|-55-|",60)
  use p:rewind
  for i=0:1:8 read x set zk=$zkey use $p write "x= ",x," $zkey= ",zk,! use p
  close p
  write !!,"** OPEN with FIXED:RECORDSIZE=60:seek=""5""",!
  open p:(fixed:recordsize=60:seek="5")
  use p
  read x set ZKEY=$zkey
  ;expect: $ZKEY= 6,0
  use $p write "x= ",x," $zkey= ",ZKEY,!
  write !,"** use with SEEK=""-3""",!
  use p:seek="-3"
  read x set ZKEY=$zkey
  ;expect: $ZKEY= 4,0
  use $p write "x= ",x," $zkey= ",ZKEY,!
  write !,"** use with SEEK=""-1"" to read from the same record. read x#20 to read a partial record",!
  use p:seek="-1"
  read x#20 set ZKEY=$zkey
  ;expect: $ZKEY= 3,20
  use $p write "x= ",x," $zkey= ",ZKEY,!
  write !,"** read x#40 to finish reading the record",!
  use p
  read x#40 set ZKEY=$zkey
  ;expect: $ZKEY= 4,0
  use $p write "x= ",x," $zkey= ",ZKEY,!
  write !,"** CLOSE NODESTROY and reOPEN with no deviceparameters",!
  close p:nodestroy
  open p
  use p
  read x set ZKEY=$zkey
  ;expect: $ZKEY= 5,0
  use $p write "x= ",x," $zkey= ",ZKEY,!
  write !,"** CLOSE NODESTROY and reOPEN with SEEK=""+2""",!
  close p:nodestroy
  open p:seek="+2"
  use p
  read x set ZKEY=$zkey
  ;expect: $ZKEY= 8,0
  use $p write "x= ",x," $zkey= ",ZKEY,!
```

```
  write !,"** CLOSE NODESTROY and reOPEN with M:SEEK=""+3""",!
  close p:nodestroy
  open p:(M:seek="+3")
  use p
  read x set ZKEY=$zkey
  ;expect: $ZKEY= 4,0
  use $p write "x= ",x," $zkey= ",ZKEY,!
  write !,"** CLOSE NODESTROY and reOPEN with APPEND:SEEK=""-1""",!
  close p:nodestroy
  open p:(append:seek="-1")
  use p
  read x set ZKEY=$zkey
  ;expect: $ZKEY= 9,0
  use $p write "x= ",x," $zkey= ",ZKEY,!
  close p
  write !,"** CLOSE DESTROY and OPEN non-fixed with SEEK=""120"" and read 60 bytes",!
  open p:seek="120"
  use p
  read x#60 set ZKEY=$zkey
  ;expect: $ZKEY= 180
  use $p write "x= ",x," $zkey= ",ZKEY,!
  write !,"** CLOSE NODESTROY and reOPEN with append:SEEK=""-60"" and read last 60 bytes",!
  close p:nodestroy
  open p:(append:seek="-60")
  use p
  read x#60 set ZKEY=$zkey
  ;expect: $ZKEY= 540
  use $p write "x= ",x," $zkey= ",ZKEY,!
  close p
  quit


GTM>do ^seekdemo
x= 0 - [-05-|-10-|-15-|-20-|-25-|-30-|-35-|-40-|-45-|-50-|-55-| $zkey= 1,0
x= 1 - [-05-|-10-|-15-|-20-|-25-|-30-|-35-|-40-|-45-|-50-|-55-| $zkey= 2,0
x= 2 - [-05-|-10-|-15-|-20-|-25-|-30-|-35-|-40-|-45-|-50-|-55-| $zkey= 3,0
x= 3 - [-05-|-10-|-15-|-20-|-25-|-30-|-35-|-40-|-45-|-50-|-55-| $zkey= 4,0
x= 4 - [-05-|-10-|-15-|-20-|-25-|-30-|-35-|-40-|-45-|-50-|-55-| $zkey= 5,0
x= 5 - [-05-|-10-|-15-|-20-|-25-|-30-|-35-|-40-|-45-|-50-|-55-| $zkey= 6,0
x= 6 - [-05-|-10-|-15-|-20-|-25-|-30-|-35-|-40-|-45-|-50-|-55-| $zkey= 7,0
x= 7 - [-05-|-10-|-15-|-20-|-25-|-30-|-35-|-40-|-45-|-50-|-55-| $zkey= 8,0
x= 8 - [-05-|-10-|-15-|-20-|-25-|-30-|-35-|-40-|-45-|-50-|-55-| $zkey= 9,0
** OPEN with FIXED:RECORDSIZE=60:seek="5"
x= 5 - [-05-|-10-|-15-|-20-|-25-|-30-|-35-|-40-|-45-|-50-|-55-| $zkey= 6,0
** use with SEEK="-3"
x= 3 - [-05-|-10-|-15-|-20-|-25-|-30-|-35-|-40-|-45-|-50-|-55-| $zkey= 4,0
** use with SEEK="-1" to read from the same record. read x#20 to read a partial record
x= 3 - [-05-|-10-|-15-| $zkey= 3,20
** read x#40 to finish reading the record
x= -20-|-25-|-30-|-35-|-40-|-45-|-50-|-55-| $zkey= 4,0
** CLOSE NODESTROY and reOPEN with no deviceparameters
x= 4 - [-05-|-10-|-15-|-20-|-25-|-30-|-35-|-40-|-45-|-50-|-55-| $zkey= 5,0
** CLOSE NODESTROY and reOPEN with SEEK="+2"
x= 7 - [-05-|-10-|-15-|-20-|-25-|-30-|-35-|-40-|-45-|-50-|-55-| $zkey= 8,0
```

```
** CLOSE NODESTROY and reOPEN with M:SEEK="+3"
x= 3 - [-05-|-10-|-15-|-20-|-25-|-30-|-35-|-40-|-45-|-50-|-55-| $zkey= 4,0
** CLOSE NODESTROY and reOPEN with APPEND:SEEK="-1"
x= 8 - [-05-|-10-|-15-|-20-|-25-|-30-|-35-|-40-|-45-|-50-|-55-| $zkey= 9,0
** CLOSE DESTROY and OPEN non-fixed with SEEK="120" and read 60 bytes
x= 2 - [-05-|-10-|-15-|-20-|-25-|-30-|-35-|-40-|-45-|-50-|-55-| $zkey= 180
** CLOSE NODESTROY and reOPEN with append:SEEK="-60" and read last 60 bytes
x= 8 - [-05-|-10-|-15-|-20-|-25-|-30-|-35-|-40-|-45-|-50-|-55-| $zkey= 540

GTM>
```

This program demonstrates the use of the SEEK deviceparameter on OPEN and USE and reOPEN after CLOSE NODESTROY. This test is shown as an M program which may be executed, followed by the expected test output. First the test creates the file called "seekfixed" with 9, 60-byte records and then REWINDs and reads each record and outputs the record followed by $ZKEY which is a record,byte pair. Note that the records are numbered from 0 to match the SEEK record offset. Later in the test the same file is OPENed VARIABLE so $ZKEY will be a byte offset in that case. Details are given after the file output.

The first OPEN has deviceparameters set to (FIXED:RECORDSIZE=60:SEEK="5") which SEEKs to record offset 5 or physical record 6. Note, FIXED length records and RECORDSIZE remain in effect after a CLOSE NODESTROY unless changed on a reOPEN. Record offset 5 is read and output along with $ZKEY= 6,0 which points to the beginning of record offset 6. Next, a USE with SEEK="-3" is done to move back 3 records to read and output record followed by $ZKEY= 4,0. A USE with SEEK="-1" moves back one record to the beginning of the record just processed. A partial read of 20 bytes is done to show a record offset 3 with a byte offset of 20 or $ZKEY= 3,20. A read of 40 bytes is then done to finish processing that record for a $ZKEY= 4,0. Next a sequence of CLOSE NODESTROY and reOPENs are done. After the first CLOSE NODESTROY a reOPEN is done with no deviceparameters. The state of the file device, including file position, is restored and a read is done of record offset 4 which is output followed by $ZKEY= 5,0. The file device is then CLOSEd NODESTROY and a reOPEN is done with the only deviceparameter being SEEK="+2". The state of the file device is restored and a relative SEEK is done 2 records later in the file with a read which outputs record offset 7 followed by $ZKEY= 8,0. The file device is then CLOSEd NODESTORY and a reOPEN is done with deviceparameters (M:SEEK="+3"). The file device is OPENed at the beginning of the file due to the presence of a deviceparameter (M) other than SEEK on reOPEN. A relative SEEK forward of 3 records is then done from the beginning of the file and record offset 3 is read and output followed by $ZKEY= 4,0. The file device is then CLOSEd NODESTORY and a reOPEN is done with the (APPEND:SEEK="-1"). APPEND moves the file position to the EOF and then the SEEK="-1" moves the file position to the beginning of record 8 - the final record in the file. Note, the APPEND is applied prior to the SEEK - regardless of deviceparameter order. The file device is then CLOSEd (DESTROY is the default) and OPENed with the only deviceparameter being absolute SEEK="120" to byte offset 120. This processing is NOFIXED by default and a read of x#60 is done and output followed by $ZKEY= 180. The output is the same as record 2 in FIXED format. Finally, the file device is then CLOSEd NODESTROY and a reOPEN is done with deviceparameters (APPEND:SEEK="-60"). This will move the file position to the EOF and go back 60 bytes which is the starting offset to the final record in the file. Another read of x#60 and is done and output followed by $ZKEY= 540 - which is the size of the file.

SHELL

SHELL Applies to: PIPE

The SHELL deviceparameter specifies the shell for the new process. By default the newly created process uses the shell specified by the $SHELL environment variable, otherwise, if the environment variable SHELL is undefined the process uses /bin/sh.

STDERR

STDERR Applies to: PIPE

The STDERR deviceparameter specifies that the stderr output from the created process goes to a PIPE device with the name of the STDERR value. This PIPE device acts as a restricted device that can appear only as the argument to USE, READ and CLOSE commands. It is implicitly READONLY and an attempt to WRITE to it triggers an error. If it has not previously acted as the argument to an explicit CLOSE command, the CLOSE of the PIPE device implicitly closes the the STDERR device.

If the OPEN command does not specify STDERR, GT.M redirects the stderr output of the co-process created by the COMMAND to the standard output of the co-process. Specify STDERR when there is a need to read the standard error of the COMMAND seperately.

STREAM

[NO]STREAM Applies to: SD FIFO PIPE

STREAM and VARIABLE are semantically equivalent unless WRAP is disabled. As long as records do not exceed the WIDTH, they are also equivalent.

When WRAP is disabled and a WRITE exceeds the WIDTH, VARIABLE format truncates the line at the WIDTH, however in STREAM format, each WRITE argument is output without truncation or line terminator and the total record can be of arbitrary length.

For STREAM or VARIABLE record format files, a READ returns when it encounters an EOL, or has read #length characters for a READ #(fixed length READ), or WIDTH characters if #length is not specified, whichever occurs first.

By default, records are VARIABLE, NOSTREAM.

Example:

```
set sd="foo.txt"
open sd:(newversion:stream)
use sd:(width=20:nowrap)
for i=1:1:10 write " the quick brown fox jumped over the lazy dog ",$x,!
use sd:(rewind:width=100)
for i=1:1 use sd read x quit:$zeof use $principal write !,i,?5,x
close sd
quit
```

The output of this example is as follows:

```
1     the quick brown fox jumped over the lazy dog 46
2     the quick brown fox jumped over the lazy dog 46
3     the quick brown fox jumped over the lazy dog 46
4     the quick brown fox jumped over the lazy dog 46
5     the quick brown fox jumped over the lazy dog 46
6     the quick brown fox jumped over the lazy dog 46
7     the quick brown fox jumped over the lazy dog 46
8     the quick brown fox jumped over the lazy dog 46
9     the quick brown fox jumped over the lazy dog 46
10    the quick brown fox jumped over the lazy dog 46
```

If you change the FORMAT to VARIABLE, the same example produces the following output.

```
1     the quick brown fox
2     the quick brown fox
3     the quick brown fox
4     the quick brown fox
5     the quick brown fox
```

```
6      the quick brown fox
7      the quick brown fox
8      the quick brown fox
9      the quick brown fox
10     the quick brown fox
```

If you remove the "!" format from the WRITE sequence for VARIABLE, the same example produces the following output:

```
1      the quick brown fox
```

With STREAM, the same example produces the following output:

```
1      the quick brown fox jumped over the lazy dog 46 the quick brown fox jumped over the lazy dog 94 the
2      quick brown fox jumped over the lazy dog 142 the quick brown fox jumped over the lazy dog 191 the q
3    uick brown fox jumped over the lazy dog 240 the quick brown fox jumped over the lazy dog 289 the qui
4    ck brown fox jumped over the lazy dog 338 the quick brown fox jumped over the lazy dog 387 the quick
5      brown fox jumped over the lazy dog 436 the quick brown fox jumped over the lazy dog 485
```

SYSTEM

SYSTEM=expr Applies to: SOC(LOCAL) SD FIFO

This deviceparameter is a synonym for OWNER that is provided in the UNIX version of GT.M for compatibility with OpenVMS applications.

Example:

```
GTM> set perm="rwx"
GTM>OPEN "test52.txt":(NEWVERSION:SYSTEM="r":GROUP=perm:WORLD=perm)
GTM>ZSYSTEM "ls -la test52.txt"
```

```
-r--rwxrwx 1 user group 0 Aug 20 18:36 test52.txt
GTM>
```

This example opens file test52.txt and sets read access for the owner, while others have complete access.

TRUNCATE

[NO]TRUNCATE Applies to: SD

Truncates the file destroying all data beyond the current file pointer. If APPEND is also specified, the file pointer will be positioned at the end of the file even if TRUNCATE is before APPEND in the list of device parameters.

TRUNCATE on a USE $PRINCIPAL command works on a stdout device when the device supports the action.

UIC

UIC=expr Applies to: SOC(LOCAL) SD FIFO

Specifies the owner and group for the file.

Specifies the group that has access to the file. The format of the string is "o,g" where g is a decimal number representing the group portion of the UIC and o is a decimal number representing the owner portion. The super-user can set the file UIC to

any value. See the man page for the chown() system call for the rules for regular users since they vary by platform and system configuration.

VARIABLE

VARIABLE Applies to: SD FIFO PIPE

Specifies the VARIABLE record length format for sequential disk files.

By default, records have variable length format.

For more information, refer to "STREAM" (page 412).

WORLD

WORLD=expr Applies to: SOC(LOCAL) SD FIFO

Specifies access permissions for users other than the owner who are not in the group specified for a file. This category of users is usually referred to as other in UNIX. The expression is a character string evaluating to null or to any combination of the letters RWX, indicating respectively Read, Write, and eXecute access. When any one of these deviceparameters appear on an OPEN of an existing file, any user category that is not explicitly specified remains unchanged.

To modify file security, the user who issues the OPEN must have ownership.

By default, OPEN and CLOSE do not modify the permissions on an existing file. Unless otherwise specified, when OPEN creates a new file, it establishes security using standard defaulting rules.

Example:

```
OPEN "test51.txt":(NEWVERSION:WORLD="rw")
```

This example opens file test51.txt and specifies Read Write permission for users not in owner's group.

WRAP

[NO]WRAP Applies to: TRM SD NULL FIFO PIPE SOC

Enables or disables automatic record termination. When the current record size ($X) reaches the maximum WIDTH and the device has WRAP enabled, GT.M starts a new record, as if the routine had issued a WRITE ! Command. When reading, WRAP only determines whether $X remains within the range of zero to WIDTH.

Note that WRAP is enabled by default for SD, NULL, FIFO, PIPE and SOCKET. For TRM, WRAP is enabled by default if the terminfo variable auto_right_margin (capname "am") is set.

NOWRAP causes GT.M to require a WRITE ! to terminate the record. NOWRAP allows $X to become greater than the device WIDTH for terminals and null devices.

The combination of STREAM and NOWRAP on disk files allows you to write data of arbitrary length without truncation. Without the STREAM option, the WRAP option determines the action taken when the record length exceeds the device WIDTH. NOWRAP causes GT.M to truncate the record, while WRAP causes GT.M to insert a format control character except for FIXED format.

NOTE: FIFO, SD and SOCKET devices opened as $PRINCIPAL (at process start-up) default to NOWRAP and for SD and FIFO devices, STREAM.

WRITEONLY

[NO]WRITEONLY Applies to: PIPE

The WRITEONLY deviceparameter specifies that the PIPE acts only to send its output to the created process. Any attempt to READ from such a PIPE triggers an error. Note that when you open a PIPE with both STDERR and WRITEONLY you can still READ from the STDERR device.

ZBFSIZE

ZBFSIZE Applies to: SOC

Allocates a buffer used by GT.M when reading from a socket. The ZBFSIZE deviceparameter should be at least as big as the largest message expected.

By default, the size of ZBFSIZE is 1024 and the maximum it can be is 1048576.

ZDELAY

Z[NO]DELAY Applies to: SOC(TCP)

Controls buffering of data packets by the system TCP stack using the TCP_NODELAY option to the setsockopt system call. This behavior is sometimes known as the Nagle algorithm. The default is ZDELAY. This delays sending additional packets until either an acknowledgment of previous packets is received or an interval passes. If several packets are sent from one end of a connection before the other end responds, setting ZNODELAY may be desirable though at the cost of additional packets being transmitted over the network. ZNODELAY must be fully spelled out.

LOCAL sockets ignore the ZDELAY deviceparameter.

Example:

```
open tcpdev:(LISTEN=portno_":TCP":attach="server":zbfsize=2048:zibfsize=1024):timeout:"SOCKET"
```

This example opens the socket device tcpdev and allocates a buffer size of 2048 bytes.

ZFF

Z[NO]FF=expr Applied to: SOC

expr specifies a string of characters, typically in $CHAR() format to send to socket device, whenever a routine issues a WRITE #. When no string is specified or when ZFF="", then no characters are sent. The default in GT.M is ZNOFF.

ZIBFSIZE

ZIBFSIZE Applies to: SOC

Sets the buffer size used by the network software (setsockopt SO_RCVBUF).

The default and the maximum values depend on the platform and/or system parameters.

Note that LOCAL sockets ignore the ZIBFSIZE deviceparameter.

OPEN Deviceparameter Table

OPEN Deviceparameters						
OPEN DEVICEPARAMETER	TRM	SD	FIFO	PIPE	NULL	SOC
APPEND		X				
ATTACH=expr						X
CHSET=encoding	X	X	X	X	X	X
COMMAND=expr				X		
CONNECT=expr						X
[NO]DELIMITER						X
[NO]EMPT[ERM]	X					
EXCEPTION=expr	X	X	X	X	X	X
[NO]FFLF		X				
FIFO			X			
[NO]FIXED		X	X	X		
[NO]FOLLOW		X				
GROUP=expr		X	X			
ICHSET=encoding	X	X	X	X	X	X
IKEY		X	X	X		
INDEPENDENT				X		
IOERROR=expr						X
KEY		X	X	X		
LISTEN=expr						X
[NO]NEWVERSION		X	X			
OCHSET=encoding	X	X	X	X	X	X
OKEY		X	X	X		
TRM: Valid for terminals and printers						
SD: Valid for sequential disk files						
FIFO: Valid for FIFOs						
NULL: Valid for null devices						
PIPE: Valid for PIPEs						
SOC: Valid for Socket devices						

OPEN Deviceparameters						
OPEN DEVICEPARAMETER	**TRM**	**SD**	**FIFO**	**PIPE**	**NULL**	**SOC**
OWNER=expr		X	X			
PARSE				X		
[NO]READONLY		X	X	X		
RECORDSIZE=intexpr		X	X	X		
SEEK=strexpr		X				
SHELL=expr				X		
STDERR=expr				X		
[NO]STREAM		X	X	X		
SYSTEM=expr		X	X			
[NO]TRUNCATE		X	X			
UIC=expr		X	X			
VARIABLE		X	X	X		
WORLD=expr		X	X			
[NO]WRAP	X	X	X	X	X	X
[NO]WRITEONLY		X	X	X		
ZBFSIZE						X
Z[NO]DELAY						X
Z[NO]FF						X
ZIBFSIZE						X

TRM: Valid for terminals and printers

SD: Valid for sequential disk files

FIFO: Valid for FIFOs

NULL: Valid for null devices

PIPE: Valid for PIPEs

SOC: Valid for Socket devices

Use

The USE command selects the current device for READs (input) and WRITEs (output).

The format of the USE command is:

```
U[SE][:tvexpr] expr[:(keyword[=expr][:...])][,...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The required expression specifies the device to make the current device.

- A USE that selects a device not currently OPENed by the process causes a run-time error.

- The optional keywords specify deviceparameters that control device behavior; some deviceparameters take arguments delimited by an equal sign (=). If there is only one deviceparameter, the surrounding parentheses are optional.

- An indirection operator and an expression atom evaluating to a list of one or more USE arguments form a legal argument for a USE.

The intrinsic special variable $IO identifies the current device, so GT.M directs all READs and WRITEs to $IO. When a GT.M image starts, $PRINCIPAL is implicitly OPENed and USEd. Once the GT.M image USEs a device, $IO holds the name of that device until the next USE command.

A USE command modifies the device in accordance with the deviceparameters that apply to the device type and ignores those that do not apply. Characteristics set with USE deviceparameters persist until another USE for the same device with the corresponding deviceparameter. Characteristics persist through USEs of other devices and, except for SD, FIFO, and PIPE, through a subsequent CLOSE and re-OPEN.

Example:

```
USE $P:(X=0:Y=$Y-1:NOECHO)
```

This example USEs the principal device. If that device is a terminal, the deviceparameters turn off echo and position the cursor to the beginning of the previous line.

USE Deviceparameters

ATTACH

ATTACH=expr Applies to: SOC

expr specifies the handle for a socket in the socketpool. ATTACH looks up expr in the socketpool's collection of sockets and brings the one found to the current SOCKET device. If an ATTACH operation is successful, the attached socket becomes the current socket for the device.

ATTACH is not compatible with any other device parameters in the USE command.A socket can move from one device to another using DETACH/ATTACH.

> **Note**
>
> A socket does not carry [I|O]CHSET with it while being moved. Such a socket uses the [I|O]CHSET of the device it is ATTACHed to. If there is input still buffered, this may cause unintentional consequences in the application if [I|O]CHSET changes. GT.M does not detect (or report) a change in [I|O]CHSET due to DETACH/ATTACH.

For information on using the ATTACH with OPEN, refer to "ATTACH" (page 393) in the OPEN Deviceparameters section.

CANONICAL

[NO]CANONICAL Applies to: TRM

Enables or disables canonical input as controlled by the ICANON terminal attribute. See the documentation on your platform for details, but in general this would be erase and kill edit functions, and lines delimited by NL (usually <LF>), EOF (usually ^D), and EOL (usually not defined).

By default, canonical input is enabled (that is [NO]CANONICAL is the default).

CENABLE

[NO]CENABLE Applies to: TRM

Enables or disables the ability to force GT.M into Direct Mode by entering <CTRL-C> at $PRINCIPAL.

If CENABLE is set, <CTRL-C> interrupts process execution. For more information on interrupt handling, refer to "Interrupt Handling" (page 88).

By default, CENABLE is set. If CTRAP contains $C(3), CENABLE is disabled.

Example:

```
use $principal:(nocenable:ctrap="":exception="")
```

CLEARSCREEN

CLEARSCREEN Applies to: TRM

Clears the terminal screen from the present cursor position to the bottom of the screen. The CLEARSCREEN deviceparameter does not change the cursor position or the $X and $Y variables.

Example:

```
U $P:(X=0:Y=0:CLEAR)
```

This example positions the cursor to "home" in the upper left corner of a VDT and clears the entire current screen "page."

CONNECT

CONNECT=expr Applies to: SOC

Enables a client connection with a server, which is located by the information provided by expr. A new socket is allocated for the client connection and is made the current socket for the device, if the operation is successful.

expr specifies the protocol and the protocol-specific information. Currently, GT.M supports TCP/IP and LOCAL (also known as UNIX domain) socket protocols.

For more information, refer to "CONNECT" (page 394).

> **Note**
>
> CONNECT is not compatible with LISTEN.

Although CONNECT can be used with USE command, FIS recommends not to use it that way, because unlike the OPEN command, there is no way to specify a timeout to the USE command. CONNECT in the USE command take a default timeout value of 0.

Example:

Refer to the "CONNECT" examples in "Examples of OPEN" (page 392).

CONVERT

[NO]CONVERT Applies to: TRM

Enables or disables GT.M from converting lowercase input to uppercase during READs.

By default, the terminal device driver operates NOCONVERT.

Example:

```
use $principal:(convert)
READ X
```

This example converts all lowercase to uppercase during READ X.

CTRAP

CTRAP=$CHAR(intexpr[,...]) Applies to: TRM

Establishes a trap facility for <CTRL> characters for the current device. The expression is a comma seperated list of ASCII characters where 0<=intexpr<=31. Other than <CTRL_C>, GT.M recognizes <CTRL> characters only when reading them from $IO. The behavior for <CTRL-C> is different in the sense that the OS recognizes it as an out-of-band interrupt including when it occurs on $PRINCIPAL when $IO'=$PRINCIPAL and delivers it immediately; When the device receives a <CTRL-n>, GT.M interrupts process execution.

Other than the <CTRL> characters (ASCII 0 through 31), terminal configuration may cause some <CTRL> characters to be seen as causing an OS action and also prevent them from ever reaching GT.M's CTRAP facility.

For example, the command USE $PRINCIPAL:CTRAP=$C(26,30,7,19) sets a trap for the ASCII characters <SUB>, <RS>, <BEL> and <DC3>.

Specifying CTRAP completely replaces the previous CTRAP list. Setting CTRAP to the empty string ("") disables character trapping.

A <CTRL-n> enabled by CTRAP produces one of the following actions:

- If an EXCEPTION deviceparameter has been issued for the device, the process executes the EXCEPTION argument.

- Otherwise, if $ETRAP is not the empty string, execute $ETRAP.

- Otherwise, if $ZTRAP is not the empty string, the process executes $ZTRAP.

- Otherwise, GT.M terminates the process.

For more information on error handling, refer to Chapter 13: "Error Processing" (page 562).

When CTRAP includes <CTRL-C>, [NO]CENABLE has no effect. CTRAPping <CTRL-C> also takes precedence over CENABLE.

DELIMITER

[NO]DELIMITER Applies to: SOC

DELIMITER establishes or replaces the list of delimiters used by the current socket. The default is NODELIMITER.

expr must be a string of the following format:

1. ':' is used to separate delimiters (it is the delimiter for delimiters).

2. '/' serves as an escape character.

> **Note**
>
> expr "ab:/::://:bc" is interpreted as four delimiters, which are "ab", ":", "/", and "bc". One socket can have 0-64 delimiters and each delimiter can contain 1-64 characters.

Example:

See "Socket (server.m)" example.

DETACH

DETACH=expr Applies to: SOC

Removes the socket identified by expr from the current socket device, without affecting any existing connection of that socket. The removed socket is placed in the socketpool (which has the reserved name "YGTMSOCKETSPOOL") and may be attached to another socket device. If the socket being removed is the current socket, then GT.M does the following:

- The socket ATTACHed prior to the removed socket, is made current, if one such exists.

- The socket ATTACHed after the removed socket, is made current, if the removed one was the first socket.

- $PRINCIPAL is made the current device ($IO), if the removed socket was the only one in the current socket device.

> **Note**
>
> A socket can move from one device to another using DETACH/ATTACH. A socket does not carry [I| O]CHSET with it while being moved. Such a socket uses the [I|O]CHSET of the device it is ATTACHed to. If there is input still buffered, this may cause unintentional consequences in the application if [I|O]CHSET changes. GT.M does not detect (or report) a change in [I|O]CHSET due to DETACH/ATTACH.
>
> DETACH is not compatible with any other device parameters in the USE command.

Example:

```
GTM>set tcp="seerv" open tcp:(listen="6321:TCP":attach="serv")::"SOCKET"
```

```
GTM>zshow "D"
/dev/pts/9 OPEN TERMINAL NOPAST NOESCA NOREADS TYPE WIDTH=80 LENG=24
seerv OPEN SOCKET TOTAL=1 CURRENT=0
    SOCKET[0]=serv DESC=3 LISTENING PASSIVE NOTRAP PORT=6321
        ZDELAY ZBFSIZE=1024 ZIBFSIZE=87380 NODELIMITER
GTM>set tcp="seerv" o tcp:(listen="6322:TCP":attach="serv2")::"SOCKET"
GTM>zshow "D"
/dev/pts/9 OPEN TERMINAL NOPAST NOESCA NOREADS TYPE WIDTH=80 LENG=24
seerv OPEN SOCKET TOTAL=2 CURRENT=1
    SOCKET[0]=serv DESC=3 LISTENING PASSIVE NOTRAP PORT=6321
        ZDELAY ZBFSIZE=1024 ZIBFSIZE=87380 NODELIMITER
    SOCKET[1]=serv2 DESC=4 LISTENING PASSIVE NOTRAP PORT=6322
        ZDELAY ZBFSIZE=1024 ZIBFSIZE=87380 NODELIMITER
```

At this point, the socket device "seerv" has two sockets associated with it.

The following command moves the "serv" socket to the "YGTMSOCKETPOOL" device.

```
GTM>use tcp:detach="serv"
GTM>use 0 zshow "D"
/dev/pts/9 OPEN TERMINAL NOPAST NOESCA NOREADS TYPE WIDTH=80 LENG=24
seerv OPEN SOCKET TOTAL=1 CURRENT=0
    SOCKET[0]=serv2 DESC=4 LISTENING PASSIVE NOTRAP PORT=6322
        ZDELAY ZBFSIZE=1024 ZIBFSIZE=87380 NODELIMITER
YGTMSOCKETPOOL OPEN SOCKET TOTAL=1 CURRENT=0
    SOCKET[0]=serv DESC=3 LISTENING PASSIVE NOTRAP PORT=6321
        ZDELAY ZBFSIZE=1024 ZIBFSIZE=87380 NODELIMITER
```

Notice how socket "serv" is now associated with the pseudo socket device "YGTMSOCKETPOOL". Its only purpose is to hold detached sockets.

```
GTM>set tcp2="s2" o tcp2:::"SOCKET"
```

This creates a new socket device.

```
GTM>zshow "D"
/dev/pts/9 OPEN TERMINAL NOPAST NOESCA NOREADS TYPE WIDTH=80 LENG=24
 s2 OPEN SOCKET TOTAL=0 CURRENT=0
seerv OPEN SOCKET TOTAL=1 CURRENT=0
    SOCKET[0]=serv2 DESC=4 LISTENING PASSIVE NOTRAP PORT=6322
        ZDELAY ZBFSIZE=1024 ZIBFSIZE=87380 NODELIMITER
YGTMSOCKETPOOL OPEN SOCKET TOTAL=1 CURRENT=0
    SOCKET[0]=serv DESC=3 LISTENING PASSIVE NOTRAP PORT=6321
        ZDELAY ZBFSIZE=1024 ZIBFSIZE=87380 NODELIMITER
```

The following command moves the serv socket from the socketpool to the tcp2 device.

```
GTM>use tcp2:attach="serv"
GTM>use 0 zshow "D"
/dev/pts/9 OPEN TERMINAL NOPAST NOESCA NOREADS TYPE WIDTH=80 LENG=24
s2 OPEN SOCKET TOTAL=1 CURRENT=0
   SOCKET[0]=serv DESC=3 LISTENING PASSIVE NOTRAP PORT=6321
       ZDELAY ZBFSIZE=1024 ZIBFSIZE=87380 NODELIMITER
seerv OPEN SOCKET TOTAL=1 CURRENT=0
   SOCKET[0]=serv2 DESC=4 LISTENING PASSIVE NOTRAP PORT=6322
       ZDELAY ZBFSIZE=1024 ZIBFSIZE=87380 NODELIMITER
YGTMSOCKETPOOL OPEN SOCKET TOTAL=0 CURRENT=-1
```

DOWNSCROLL

DOWNSCROLL Applies to: TRM

If $Y=0, DOWNSCROLL does nothing. Otherwise, DOWNSCROLL moves the cursor up one line on the terminal screen and decrements $Y by one. DOWNSCROLL does not change the column position or $X. Some terminal hardware may not support DOWNSCROLL.

ECHO

[NO]ECHO Applies to: TRM

Enables or disables the echo of terminal input. If you disable ECHO, the EDITING functions will be disabled and any input is not available for later recall.

By default, terminal input ECHOes.

Example:

```
use $principal:noecho
```

This example disables the echo of terminal input.

EDITING

[NO]EDITING Applies to: TRM

Enables the EDITING mode for the $PRINCIPAL device. If you enable EDITING, GT.M allows the use of the left and right cursor movement keys and certain <CTRL> characters within the current input line. You can recall the last input line using the up or down arrow key. The editing functions are the same as during direct mode command input as described in the "Line Editing" section of the "Operating & Debugging in Direct Mode" chapter except that backspace is not treated the same as the erase character from terminfo which is usually delete (ASCII 127). NOECHO disables EDITING mode.

Set the environment variable gtm_principal_editing to specify the mode for EDITING. For example, gtm_principal_editing="EDITING" enables EDITING mode at GT.M startup. You can also specify the mode for INSERT. For example, gtm_principal_editing="NOINSERT:EDITING". If you specify both modes then separate them with a colon (":") and put them in any order.

By default, EDITING mode is disabled.

If you enable the EDITING mode, escape sequences do not terminate READs.

Enabling PASTHRU mode supersedes EDITING mode.

If any of the EDITING <CTRL> characters are in the CTRAP list, their editing functions are not available since CTRAP takes precedence. However, EDITING <CTRL> characters take precedence over the TERMINATOR list.

> **Note**
>
> M READ EDITING depends on the values of $X and $Y being correct. If the application sends its own escape sequences or control characters, which change the cursor position, it must properly update $X and $Y before doing a M READ with EDITING enabled to ensure correct formatting during input.

EMPTERM

[NO]EMPT[ERM] Applies to: TRM

Allows an "Erase" character on an empty input line to terminate a READ or READ # command. The default is NOEMPTERM. The gtm_principal_editing environment variable specifies the initial setting of [NO]EMPTERM. The TERMINFO specified by the current value of the TERM environment variable defines capnames values "kbs" and/or "kdch1" with character sequences for "Erase." If "kbs" or "kdch1" are multi-character values, you must also specify the ESCAPE or EDIT deviceparameters for EMPTERM recognition.

The erase character as set and shown by stty also terminates a READ command with an empty input line. You can set this erase character to various values using the stty shell command. Typical values of an erase character are <CTRL-H> and <CTRL-?>. Characters set and shown with stty setting must match what the terminal emulator sends.

The environment variable TERM must specify a terminfo entry that matches both what the terminal (or terminal emulator) sends and expects.

ERASELINE

ERASELINE Applies to: TRM

Clears the current line from the physical cursor position to the end of the line. ERASELINE does not affect the physical cursor position, or $X and $Y.

ESCAPE

[NO]ESCAPE Applies to: TRM

Enables or disables GT.M processing of escape sequences.

The following events result when a terminal has ESCAPE sequence processing enabled. When an <ESC> or <CSI> arrives in the terminal input, the device driver verifies the sequence that follows as a valid ANSI escape sequence, terminates the READ, and sets $ZB to contain the entire escape sequence. In the case of a READ * when ESCAPE sequence processing is enabled and an escape introducer is read, the entire escape sequence is returned in $ZB and the ASCII representation of the first character is returned in the argument of the READ *.

When escape processing is disabled, READ *x returns 27 in x for an <ESC>. If the escape introducer is also a TERMINATOR, $ZB has a string of length one (1), and a value of the $ASCII() representation of the escape introducer; otherwise, $ZB holds the empty string. For single character and short fixed reads with NOESCAPE, the remaining characters in the escape sequence will be in the input stream for subsequent READS regardless of [NO]TYPEAHEAD.

An application that operates with (NOESCAPE:TERM=$C(13)) must provide successive READ * commands to remove the remaining characters in the escape sequence from the input stream.

By default, ESCAPE processing is disabled.

Example:

```
use $principal:(noescape:term=$c(13))
```

This example disables the escape sequence processing and set $c(13) as the line terminator.

EXCEPTION

EXCEPTION=expr Applies to: All devices

Defines an error handler for an I/O device. The expression must contain a fragment of GT.M code (for example, GOTO ERRFILE) that GT.M XECUTEs when the driver for the device detects an error, or an entryref to which GT.M transfers control, as appropriate for the current gtm_ztrap_form.

For more information on error handling, refer to Chapter 13: "Error Processing" (page 562).

FFLF

[NO]FFLF Applies to: SD

The [NO]FFLF deviceparameter controls whether WRITE # produces only a form-feed (<FF>) or a form-feed and line-feed (<FF><LF>). Previously, GT.M used <FF><LF> which deviated from the standard, but out of concern for existing practice the default remains <FF><LF>.

Additionally, the "gtm_nofflf" environment variable controls the default WRITE # behavior of GT.M. If it is unset or set to 0, N[O] or F[ALSE], the default behavior is unchanged. If it is set to 1, Y[ES] or T[RUE], the default behavior of WRITE # is changed to produce only a form-feed (<FF>), though M programs can still control behavior by specifying the FFLF deviceparameter.

For an example, refer to the description of FFLF deviceparameter of OPEN.

FILTER

[NO]FILTER[=expr] Applies to: TRM SOC NULL

Specifies character filtering for specified cursor movement sequences. Filtering requires character by character examination of all output and reduces I/O performance.

Each FILTER deviceparameter can have only one argument. However, multiple FILTER deviceparameters can appear in a single USE command, each with different arguments.

The valid values for expr:

- [NO]CHARACTERS enables or disables maintenance of $X and $Y according to the M ANSI standard for the characters <BS>, <LF>, <CR> and <FF>. CHARACTERS causes the device driver to examine all output for the above characters, and to adjust $X and $Y accordingly. By default, GT.M performs special maintenance on $X and $Y only for M format control characters, WRAPped records, and certain action deviceparameters.

- In UTF-8 mode, FILTER recognizes the line terminators specified by the Unicode® standard.

- [NO]ESCAPE alters the effect of ANSI escape sequences on $X and $Y. ESCAPE causes GT.M to filter the output, searching for ANSI escape sequences and preventing them from updating $X and $Y. By default, GT.M does not screen output for escape sequences.

By default, GT.M does not perform output filtering. For GT.M to maintain $X for non-graphic characters as described by the standard, FILTER="CHARACTERS" must be enabled. Output filtering adds additional overhead to I/O processing.

Example:

```
use tcpdev:filter="NOESCAPE"
```

This example removes the effect of escape sequences on the maintenance $X and $Y.

FOLLOW

[NO]FOLLOW Applies to: SD

Configures READ to return only when it has a complete record or reaches any specified timeout; it waits for more input rather than terminating on an EOF (end-of-file) condition.

The USE command can switch a device from NOFOLLOW to FOLLOW or from FOLLOW to NOFOLLOW. This provides a READ mode of operation similar to a tail -f in UNIX.

HOSTSYNC

[NO]HOSTSYNC Applies to: TRM

Enables or disables the use of XON/XOFF by the host to throttle input and prevent impending buffer overruns for a terminal. This deviceparameter provides a control mechanism for the host over asynchronous communication lines to help prevent data loss when hardware is slow and/or processing load is high.

By default, HOSTSYNC is disabled.

HUPENABLE

[NO]HUPENABLE Applies to: TRM and SOC

Enables or disables the recognition by the process of the loss ("hang up") of the PRINCIPAL device terminal. When enabled the process receives a SOCKHANGUP or TERMHANGUP error if the O/S signals that the device assigned to the process as the PRINCIPAL device has disconnected. In addition, a SOCKHANGUP or TERMHANGUP error implicitly sets the device to NOHUPENABLE so if a process anticipates multiple disconnects/hangups, it should explicitly issue a USE $PRINCIPAL:HUPENABLE. If GT.M is configured to ignore such a signal, a process may subsequently receive an IOEOF or a SOCKWRITE/TERMWRITE error from an attempt to respectively READ from, or WRITE to the missing device. GT.M terminates a process that ignores more than one of these messages and, if the process is not in Direct Mode, sends a NOPRINCIO message to the operator log.

If defined, the gtm_hupeable environment variable determines the initial process behavior, and if that is undefined GT.M does not immediately report a terminal disconnect.

IKEY

Applies to: SD, PIPE, and FIFO

IKEY allows the use of a seperate key to READ from a device; for example, when a GT.M process is an element of a UNIX pipe. The format of the IKEY deviceparameter is:

IKEY="key_name [IV]"

key_name is case-sensitive and must match a key name in the "files" section of the gtmcrypt_config file. The optional IV specifies an initialization vector to use for encryption and decryption.

For more information, refer to the description of KEY deviceparameter of OPEN.

INREWIND

Applies to: SD

Performs a REWIND on input when $PRINCIPAL identifies a device that supports REWIND. Use this deviceparameter with $PRINCIPAL when redirected from a file. For more information, refer to "SEEK=strexpr" (page 430).

INSEEK=strexpr

Applies to: SD

Performs a SEEK on input when $PRINCIPAL identifies a device that supports SEEK. Use this deviceparameter with $PRINCIPAL when redirected from a file. For more information, refer to "SEEK=strexpr" (page 430).

INSERT

[NO]INSERT Applies to: TRM

Enables or disables insert mode for the $PRINCIPAL device. If INSERT mode is enabled, GT.M inserts input characters at the logical position in the input stream designated by the virtual cursor as defined by $X and $Y, for example in the middle of the line/record. If INSERT mode is disabled, input characters overwrite the existing characters in the input stream at the logical position designated by the virtual cursor. You can toggle the insert mode within a direct mode line or if EDITING is enabled for a single READ argument's input using the terminal's INSERT key. The INSERT mode is reset to the default or what was last specified with USE at the beginning of each direct mode line or READ argument.

IOERROR

IOERROR=expr Applies to: SOC

Enables exception handling in socket devices. expr specifies the I/O error trapping mode. A value equal to "TRAP" specifies that I/O errors on a device raise error conditions. A value equal to "NOTRAP", or when IOERROR is not specified, indicates that an I/O error on a device does not raise error conditions.

> **Note**
>
> GT.M currently handles exception handling at device level instead of socket level.

Example:

```
use sock:(ioerror="TRAP":exception="zgoto "_$zlevel_":error")
```

This example enables exception handling in socket device sock and specifies that all I/O errors on sock raise the error condition.

If $LENGTH(strexpr)&("Tt"[$EXTRACT(strexpr)) then Error Trapping is enabled; otherwise the application must check $DEVICE for errors.

KEY

Applies to: SD, PIPE, and FIFO

Specifies information about the key file to use for reading and writing encrypted data. The syntax of the KEY deviceparameter is as follows:

KEY="key_name [IV]"

key_name is case-sensitive and must match a key name in the "files" section of the gtmcrypt_config file. The optional IV specifies an initialization vector to use for encryption and decryption.

For more information and an example, refer to the description of KEY deviceparameter of OPEN.

LENGTH

[Z]LENGTH=intexpr Applies to: TRM SOC SD FIFO PIPE NULL

Sets the virtual page length for an I/O device to the integer expression. You can specify the virtual page length up to 1,048,576. The page length controls the point at which the device driver automatically resets $Y to 0.

By default, for terminals, GT.M uses the terminfo variable lines (which may be from the terminal definition or from a stty command) as the initial value for LENGTH. The default length for null device and socket device is 66.

Setting LENGTH to zero prevents resetting $Y to zero.

Example:

```
use sock:(width=80:znoff:zlength=24)
```

This example sets the virtual page length to 24 for socket device sock.

LISTEN

LISTEN=expr Applies to: SOC

A new socket is allocated to listen for a connection. It is made the current socket for the device, if the operation is successful.

expr specifies the protocol and the protocol-specific information. Currently, GT.M supports TCP/IP and LOCAL (also known as UNIX domain) socket protocols.

For more information, refer to "LISTEN" (page 402).

Example:

```
GTM>set tcp="seerv" open tcp:(listen="6321:TCP":attach="serv")::"SOCKET"
GTM>use tcp:listen="6322:TCP"
GTM>use 0 zshow "D"
/dev/pts/9 OPEN TERMINAL NOPAST NOESCA NOREADS TYPE WIDTH=80 LENG=24
seerv OPEN SOCKET TOTAL=2 CURRENT=1
SOCKET[0]=serv DESC=3 LISTENING PASSIVE NOTRAP PORT=6321
     ZDELAY ZBFSIZE=1024 ZIBFSIZE=87380 NODELIMITER
SOCKET[1]=h12185825450 DESC=4 LISTENING PASSIVE NOTRAP PORT=6322
     ZDELAY ZBFSIZE=1024 ZIBFSIZE=87380 NODELIMITER
```

OKEY

Applies to: SD, PIPE, and FIFO

OKEY allows the use of a seperate key for WRITE to a device; for example, when a GT.M process is an element of a UNIX pipe. The format of the IKEY deviceparameter is:

OKEY="key_name [IV]"

key_name is case-sensitive and must match a key name in the "files" section of the gtmcrypt_config file. The optional IV specifies an initialization vector to use for encryption and decryption.

For more information, refer to the description of KEY deviceparameter of OPEN.

OPTIONS

OPTIONS=expr Applies to: SOC

Specifies setsockopt() options to be set for sockets. The value of the expression is a comma separated list of option names. If the option takes a value, it is given after an equal sign (=) following the name.

The supported options are:

```
KEEPALIVE   a non zero value enables SO_KEEPALIVE.  A zero value disables SO_KEEPALIVE.
KEEPCNT     sets the TCP_KEEPCNT socket value.
KEEPIDLE    sets the TCP_KEEPIDLE socket value.
KEEPINTVL   sets the TCP_KEEPINTVL socket value.
SNDBUF      sets the size of the socket's network send buffer (SO_SNDBUF) in bytes.
```

Example:

USE dev:OPTIONS="KEEPALIVE=1,KEEPIDLE=50"

This enables SO_KEEPALIVE and sets TCP_KEEPIDLE to 50 seconds.

> **Note**
>
> Please review the man page for setsockopt for more information on the use of these options. On Linux, "man 7 socket" and "man 7 tcp" provide additional information.

OUTREWIND

Applies to: SD

Performs a REWIND on output when $PRINCIPAL identifies a device that supports REWIND. Use this deviceparameter with $PRINCIPAL when redirected to a file. For more information, refer to "REWIND" (page 430).

OUTSEEK=strexpr

Applies to: SD

Performs a SEEK on output when $PRINCIPAL identifies a device that supports SEEK. Use this deviceparameter with $PRINCIPAL when redirected to a file. For more information, refer to "SEEK=strexpr" (page 430).

PASTHRU

[NO]PASTHRU Applies to: TRM

Enables or disables interpretation of the ERASE character for a terminal. PASTHRU shifts management of handling and response to ERASE characters in the input stream from GT.M to the application code.

Exercise caution with PASTHRU in debugging, because using a PASTHRU terminal in Direct Mode is somewhat awkward.

[NO]TTSYNC must be used with [NO]PASTHRU to control XON/XOFF handling.

By default, the device driver operates NOPASTHRU.

PASTHRU supersedes line editing.

READSYNC

[NO]READSYNC Applies to: TRM

Enables or disables automatic output of <XON> before a READ and <XOFF> after a READ.

By default, the terminal drivers operate NOREADSYNC.

REWIND

REWIND Applies to: SD

REWIND places the file pointer to the beginning of the file.

By default, USE does not REWIND.

REWIND on redirected output for $PRINCIPAL is the same as OUTREWIND.

SEEK=strexpr

SEEK Applies to: SD

Positions the current file pointer to the location specified in strexpr. The format of strexpr is a string of the form "[+|-]integer" where an unsigned value specifies an offset from the beginning of the file, and an explicitly signed value specifies an offset relative to the current file position. For STREAM or VARIABLE format, the positive intexpr after any sign is a byte offset, while for a FIXED format, it is a record offset. In order to deal with the possible presence of a Byte Order Marker (BOM), SEEK for a FIXED format file written in a UTF character set must follow at least one prior READ since the device was created.

SEEK on redirected input for $PRINCIPAL is the same as INSEEK.

SOCKET

SOCKET=expr Applies to: SOC

Makes the socket specified by the handle named in expr the current socket for the Socket device . If the named socket is a listening socket, it checks for an incoming connection request and if one is available, it accepts the request and creates a new connected socket in which case $KEY provides information on the new socket Specifying a socket handle not contained in the Socket device generates an error.

> **Note**
>
> SOCKET is compatible with DELIMITER only.

For a usage example, refer to the socketexamplemulti2.m in the Section : "Socket Device Examples" (page 390).

TERMINATOR

[NO]TERMINATOR[=expr] Applies to: TRM

Specifies which of the 256 ASCII characters terminate a READ. For example, TERMINATOR=$C(0) makes <NUL> the terminator.

When NOESCAPE is in effect, TERMINATOR controls whether or not <ESC> or <CSI> are treated as terminators, however, when ESCAPE processing is enabled, the entire escape sequence is treated as a terminator regardless of the TERMINATOR specification.

When EDITING is enabled, the control characters used for editing are not treated as terminators even if they are in the TERMINATOR list.

You can define any control character as a terminator, but they are all single character.

When the terminal is in UTF-8 mode (chset=utf8,) GT.M limits the terminator characters to the first 127 which are common between ASCII and UTF-8 encodng. In M mode, any of the 256 characters may be specified a terminator.

In UTF-8 mode, if CR is in the terminator list (either by default or explicitly,) GT.M ignore the following LF to keep with the standard Unicode® line terminator definitions.

NOTERMINATOR eliminates all terminators. When a terminal has all terminators disabled, fixed length READ and READ * terminate on receipt of some number of characters, and a timed READ terminates on timeout, but any other READ only terminates when the input fills the terminal read buffer.

By default, terminals recognize <CR>, <LF>, and <ESC> as terminators (that is, TERMINATOR=$C(10, 13,27)). TERMINATOR="" restores the default. In UTF-8 mode, the usual UTF-8 line terminators are also included in the default set of terminators.

Example:

```
GTM> USE $P:TERM=$C(26,13,11,7)
```

This example enables the ASCII characters <SUB>, <CR>, <VT> and <BEL> as READ terminators.

TRUNCATE

[NO]TRUNCATE Applies to: SD

Enables or disables overwriting of existing data in sequential files. Because the position of each record depends on the prior record, a WRITE destroys the ability to reliably position to subsequent records in a file. Therefore, by default (NOTRUNCATE), GT.M permits WRITEs only when the file pointer is positioned at the end-of-file. When a device has TRUNCATE enabled, a WRITE issued when the file pointer is not at end-of-file truncates the file by destroying all data from the file pointer to the end-of-file.

By default, OPEN accesses files NOTRUNCATE, which does not allow overwriting of sequential files.

This deviceparameter may not be supported by your platform.

TTSYNC

[NO]TTSYNC Applies to: TRM

Enables or disables recognition of XON/XOFF for terminal output.

> **Note**
>
> A terminal may have its own handling of XON/XOFF, controlled by a set-up mode or by switches. If an application requires program recognition of <CTRL-S> and <CTRL-Q>, the terminals may require reconfiguration.

TYPEAHEAD

[NO]TYPEAHEAD Applies to: TRM

Enables or disables type-ahead buffering for a terminal. When TYPEAHEAD is disabled, any pending input which has not yet been read will be discarded before input is read for each READ argument. When TYPEAHEAD is enabled, any input not read by one READ argument will remain available for the next READ argument or command.

The size of the type-ahead buffer limits the amount of data entered at the terminal that the device driver can store in anticipation of future READs.

By default, the terminal device driver accepts TYPEAHEAD.

UPSCROLL

UPSCROLL Applies to: TRM

Moves the cursor down one line on the terminal screen. If $Y=LENGTH-1, UPSCROLL sets $Y=0. Otherwise UPSCROLL increments $Y by one. If the cursor is physically at the bottom of the page, the screen scrolls up one line. UPSCROLL does not change the column position or $X.

WIDTH

[Z]WIDTH=intexpr Applies to: TRM SOC NULL SD FIFO PIPE

Sets the device's logical record size and enables WRAP. The default WIDTH for SD, FIFO, and PIPE is taken from the RECORDSIZE.

NOWRAP and WIDTH supersede each other. When WIDTH and NOWRAP appear together on the same USE command, the final one controls the device behavior.

In M mode if WIDTH is set to 0, GT.M uses the default WIDTH of the TRM and SOC devices, in other words: USE x:WIDTH=0 is equivalent to USE x:(WIDTH=<device-default>:NOWRAP.

Terminals inherit their default WIDTH in GT.M from the invoking shell environment. The default WIDTH for null and SOCKET devices is 255.

For SD and SOC which support 1MiB strings, you can specify WIDTH up to 1,048,576.

In UTF-8 mode and TRM, SD, FIFO, and PIPE output, the WIDTH deviceparameter is in units of display-columns and is used with $X to control truncation and WRAPing of output and maintenance of $X and $Y.

In UTF-8 mode and SOC, the WIDTH deviceparameter is in units of UTF-8 code points (characters), and is used with $X to control truncation and wrapping for output and maintenance of $X and $Y.

Along with device WIDTH and WRAP, GT.M format control characters, and the FILTER device characteristic, also have an effect on $X.

WRAP

[NO]WRAP Applies to: TRM SOC NULL SD FIFO PIPE

Enables or disables automatic record termination. When the current record size ($X) reaches the maximum WIDTH and the device has WRAP enabled, GT.M starts a new record, as if the routine had issued a WRITE ! command. When reading, WRAP only determines whether $X remains within the range of zero to WIDTH.

Note that WRAP is enabled by default for SD, NULL, FIFO, PIPE and SOCKET. For TRM, WRAP is enabled by default if the terminfo variable auto_right_margin (capname "am") is set.

NOWRAP causes GT.M to require a WRITE ! to terminate the record. NOWRAP allows $X to become greater than the device WIDTH for terminals and null devices.

The combination of STREAM and NOWRAP on disk files allows you to write data of arbitrary length without truncation. Without the STREAM option, the WRAP option determines the action taken when the record length exceeds the device WIDTH. NOWRAP causes GT.M to truncate the record, while WRAP causes GT.M to insert a format control character except for FIXED format.

Example:

See WRAP examples in the OPEN deviceparameters section.

X

X=intexpr Applies to: TRM

$X positions the cursor to a vertical column on the terminal. If NOWRAP is enabled or intexpr<WIDTH, GT.M sets $X=intexpr. If WRAP is enabled and intexpr>WIDTH, GT.M sets $X=intexpr#WIDTH, where # is the GT.M modulo operator. The resulting $X determines the actual physical position.

To ensure that $Y and $X match what is occurring visually on the terminal, the GT.M deviceparameters and the device characteristics must match at all times.

The terminal hardware may affect physical cursor positioning. The X deviceparameter does not change the cursor row or update $Y.

See Also

- "Y" (page 433)
- "Using Terminals" (page 353)
- "Maintenance of $X and $Y" (page 348)

Y

Y=intexpr Applies to: TRM

Positions the cursor to a horizontal row on the terminal.

GT.M sets $Y=intexpr#LENGTH, where # is the GT.M modulo operator. If intexpr<LENGTH, the resulting $Y determines the physical position. If intexpr>LENGTH, the cursor is positioned so that $Y=intexpr#LENGTH, where # is the GT.M module operator. The terminal hardware may affect physical cursor positioning.

To ensure that $Y and $X match what is occurring visually on the terminal, the GT.M deviceparameters and the device characteristics must match at all times. For example, if a process initiates a subprocess that changes the terminal wrap setting from NOWRAP, previously set with the GT.M USE command to WRAP , GT.M does not reflect the change when the subprocess completes. Therefore, wraps on the terminal do not reflect in the values of $X and $Y.

The Y deviceparameter does not change the cursor column or update $X.

See Also	
	• "X" (page 433)
	• "Using Terminals" (page 353)
	• "Maintenance of $X and $Y" (page 348)

ZBFSIZE

ZBFSIZE Applies to: SOC

Allocates a buffer used by GT.M when reading from a socket. The ZBFSIZE deviceparameter should be at least as big as the largest message expected.

By default, the size of ZBFSIZE is 1024 and the maximum it can be is 1048576.

ZDELAY

Z[NO]DELAY Applies to: SOC

Controls buffering of data packets by the system TCP stack using the TCP_NODELAY option to the SETSOCKOPT system call. This behavior is sometimes known as the Nagle algorithm. The default is ZDELAY. This delays sending additional packets until either an acknowledgement of previous packets is received or an interval passes. If several packets are sent from one end of a connection before the other end responds, setting ZNODELAY may be desirable though at the cost of additional packets being transmitted over the network. ZNODELAY must be fully spelled out.

ZFF

Z[NO]FF=expr Applies to: SOC

expr specifies a string of characters, typically in $CHAR() format to send to socket device, whenever a routine issues a WRITE #. When no string is specified or when ZFF="", then no characters are sent. The default in GT.M is ZNOFF.

Example:

```
u tcpdev:(zwidth=80:zff=$char(13):zlength=24)
```

This example sends $char(13) to the current socket of device tcpdev on every WRITE #.

ZIBFSIZE

ZIBFSIZE Applies to: SOC

Sets the buffer size used by the network software (setsockopt SO_RCVBUF).

The default and the maximum values depend on the platform and/or system parameters.

Note that LOCAL sockets ignore the ZIBFSIZE deviceparameter.

USE Deviceparameters Summary

USE Deviceparameters						
USE DEVICEPARAMETER	TRM	SD	FIFO	PIPE	NULL	SOC
ATTACH						X
CANONICAL	X					
[NO]CENABLE	X					
[I|O]CHSET	X	X	X	X	X	X
CLEARSCREEN	X					
CONNECT						X
[NO]CONVERT	X					
CTRAP=expr	X					
[NO]DELIMITER						X
DETACH=expr						X
DOWNSCROLL	X					
[NO]ECHO	X					
[NO]EMPTERM	X					
ERASELINE	X					
[NO]ESCAPE	X					
EXCEPTION=expr	X	X	X	X	X	X
[NO]FFLF		X				
[NO]FILTER[=expr]	X				X	X
FLUSH	X					X
[NO]FOLLOW		X				
[NO]HOSTSYNC	X					
[NO]HUPENABLE	X					X
TRM: Valid for terminals and printers						
SD: Valid for sequential files						
FIFO: Valid for FIFOs						
PIPE: Valid for PIPE devices						
NULL: Valid for null devices						
SOC: Valid for socket devices						

USE Deviceparameters						
USE DEVICEPARAMETER	**TRM**	**SD**	**FIFO**	**PIPE**	**NULL**	**SOC**
IKEY		X	X	X		
INREWIND		X				
INSEEK		X				
IOERROR						X
KEY		X	X	X		
[Z]LENGTH=expr	X	X	X	X	X	X
LISTEN						X
OKEY		X	X	X		
OUTREWIND		X				
OUTSEEK		X				
[NO]PASTHRU	X					
REWIND		X				
SEEK=strexpr		X				
SOCKET						X
TERMINATOR[=expr]	X					
[NO]TRUNCATE		X				
[NO]TYPEAHEAD	X					
UPSCROLL	X					
[Z]WIDTH=intexpr	X	X	X	X	X	X
[NO]WRAP	X	X	X	X	X	X
X=intexpr	X				X	
Y=intexpr	X				X	
ZBFSIZE						X
Z[NO]DELAY						X
TRM: Valid for terminals and printers						
SD: Valid for sequential files						
FIFO: Valid for FIFOs						
PIPE: Valid for PIPE devices						
NULL: Valid for null devices						
SOC: Valid for socket devices						

USE Deviceparameters						
USE DEVICEPARAMETER	TRM	SD	FIFO	PIPE	NULL	SOC
Z[NO]FF						X
ZIBFSIZE						X
TRM: Valid for terminals and printers						
SD: Valid for sequential files						
FIFO: Valid for FIFOs						
PIPE: Valid for PIPE devices						
NULL: Valid for null devices						
SOC: Valid for socket devices						

READ

The READ command transfers input from the current device to a global or local variable specified as a READ argument. For convenience, READ also accepts arguments that perform limited output to the current device.

The format of the READ command is:

```
R[EAD][:tvexpr] glvn|*glvn|glvn#intexpr|strlit|fcc[,...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- A subscripted or unsubscripted global or local variable name specifies a variable into which to store the input; the variable does not have to exist prior to the READ; if the variable does exist prior to the READ, the READ replaces its old value.

- For fixed format files, READ X always read WIDTH characters.

- For VARIABLE or STREAM format files, READ X reads up to WIDTH characters, stopping if a line terminator or end of file is found first.

- When an asterisk (*) immediately precedes the variable name, READ accepts one character of input and places the ASCII code for that character in the variable.

- When a number sign (#) and a non-zero integer expression immediately follow the variable name, the integer expression determines the maximum number of characters accepted as input to the read; such reads terminate when GT.M reads the number of characters specified by the integer expression or a terminator in the input stream, whichever occurs first.

- To provide a concise means of issuing prompts, GT.M sends string literal and format control character (!,?intexpr,#) arguments of a READ to the current device as if they were arguments of a WRITE.

- An indirection operator and an expression atom evaluating to a list of one or more READ arguments form a legal argument for a READ.

The maximum length of the input string is the smaller of the device buffer size limitation or the GT.M maximum string size (1,048,576 bytes). If a record is longer than the maximum record length, GT.M returns the record piece by piece during sequential reads, for devices that allow it.

When a string literal appears as an argument to a READ, M writes the literal to the current device. String literals appear as READ arguments to serve as prompts for input. GT.M does not permit expression arguments on a READ to act as prompts. Variable prompts must appear as arguments to a WRITE. If a variable appears as an argument to a READ, GT.M always interprets it as input, never as output. This facility is used mostly with terminal I/O.

The READ commands adjust $X and $Y, based on the length of the input read.

In UTF-8 mode, the READ command uses the character set value specified on the device OPEN as the character encoding of the input device. If character set "M" or "UTF-8" is specified, the data is read with no transformation. If character set is "UTF-16", "UTF-16LE", or "UTF-16BE", the data is read with the specified encoding and transformed to UTF-8. If the READ command encounters an illegal character or a character outside the selected representation, it produces a run-time error. The READ command recognizes all Unicode® line terminators for non-FIXED devices. See "Line Terminators" section for more details. In M mode, characters and bytes have a one-to-one relationship and therefore READ can be used to read bit-streams of non-character data.

READ * Command

The READ * command reads one character from the current device and returns the decimal ASCII representation of that character into the variable specified for the READ * command. READ * appears most frequently in communication protocols, or in interactive programs where single character answers are appropriate.

In UTF-8 mode, the READ * command accepts one Unicode® character input and puts the numeric code-point value for that character into the variable. The READ * command reads one to four bytes, depending on the encoding and returns the numeric code-point value of the character. If ICHSET specifies "UTF-16", "UTF-16LE" or "UTF-16BE", the READ * command reads a byte pair or two byte pairs (if it is a surrogate pair) and returns the numeric code-point value. If ICHSET is M, the READ * command reads a single byte and returns the numeric byte value just like in M mode.

The following example reads the value "A", and returns the decimal ASCII representation of "A" in the variable X.

Example:

```
GTM> READ *X
A
GTM> WRITE X
65
```

If a timeout occurs before GT.M reads a character, the READ * returns a negative one (-1) in the variable.

```
GTM>Set filename="mydata.out"; assume that mydata.out contains "主要雨在西班牙停留在平原".
GTM>Open filename:(readonly:ichset="UTF-16LE")
GTM>Use filename
GTM>Read *x
GTM>Close filename
GTM>Write $char(x)
```

In this example, the READ * command reads the first character of the file mydata.out according to the encoding specified by ICHSET.

READ X#maxlen Command

The READ X#maxlen command limits the maximum size of the input to a maximum of "maxlen" characters, where maxlen is an integer expression.

If a READ follows a READ X#maxlen command, the READ returns the remainder of the current record.

If a terminator arrives before maxlen characters are received the READ X#maxlen terminates.

For fixed format files, If WIDTH - $X is greater than len, READ X#maxlen reads maxlen characters otherwise READ reads WIDTH - $X characters. Fewer may be returned if end of file is reached.

For VARIABLE or STREAM format files, READ X#maxlen reads up to MIN(maxlen, WIDTH - $X) characters, stopping if it finds line terminator or end of file.

Write

The WRITE command transfers a character stream specified by its arguments to the current device.

The format of the WRITE command is:

```
W[RITE][:tvexpr] expr|*intexpr|fcc[,...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- An expression argument supplies the text of a WRITE.

- When a WRITE argument consists of a leading asterisk (*) followed by an integer expression, WRITE outputs one character associated with the code specified by the integer evaluation of the expression.

- WRITE also accepts format control characters as arguments; format control characters modify the position of a virtual cursor: an exclamation point (!) produces the device specific record terminator (for example, new line for a terminal), a number sign (#) produces device specific page terminator (for example, form feed for a terminal) and a question mark (?) followed by an expression moves the virtual cursor to the column specified by the integer evaluation of the expression if the virtual cursor is to the "left" of the specified column.

- When directed to a device bound to a mnemonicspace, WRITE also accepts controlmnemonics, which are keywords specific to the binding – they are delimited by a slash (/) prefix and optionally followed by a parenthetical list of arguments. The parentheses "( )" are optional when there are no arguments, but must appear even if there is a single argument

- An indirection operator and an expression atom evaluating to a list of one or more WRITE arguments form a legal argument for a WRITE.

- In the UTF-8 mode, the WRITE command uses the character set specified on the device OPEN as the character encoding of the output device. If character set specifies "M" or "UTF-8", GT.M WRITEs the data with no transformation. If character set specifies "UTF-16", "UTF-16LE" or "UTF-16BE", the data is assumed to be encoded in UTF-8 and WRITE transforms it to the character encoding specified by character set device parameter.

- If a WRITE command encounters an illegal character in UTF-8 mode, it produces a run-time error irrespective of the setting of VIEW "BADCHAR".

GT.M can write up to 1,048,576 bytes (the GT.M maximum string size) as a result of a single WRITE argument. GT.M buffers output into a "logical record" for all devices except sockets without DELIMITERs and sequential devices with STREAM enabled. The WRITE command appends a string to the current record of the current device. GT.M does not write to the output device until the buffer is full, a GT.M format control character forces a write, a USE command, a CLOSE command, or, for terminals, the buffer becomes stale. The GT.M compiler breaks a concatenated WRITE argument into a series of WRITE arguments to eliminate the overhead of the concatentation. If circumstances provide a reason for a single WRITE, perform the concatenation prior to the WRITE.

Each device has a WIDTH and a LENGTH that define the virtual "page". The WIDTH determines the maximum size of a record for a device, while the LENGTH determines how many records fit on a page. When the current record size ($X) reaches the maximum WIDTH and the device has WRAP enabled, GT.M starts a new record. When the current line ($Y) reaches the maximum LENGTH, GT.M starts a new page.

For devices OPENed with a CHSET supported in UTF-8 mode, WRITE * takes intexpr as a code-point and writes the associated Unicode® character in the encoding specified by CHSET. For devices OPENed in M mode, WRITE * takes intexpr as an ASCII value and writes the associated ASCII character.

The WRITE command also has several format control characters that allow the manipulation of the virtual cursor. For all I/O devices, the GT.M format control characters do the following:

- WRITE !: Clears $X and increments $Y and terminates the logical record in progress. The definition of "logical record" varies from device to device, and is discussed in each device section.

- WRITE #: Clears $X and $Y and terminates the logical record in progress.

- WRITE ?n: If n is greater than $X, writes n-$X spaces to the device, bringing $X to n. If n is less than or equal to $X, WRITE ?n has no effect. When WRAP is enabled and n exceeds the LENGTH of the line, WRITE ?n increments $Y.

> **Note**
>
> If $X is less than WIDTH, WRITE ! writes WIDTH - $X spaces.

For devices OPENed with a CHSET supported in UTF-8 mode, WRITE * takes intexpr as a code-point and writes the associated Unicode character in the encoding specified by CHSET. For devices OPENed in M mode, WRITE * takes intexpr as an ASCII value and writes the associated ASCII character.

In UTF-8 mode, if a WRITE command encounters an illegal character, it produces a run-time error irrespective of the setting of VIEW "BADCHAR".

For more information, see the sections on specific I/O devices.

WRITE *

When the argument of a WRITE command consists of a leading asterisk (*) followed by an integer expression, the WRITE command outputs the character represented by the code-point value of that integer expression.

With character set M specified at device OPEN, the WRITE * command transfers the character (byte) associated with the numeric value of the integer expression. With character UTF-8 specified at device OPEN, the WRITE command outputs the character associated with the numeric code-point value. If character set "UTF-16", "UTF-16LE" or "UTF-16BE" is specified, WRITE * transforms the character code to the mapping specified by that character set.

Close

The CLOSE command breaks the connection between a process and a device.

The format of the CLOSE command is:

```
C[LOSE][:tvexpr] expr[:(keyword[=expr][:...])][,...]
```

- The optional truth-valued expression immediately following the command is a command postconditional that controls whether or not GT.M executes the command.

- The required expression specifies the device to CLOSE.

- The optional keywords specify deviceparameters that control device behavior; some deviceparameters take arguments delimited by an equal sign (=); if there is only one keyword, the surrounding parentheses are optional.

- An indirection operator and an expression atom evaluating to a list of one or more CLOSE arguments form a legal argument for a CLOSE.

When a CLOSE is issued, GT.M flushes all pending output to the device, and processes any deviceparameters. CLOSEing a device not currently OPEN has no effect.

If a partial record has been output, a WRITE ! is done to complete it. To suppress this action, set $X to zero before the CLOSE.

GT.M retains the characteristics of all device types, except a sequential file, for use in case of subsequent re-OPENs. If the device is a sequential file, characteristics controlled by deviceparameters are lost after the CLOSE.

If the device being CLOSEd is $IO, GT.M implicitly USEs $PRINCIPAL. GT.M ignores CLOSE $PRINCIPAL.

Example:

```
CLOSE SD:RENAME=SD_".SAV"
```

This closes the device and, if it is a disk file, renames it to have the type .SAV.

Example:

```
CLOSE SOCKDEV:(SOCKET="LOCALSOCK1":DELETE)
```

This deletes the socket file associated with LOCALSOCK1 if it is a LOCAL socket and closes only the named socket on the socket device.

CLOSE Deviceparameters

DELETE

DELETE Applies to: SD FIFO SOC(LOCAL)

Instructs GT.M to delete the disk file after GT.M closes it.

DESTROY

[NO]DESTROY Applies to: SD, FIFO, SOC

Determines whether the process retains device characteristics after CLOSE. The default is DESTROY for sequential disk files and FIFO devices and NODESTROY for SOCKET devices. While NODESTROY allows the re-OPEN of previously CLOSE'd device with the same characteristics as when it was last CLOSE'd, as described by the M standard, every tracked device uses process memory. A device that has been DESTROYed on CLOSE cannot be re-opened with the previous characteristics, but reclaims memory used by the process for that device. [NO]DESTROY is ignored for CLOSE of a specific socket rather than the entire socket device.

Because it forms a communication link with a process it creates, CLOSE of a PIPE device always eliminates the device and hence ignores any [NO]DESTROY deviceparameter.

EXCEPTION

EXCEPTION=expr Applies to: All devices

Defines an error handler for an I/O device. The expression must contain a fragment of GT.M code (for example, GOTO ERRFILE) that GT.M XECUTEs when the driver for the device detects an error, or an entryref to which GT.M transfers control, as appropriate for the current gtm_ztrap_form.

The expression must contain a fragment of GT.M code (for example, GOTO ERRFILE) that GT.M XECUTEs when the driver for the device detects an error, or an entryref to which GT.M transfers control, as appropriate for the current gtm_ztrap_form.

For more information on error handling, refer to Chapter 13: "Error Processing" (page 562).

GROUP

GROUP=expr Applies to: SOC(LOCAL), SD, FIFO

Specifies access permission on a UNIX file for other users in the file owner's group. The expression is a character string evaluating to null or to any combination of the letters RWX, indicating respectively Read, Write, and eXecute access. When any one of these deviceparameters (OWNER, GROUP, WORLD) appear on a CLOSE of an existing file, any user category, that is not explicitly specified remains unchanged.

In order to modify file security, the user who issues the CLOSE must have ownership.

By default, CLOSE does not modify the permissions on an existing file.

OWNER

OWNER=expr Applies to: SOC(LOCAL) SD FIFO

Specifies access permission on a UNIX file for the owner of the file. The expression is a character string evaluating to null or to any combination of the letters RWX, indicating respectively Read, Write, and eXecute access. When any one of these deviceparameters appear on a CLOSE of an existing file, any user category (GROUP, SYSTEM, WORLD), that is not explicitly specified remains unchanged.

In order to modify file security, the user who issues the CLOSE must have ownership.

By default, CLOSE does not modify the permissions on an existing file.

RENAME

RENAME=expr Applies to: SD

Changes the file name to the name contained in the argument string. CLOSE ignores RENAME when it specifies the same file as that of the CLOSE file-specification. When the files are different, the original file specified by the CLOSE no longer exists. RENAME gives an error if the specified file exists, while REPLACE does not. When the expression omits part of the pathname, GT.M constructs the full pathname by applying the defaults discussed in the section on device specifications.

If the process has sufficient access permissions, it may use RENAME to specify a different directory as well as file name. RENAME cannot move a file to a different filesystem.

REPLACE

REPLACE=expr Applies to: SD

Changes the file name to the name contained in the argument string. REPLACE overwrites any existing file, while RENAME does not. CLOSE ignores REPLACE when it specifies the same file as that of the CLOSE file-specification. When the files are different, the original file specified by the CLOSE no longer exists. When the expression omits part of the pathname, GT.M constructs the full pathname by applying the defaults discussed in the section on device specifications.

If the process has sufficient access permissions, it may use REPLACE to specify a different directory as well as file name. REPLACE cannot move a file to a different filesystem.

SOCKET

SOCKET=expr Applies to: SOC

The socket specified in expr is closed. Specifying a socket that has not been previously OPENed generates an error. If no SOCKET deviceparameter is specified on a CLOSE for a socket device, the socket device and all sockets associated with it are closed.

If the device being CLOSEd is $IO, GT.M implicitly USEs $PRINCIPAL.

SYSTEM

SYSTEM=expr Applies to: SOC(LOCAL) SD FIFO

This deviceparameter is a synonym for OWNER that is maintained in UNIX for compatibility with VMS applications.

By default, CLOSE does not modify the permissions on an existing file.

TIMEOUT

TIMEOUT=expr Applies to: PIPE

Performs a timed check (in seconds) on the termination status of the PIPE co-process of a PIPE device that is not OPEN'd with the INDEPENDENT deviceparameter. intexpr specifies time in seconds. The default is 2 seconds if TIMEOUT is not specified.

UIC

UIC=exprgroup number Applies to: SOC(LOCAL) SD FIFO

Specifies the group that has access to the file. The format of the string is "g,i" where g is a decimal number representing the group portion of the UIC and i is a decimal number representing the individual portion.

Specifies the owner and affects access to the file. The expression evaluates to the numeric identifier of the new owner.

WORLD

WORLD=expr Applies to: SOC(LOCAL) SD FIFO

Specifies access permissions for users not in the owner's group on a UNIX file. The expression is a character string evaluating to null or to any combination of the letters RWX, indicating respectively Read, Write, and eXecute access. When any one of these deviceparameters (OWNER, GROUP, WORLD) appears on a CLOSE of a new file, any user category that is not explicitly specified is given the default character string. When any one of these deviceparameters (OWNER, GROUP, WORLD) appears on a CLOSE of an existing file, any user category , that is not explicitly specified remains unchanged.

In order to modify file security, the user who issues the CLOSE must have ownership.

By default, CLOSE and CLOSE do not modify the permissions on an existing file. Unless otherwise specified, when CLOSE creates a new file, it establishes security using standard defaulting rules.

In order to modify file security, the user who issues the CLOSE must have ownership.

CLOSE Deviceparameters Table

CLOSE Deviceparameters				
CLOSE DEVICEPARAMETER	**TRM**	**SD**	**FIFO**	**SOC**
"DELETE" (page 441)		X	X	X
"DESTROY" (page 441)		X	X	X
"EXCEPTION" (page 442)	X	X	X	X
"GROUP" (page 442)		X	X	
"OWNER" (page 442)		X	X	
"RENAME" (page 442)		X	X	
"REPLACE" (page 442)		X	X	
"REWIND" (page 408)		X		
"SOCKET" (page 443)				X
"SYSTEM" (page 443)		X	X	
"UIC" (page 443)		X	X	
"WORLD" (page 443)		X	X	
SD: Valid for sequential disk files				
TRM: Valid for terminals and printers				
FIFO: Valid for FIFOs				
NULL: Valid for NULL devices				
SOC: Valid for Socket devices				

> **Note**
>
> Since EXCEPTION is the only CLOSE deviceparameter that applies to NULL, the NULL device column is not shown in the table above.

Deviceparameter Summary Table

The following table lists all of the deviceparameters and shows the commands to which they apply.

Deviceparameter Summary			
DEVICEPARAMETER	**OPEN**	**USE**	**CLOSE**
APPEND	X		
ATTACH		X	
BLOCKSIZE=intexpr	X		
[NO]CENABLE		X	
CLEARSCREEN		X	
CONNECT	X	X	
[NO]CONVERT		X	
CTRAP		X	
DELETE			X
[NO]DELIMITER	X	X	
[NO]DESTROY			X
DETACH		X	
DOWNSCROLL		X	
[NO]ECHO		X	
ERASELINE		X	
[NO]ESCAPE		X	
EXCEPTION=expr	X	X	X
[NO]FFLF	X	X	
[NO]FILTER[=expr]		X	
[NO]FIXED	X		
FLUSH		X	
GROUP=expr	X	X	X
KEY	X	X	
IKEY	X	X	
IOERROR=expr	X	X	
[NO]HOSTSYNC		X	
[NO]HUPENABLE		X	
[Z]LENGTH=intexpr		X	
NEWVERSION	X		
OKEY	X	X	

Deviceparameter Summary			
DEVICEPARAMETER	OPEN	USE	CLOSE
OWNER=expr	X	X	X
[NO]PASTHRU		X	
[NO]RCHK	X	X	
[NO]READONLY	X		
RECORDSIZE=intexpr	X		
RENAME=expr			X
REPLACE=expr			X
[NO]RETRY	X	X	
REWIND	X	X	X
SKIPFILE=intexpr		X	
SOCKET		X	X
SPACE=intexpr		X	X
[NO]STREAM	X		
SYSTEM=expr	X		X
TERMINATOR=expr		X	
TIMEOUT=expr			X
[NO]TRUNCATE	X	X	
[NO]TTSYNC		X	
[NO]TYPEAHEAD		X	
UIC=expr	X		X
UPSCROLL		X	
VARIABLE	X		
[Z]WIDTH=intexpr		X	
WORLD=expr	X		X
[Z][NO]WRAP	X	X	
WRITELB=expr		X	
X=intexpr		X	
Y=intexpr		X	
ZBFSIZE	X	X	
Z[NO]DELAY	X	X	

Deviceparameter Summary			
DEVICEPARAMETER	OPEN	USE	CLOSE
Z[NO]FF	X	X	
ZIBFSIZE	X	X	
LISTEN=expr	X	X	

Chapter 10. Utility Routines

Revision History		
Revision V7.1-004	27 June 2024	• In "%YGBLSTAT()" (page 515), Added WFR,BUS,BTS,STG,KTG,ZTG,DEXA,GLB,JNL,MLK,PF to the YGBLSTAT output
Revision V7.1-003	23 November 2023	• In "Deviceparameter Summary Table" (page 444), Add [NO]HUPENABLE to Deviceparameter Summary Table
Revision V7.1-002	19 September 2023	• In " %FREECNT" (page 509), add more information and remove redundant explanation
Revision V7.0-002	23 March 2022	• In "%PEEKBYNAME()" (page 510), formatting cleanup
Revision V7.0-001	24 November 2021	• In "Deviceparameter Summary Table" (page 444), GTM-9452 - Add deviceparameter REPLACE to provide the overwrite functionality
Revision V7.0-000	12 February 2021	• In "%PEEKBYNAME()" (page 510), Add n_wrt_per_flu to the ^%PEEKBYNAME table • In "System Management Utilities" (page 508), removed the %DSEWRAP section
Revision V6.3-012	08 April 2020	• In "%YGBLSTAT()" (page 515), add a note about %YGS and an example. • In "GT.M Utilities Summary Table" (page 519), add %JSWRITE • In "%JSWRITE" (page 464), add %JSWRITE
Revision V6.3-011	20 December 2019	• In "Internationalization Utilities " (page 505), Move I18N utility documentation to this utility chapter to be references in the I18N chapter
Revision V6.3-009	27 June 2019	• In " Conversion Utilities" (page 459), add comments on radix conversions; remove the point topic headings. • In " %RI" (page 499), specify that ^%RI handles lines of up to 1 MiB. • In " %RO" (page 499), specify that ^%RO handles code lines up to 1 MiB.
Revision V6.3-008	24 April 2019	• In "%YGBLSTAT()" (page 515), Add documentation for the IN function
Revision V6.3-007	04 February 2019	• In "%PEEKBYNAME()" (page 510), add gd_region.open to the PEEKBYNAME table

		• In " Examples of %GSEL" (page 492), update examples to reflect new output format.
		• In " %GSEL" (page 491), add clarification for behavior when '?' is the first character of a global name in the search string.
Revision V6.3-006	26 October 2018	• In "%TRIM" (page 481), add information about the second expression and general improvements.
		• In "UTF-8 Mode Utility Routines " (page 518), change title to "UTF-8 Mode Utility Routines".
Revision V6.3-004	23 March 2018	• In "%PEEKBYNAME()" (page 510), Added note about RO database implications
		• In " %GCE" (page 486), change strexpr to expr; add more information about $DEVICE.
		• In " %GSE" (page 491), Updated %GSE entry to contain QUIET Utility Label
		• In " %RCE" (page 495), Updated %RCE entry to include QUIET and QCALL utility labels.
		• In " %RSE" (page 501), Updated %RSE entry to include QUIET and QCALL utility labels.
Revision V6.3-002	22 August 2017	• In "%PEEKBYNAME()" (page 510), add an entry for jnlpool_ctl_struct.instfreeze_environ_inited replication parameter.
Revision V6.3-001	20 March 2017	• In "%PEEKBYNAME()" (page 510), New section: "%PEEKBYNAME()" (page 510). Added asyncio, wcs_Wterror_invoked_cntr and wcs_wip_lvl
		• In " %XCMD" (page 509), made general improvements for clarity. Added examples that did not use xec
		• In "%YGBLSTAT()" (page 515), added the description of %YGBLSTATS.
		• In " Global Utilities" (page 483), added the description of the %ZSHOWVTOLCL utility.
		• In "%GI" (page 489), specified that ^%GI accepts records having up to 1MiB string length.
Revision V6.0-001	21 March 2013	Added the following sections:
		• "String Utilities" (page 481)
		In " %XCMD" (page 509), added the description of the LOOP^%XCMD utility label.

GT.M provides library utilities to perform frequently used tasks, and to access frequently used information. Most of the utilities are for GT.M programmers, but some provide tools for system administration and operation.

The GT.M utilities fall into the following general categories:

- Date and time utilities

- Conversion utilities

- Mathematic utilities

- Global utilities

- Routine utilities

- Internationalization utilities

- System Management utilities

- UTF-8 mode Utility Routines

The GT.M distribution includes the source files for these utilities. The default installation compiles them to produce object modules in the $gtm_dist distribution library.

You may wish to examine the utilities and include some of them in your programs if the programs access the function frequently or you may want to modify the utilities to better fit your particular needs. If you modify a utility, store your copy in a directory that precedes gtm_dist in the search list $ZROUTINES to prevent a new release of GT.M from overwriting your copy.

Using the Utilities

You can either use a utility in Direct Mode or include it in a source application program with one or more of the following formats.

- DO ^%UTILITYNAME

- DO LABEL^%UTILITYNAME

- $$FUNC^%UTILITYNAME[(para1,...)]

Many utilities contain labels that invoke variations of the basic utility functionality. Some also provide the label FUNC to invoke an extrinsic function with optional or required parameters.

GT.M passes input to non-extrinsic forms of the utilities interactively or by using "input" variables. GT.M passes output from non-extrinsic forms of the utilities using "output" variables. For extrinsic entry points, the utilities receive input as parameters and pass output as the returned result. For other entry points, GT.M uses predefined "input" and "output" variables to pass information. Some utilities interactively request user inputs and display their results. Each utility is described individually in this chapter where appropriate labels, input, and output variables are identified.

By convention, the utilities use upper-case variables for external input and output. Since M is case-sensitive, when an invocation uses a lower-case or misspelled variable name, the routine does not output the expected information. Instead it supplies a default value, if one exists, or produces an error message.

Example:

```
GTM>SET %ds="11/22/2010"
```

```
GTM>DO INT^%DATE
GTM>ZWRITE
%DN=62047
%ds="11/22/2010"
```

This example sets the lowercase variable %ds to the date 11/22/2010. Since the %DATE routine expects the input to be provided in the uppercase %DS variable, it returns a default value in the output variable $DN. The default is the $HOROLOG format of the current date, which is 11/17/2010 in the example.

> **Note**
>
> Utility programs written in M (such as %GO) run within mumps processes and behave like any other code written in M. Encryption keys are required if the mumps process accesses encrypted databases. A process running a utility program written in M that does not access encrypted databases (such as %RSEL) does not need encryption keys just to run the utility program.

Date and Time Utilities

The date and time utilities are:

%D: Displays the current date using the [d]d-mmm-[yy]yy format.

%DATE: Converts input date to the $HOROLOG format.

%H: Converts date and time to and from $HOROLOG format.

%T: Displays the current time in [h]h:mm AM/PM format.

%TI: Converts time to $HOROLOG format.

%TO: Converts the current time from $HOROLOG format to [h]h:mm AM/PM format.

The "%" sign has been removed from the topic headings below, intentionally.

The Intrinsic Special Variable $ZDATEFORM interprets year inputs with two digits as described in the following table:

$ZDATEFORM	INTERPRETATION OF 2 DIGIT YEAR	OUTPUT OF %D
0:	20th century (1900 - 1999)	2 digits
1:	current century (2000 - 2099)	4 digits
(1841-9999):	the next 99 years starting from $ZDATEFORM (x - x+99)	4 digits
other:	current century (2000 - 2099)	4 digits

Example:

If $ZDATEFORM is 1965, an input year of 70 would be interpreted as 1970, whereas an input year of 10 would be taken as 2010.

%D

The %D utility displays the current date using the [d]d-mmm-[yy]yy format. If a routine uses this function repetitively, put the utility code directly into the M program.

Utility Labels

INT: Sets variable %DAT to current date.

FUNC[()]: Invokes an extrinsic function returning today's date.

Output Variables

%DAT: Contains the current date..

Examples of %D

For the following examples, $ZDATEFORM is assumed to be one (1).

Example:

```
GTM>DO ^%D
22-NOV-2010
```

This example invokes %D in Direct Mode. Then %D displays the current date.

Example:

```
GTM>DO INT^%D
GTM>ZWRITE
%DAT="22-NOV-2010"
```

This example invokes %D with the label INT (INT^%D). The variable %DAT contains the current date. ZWRITE displays the contents of the output variable.

Example:

```
GTM>WRITE $$FUNC^%D
22-NOV-2010
```

This example invokes %D as an extrinsic function with the label FUNC. $$FUNC^%D returns today's date.

%DATE

The %DATE utility converts an input date to the $HOROLOG format. The $HOROLOG format represents time as the number of days since December 31, 1840. The routine has entry points for interactive or non-interactive use.

Utility Labels

INT: Converts %DS input non-interactively, if defined, otherwise the current date.

FUNC(t): Invokes an extrinsic function returning $HOROLOG format of the argument.

Prompts

Date: Interactively requests a date for conversion to $HOROLOG format.

Input Variables

%DS: Contains input date; refer to %DATE Input Formats table.

Output Variables

%DN: Contains output date in $HOROLOG format

Date Input Formats Table

%DATE Input Formats		
ELEMENT	DESCRIPTION	EXAMPLES
DAYS	1 or 2 digits	1,01,24
MONTHS	1 or 2 digits	3,03,12
	Abbreviations accepted	MAR
	Numeric months precede days	1/5 is 5 Jan
	Alpha months may precede or follow days	3 MAR MAR 3
YEARS	2 or 4 digits	11/22/98 11/22/2002
	A missing year defaults to current year	11/22
TODAY	Abbreviation accepted	T[ODAY]
	t+/- N. no. of days	t+1 t-3
TOMORROW	Abbreviation accepted	TOM[ORROW]
YESTERDAY	Abbreviation accepted	Y[ESTERDAY]
NULL INPUT	Defaults to today	
DELIMITERS	All non-alphanumeric character(s) except the + or - offset	11/22/98 11 Nov 98 22 Nov, 2002 11-22-2002

Examples of %DATE

Example:

```
GTM>DO ^%DATE
Date:
GTM>ZWRITE
```

```
%DN=62047
```

This example invokes %DATE at the GTM> prompt. After pressing <RETURN> at the Date: prompt, %DATE converts today's date (for example, 11/22/2010) to the $HOROLOG format. ZWRITE displays the contents of the output variable.

Example:

```
GTM>DO INT^%DATE
GTM>ZWRITE
%DN=59105
```

This example invokes INT^%DATE, which converts the current date non-interactively into $HOROLOG format. ZWRITE displays the contents of the output variable.

Example:

```
GTM>SET %DS="10/20/2010"
GTM>DO INT^%DATE
GTM>ZWRITE
%DN=62019
%DS="10/20/2010"
```

This example sets the input variable %DS prior to invoking INT^%DATE, which converts that date non-interactively to $HOROLOG format.

Example:

```
GTM>WRITE $$FUNC^%DATE("10/20/2010")
62010
```

This example invokes %DATE with the label FUNC as an extrinsic function to convert an input date to $HOROLOG. If the invocation does not supply a date for $$FUNC^%DATE, FUNC converts the current date.

Example:

```
GTM>WRITE $ZDATEFORM
1975
GTM>WRITE $$FUNC^%DATE("10/20/80")
51062
GTM>WRITE $ZDATE(51062)
10/20/1980
GTM>WRITE $$FUNC^%DATE("10/20/10")
62019
GTM>WRITE $ZDATE(62019)
10/20/2010
```

This example shows the use of a year limit in $ZDATEFORM. Two digit years are interpreted to be in the interval (1975, 2074) since $ZDATEFORM is 1975; the input year "80" is interpreted as the year "1980" and "10" is interpreted as the year "2010". The example invokes FUNC^%DATE to convert the input date to $HOROLOG format. $ZDATE() is used to convert the $HOROLOG format date to mm/dd/yyyy format.

%H

The %H utility converts date and time to and from $HOROLOG format.

Utility Labels

%CDS: Converts %DT $HOROLOG input date to mm/dd/yyyy format.

%CTS: Converts %TM $HOROLOG input time to external format.

%CDN: Converts %DT input date to $HOROLOG format.

%CTN: Converts %TM input time to $HOROLOG format.

CDS(dt): Extrinsic entry that converts the $HOROLOG argument to external date format.

CTS(tm): Extrinsic entry that converts the $HOROLOG argument to external time format.

CDN(dt): Extrinsic entry that converts the argument to $HOROLOG format.

CTN(tm): Extrinsic entry that converts the argument to $HOROLOG format.

Input Variables

%DT: Contains input date in either $HOROLOG or mm/dd/[yy]yy format, depending on the format expected by the utility entry point.

%TM: Contains input time in either $HOROLOG or [h]h:mm:ss format, depending on the format expected by the utility entry point.

Output Variables

%DAT: Contains converted output date,

%TIM: Contains converted output time,

Examples of %H

Example:

```
GTM>SET %DT=+$H DO %CDS^%H
GTM>ZWRITE
%DAT="10/20/2010"
%DT=62047
```

This example sets %DT to the current date in $HOROLOG format and converts it to mm/dd/yyyy format by invoking %H at the label %CDS. %H returns the converted date in the variable %DAT. ZWRITE displays the contents of the variables.

Example:

```
GTM>SET %DT="10/20/2002" DO %CDN^%H
GTM>ZWRITE
%DAT=59097
%DT="10/20/2002"
```

This example sets the variable %DT to a date in mm/dd/yyyy format and invokes %H at the label %CDN. %H returns the converted date in the variable %DAT. ZWRITE displays the contents of the variables.

Example:

```
GTM>SET %TM=$P($H,",",2) DO %CTS^%H
GTM>ZWRITE
%TIM="17:41:18"
%TM=63678
```

This example sets the variable %TM to the current time in $HOROLOG format using a $PIECE() function to return only those digits of the $HOROLOG string that represent the time. The example then invokes %H at the label %CTS. %H returns the converted time in the variable %TIM. ZWRITE displays the contents of the variables.

Example:

```
GTM>SET %TM="17:41:18" DO %CTN^%H
GTM>ZWRITE
%TIM=63678
%TM="17:41:18"
```

This example sets the variable %TM to a time in hh:mm:ss format, and invokes %H at the label %CTN. %H returns the converted time in the variable %TIM. ZWRITE displays the contents of the variables.

Example:

```
GTM>WRITE $$CDS^%H(62019)
11/17/2010
```

This invokes CDS^%H as an extrinsic function to convert the external argument to external date format.

Example:

```
GTM>WRITE $ZDATEFORM
1980
GTM>WRITE $$CDN^%H("10/20/02")
59097
GTM>WRITE $ZDATE(59097)
10/20/2002
GTM>WRITE $$CDN^%H("10/20/92")
55445
GTM>WRITE $ZDATE(55445)
10/20/1992
```

This example shows the use of a year limit in $ZDATEFORM. Two digit years are interpreted to be in the interval of 1980 - 2079; since $ZDATEFORM is 1980, the input year "02" is interpreted as "2002" and "92" is interpreted as "1992". This example invokes CDN^%H to convert the argument in mm/dd/yy format to $HOROLOG format. $ZDATE() is used to conver the $HOROLOG format date to mm/dd/yyyy format.

%T

The %T utility displays the current time in [h]h:mm AM/PM. If a routine uses this function repetitively, put the utility code directly into the M program.

Utility Labels

INT: Sets %TIM to current time in [h]h:mm AM/PM format.

FUNC[()]: Invokes an extrinsic function returning the current time.

Output Variables

%TIM: Contains current time in [h]h:mm AM/PM format.

Examples of %T

Example:

```
GTM>DO ^%T
8:30 AM
```

This example invokes %T, which prints the current time and does not set %TIM.

Example:

```
GTM>DO INT^%T
GTM>ZWRITE
%TIM="8:30 AM"
```

This example invokes INT^%T, which sets the variable %TIM to the current time. ZWRITE displays the contents of the variable.

Example:

```
GTM>WRITE $$FUNC^%T
8:30 AM
```

This example invokes FUNC as an extrinsic function, which returns the current time.

%TI

The %TI utility converts time to $HOROLOG format. The $HOROLOG format represents time as the number of seconds since midnight. %TI returns the converted time in the variable %TN. The routine has entry points for interactive or non-interactive use.

Utility Labels

INTNon-interactively converts %TS to $HOROLOG format; if %TS is not defined, then current time is converted.

FUNC[(ts)]Invokes an extrinsic function returning $HOROLOG format of the argument, or if no argument, the $HOROLOG format of the current time.

Prompts

Time: Requests time in [h]h:mm:ss format to convert to $HOROLOG format.

Input Variables

%TS Contains input time.

The following table summarizes input formats accepted by %TI.

%TI Input Formats		
ELEMENT	DESCRIPTION	EXAMPLES
HOURS	1 or 2 digits	3,03,12
MINUTES	2 digits	05,36
AM or PM	AM or PM required	9:00 AM or am 9:00 PM or pm
	Abbreviation accepted	9:00 A or a 9:00 P or p
NOON	Abbreviation accepted	N[OON]
MIDNIGHT or MIDNITE	Abbreviation accepted	M[IDNIGHT] or m[idnight] M[IDNITE] or m[idnite]
MILITARY	No punctuation (hhmm)	1900, 0830
NULL INPUT	Defaults to current time	
DELIMITERS	Colon between hours and minutes	3:00

Output Variables

%TN: Contains output time in $HOROLOG format

Examples of %TI

Example:

```
GTM>DO ^%TI
Time: 4:02 PM
GTM>ZWRITE
%TN=57720
```

This example invokes %TI, which prompts for an input time. Press <RETURN> to convert the current time. ZWRITE displays the contents of the output variable.

Example:

```
GTM>ZWRITE
GTM>DO INT^%TI
GTM>ZWRITE
%TN=40954
```

This example invokes INT^%TI to convert the current time non-interactively. ZWRITE displays the contents of the output variable %TN.

Example:

```
GTM>SET %TS="8:30AM"
```

```
GTM>DO INT^%TI
GTM>ZWRITE
%TN=30600
%TS="8:30AM"
```

This example sets the variable %TS prior to invoking INT^%TI. %TI uses %TS as the input time. ZWRITE displays the contents of the variables.

Example:

```
GTM>WRITE $$FUNC^%TI("8:30AM")
30600
```

This example invokes %TI as an extrinsic function to convert the supplied time to $HOROLOG format. If there is no argument (i.e., $$FUNC^%TI), %TI converts the current time.

%TO

The %TO utility converts the input time from $HOROLOG format to [h]h:mm AM/PM format. Put the utility code directly into the M program if the routine uses this function repetitively.

Utility Labels

INT: Converts non-interactively %TS, or if %TS is not defined the current time to [h]h:mm AM/PM format.

Input Variables

%TN: Contains input time in $HOROLOG format.

Output Variables

%TS: Contains output time in [h]h:mm AM/PM format.

Examples of %TO

Example:

```
GTM>DO INT^%TI,^%TO
GTM>ZWRITE
%TN=62074
%TS="5:14 PM"
```

This example invokes INT^%TI to set %TN to the current time and invokes %TO to convert the time contained in %TN to the [h]h:mm AM/PM format. %TO returns the converted time in the variable %TS. ZWRITE displays the contents of the variables.

Conversion Utilities

The conversion utilities are:

%DH: Decimal to hexadecimal conversion.

%DO: Decimal to octal conversion.

%HD: Hexadecimal to decimal conversion.

%HO: Hexadecimal to octal conversion.

%LCASE: Converts a string to all lower case.

%OD: Octal to decimal conversion.

%OH: Octal to hexadecimal conversion.

%UCASE: Converts a string to all upper case.

The conversion utilities can be invoked as extrinsic functions.

While the radix conversions work for very long values, the performance degrades with length and may be impractical beyond some point. If your application limits the input(s) and performance is important, you can get some speed improvement by eliminating behavior on which your code does not rely.

%DH

The %DH utility converts numeric values from decimal to hexadecimal. %DH defaults the length of its output to eight digits. However the input variable %DL overrides the default and controls the length of the output. The routine has entry points for interactive or non-interactive use.

Utility Labels

INT: Converts interactively entered decimal number to hexadecimal number with the number of digits specified.

FUNC(d[,l]): Invokes %DH as an extrinsic function returning the hexadecimal equivalent of the argument.

Input Variables

%DH: As input, contains input decimal number.

%DL: Specifies how many digits appear in the output, defaults to eight.

Prompts

Decimal: Requests a decimal number for conversion to hexadecimal.

Digits: Requests the length of the output in digits; eight by default.

Output Variables

%DH: As output, contains the converted number in hexadecimal.

Examples of %DH

Example:

```
GTM>DO INT^%DH
```

```
Decimal: 12
Digits: 1
GTM>ZWRITE
%DH="C"
```

This example invokes %DH interactively with INT^%DH. %DH prompts for a decimal number and output length, then returns the result in the variable %DH. ZWRITE displays the contents of the variables.

Example:

```
GTM>SET %DH=12
GTM>DO ^%DH
GTM>ZWRITE
%DH="0000000C"
%DL=8
```

This example sets the read-write variable %DH to 12 and invokes %DH to convert the number to a hexadecimal number. Because the number of digits was not specified, %DH used the default of 8 digits. Set %DL to specify the number of output digits.

Example:

```
GTM>WRITE $$FUNC^%DH(12,4)
000C
```

This example invokes %DH as an extrinsic function using the FUNC label. The first argument specifies the input decimal number and the optional, second argument specifies the number of output digits. If the extrinsic does not have a second argument, the length of the output defaults to eight characters.

%DO

The %DO utility converts numeric values from decimal to octal. The default length of its output is 12 digits. The value assigned to the input variable %DL overrides the default and controls the length of the output. The routine has entry points for interactive or non-interactive use.

Utility Labels

INT: Converts the specified decimal number to an octal number with the specified number of digits, interactively.

FUNC(d[,ln]): Invokes %DO as an extrinsic function, returning the octal equivalent of the argument.

Prompts

Decimal: Requests a decimal number for conversion to octal.

Digits: Requests the length of the output in digits; 12 by default.

Input Variables

%DO: As input, contains input decimal number.

%DL: Specifies the number of digits in the output, defaults to 12.

Output Variables

%DO: As output, contains the converted number in octal.

Examples of %DO

Example:

```
GTM>DO INT^%DO
Decimal: 12
Digits: 4
GTM>ZWRITE
%DO="0014"
```

This example invokes %DO interactively with INT^%DO. %DO prompts for a decimal number and an output length. If the output value of %DO has leading zeros, the value is a string. ZWRITE displays the contents of the variables.

Example:

```
GTM>SET %DO=12
GTM>DO ^%DO
GTM>ZWRITE
%DO="000000000014"
```

This example sets the read-write variable %DO to 12 and invokes %DO to convert the number non-interactively. Because the number of digits was not specified, %DO used the default of 12 digits. Set %DL to specify the number of output digits. ZWRITE displays the contents of the variables.

Example:

```
GTM>WRITE $$FUNC^%DO(12,7)
0000014
```

This example invokes %DO as an extrinsic function with the label FUNC. The first argument specifies the number to be converted and the optional, second argument specifies the number of output digits. If the second argument is not specified, %DO uses the default of 12 digits.

%HD

The %HD utility converts numeric values from hexadecimal to decimal. %HD returns the decimal number in the read-write variable %HD. %HD rejects input numbers beginning with a minus (-) sign and returns null (""). The routine has entry points for interactive or non-interactive use.

Utility Labels

INT: Converts hexadecimal number entered interactively to decimal number.

FUNC(h): Invokes %HD as an extrinsic function returning the decimal equivalent of the argument.

Prompts

Hexadecimal: Requests a hexadecimal number for conversion to decimal.

Input Variables

%HD: As input, contains input hexadecimal number.

Output Variables

%HD: As output, contains the converted number in decimal.

Examples of %HD

Example:

```
GTM>DO INT^%HD
Hexadecimal:E
GTM>ZWRITE
%HD=14
```

This example invokes %HD in interactive mode with INT^%HD. %HD prompts for a hexadecimal number, then returns the converted number in the variable %HD. ZWRITE displays the contents of the variable.

Example:

```
GTM>SET %HD="E"
GTM>DO ^%HD
GTM>ZWRITE
%HD=14
```

This example sets the read-write variable %HD to "E" and invokes %HD to convert non-interactively the value of %HD to a decimal number. %HD places the converted value into the read-write variable %HD.

Example:

```
GTM>WRITE $$FUNC^%HD("E")
14
```

This example invokes %HD as an extrinsic function with the label FUNC and writes the results.

%HO

The %HO utility converts numeric values from hexadecimal to octal. %HO returns the octal number in the read-write variable %HO. %HO rejects input numbers beginning with a minus (-) sign and returns null (""). The routine has entry points for interactive or non-interactive use.

Utility Labels

INT: Converts hexadecimal number entered interactively to octal number.

FUNC(h): Invokes %HO as an extrinsic function returning the octal equivalent of the argument.

Prompts

Hexadecimal: Requests a hexadecimal number for conversion to octal.

Input Variables

%HO: As input, contains input hexadecimal number.

Output Variables

%HO: As output, contains the converted number in octal.

Examples of %HO

Example:

```
GTM>DO INT^%HO
Hexadecimal:C3
GTM>ZWRITE
%HO=303
```

This example invokes %HO in interactive mode using INT^%HO. %HO prompts for a hexadecimal number that it converts to an octal number. ZWRITE displays the contents of the variable.

Example:

```
GTM>SET %HO="C3"
GTM>DO ^%HO
GTM>ZWRITE
%HO=303
```

This example sets the read-write variable %HO to "C3" and invokes %HO to convert the value of %HO non-interactively. ZWRITE displays the contents of the variable.

Example:

```
GTM>WRITE $$FUNC^%HO("C3")
303
```

This example invokes %HO as an extrinsic function with the FUNC label.

%JSWRITE

The ^%JSWRITE utility routine converts a glvn structure or a series of SET @ arguments to a string of JS objects. The format of the ^%JSWRITE utility is:

```
^%JSWRITE(glvnode,[expr1,expr2])
```

- glvnode specifies the string containing the subscripted/unsubscripted global or local variable name. When glvnode evaluates to an empty string ("") or there are no arguments, %JSWRITE considers all subscripted local variables in scope for conversion.

- If expr1 specifies "#", ^%JSWRITE displays JS objects of the entire tree starting from the glvnode till the end of the glvn.

- If expr1 specifies "*", ^%JSWRITE displays JS objects for all nodes descending from the specified glvn node.

- Specifying "*" and "#" together produces an ILLEGALEXPR2 error.

- Specifying [expr1], that is, with a leading "[" and trailing "]", ^%JSWRITE displays the JSON objects in an array collection. Without [], you need to transform the object strings to the desired destination object format.

- If expr2 specifies "noglvname", ^%JSWRITE excludes the first key containing the name of the glvn root from the JS object output.

- The default $ETRAP for %JSWRITE is **if (""=$etrap) new $etrap set $etrap="do errorobj"_"^"_$text(+0),$zstatus=""**. To override the default error handler, set $ETRAP to any non-empty value.

- GT.M is not a JavaScript runtime environment. Therefore, we recommend parsing all output of ^%JSWRITE either using a JSON parser such as JSON.parse() in an appropriate JavaScript run-time environment, a web server via setting its response header to **'Content-Type:application/json'**, or an application where JSON parsing is available.

- When appropriate, enclose invocations of ^%JSWRITE in a TSTART/COMMIT boundary to prevent any blurred copy of data that is actively updated.

- When appropriate, use GT.M alias containers to take appropriate local variables temporarily out of scope and then run the argumentless form of ^%JSWRITE.

Examples:

```
# Demo lv
GTM>zwrite demodevtest
demodevtest("Developer1","Token1","testSetup")="runtest holt maintest cpipe"
demodevtest("Developer1","Token1","testSetup","65401,11987")=1
demodevtest("Developer1","Token1","testSetup","holt","t")="mtest"
demodevtest("Developer1","Token1","testSetup","holt","t","SendReport",65401,12073)=1
demodevtest("Developer1","Token1","testSetup","holt","t","cpipe",65401,12025)=0
demodevtest("Developer1","Token2","testSetup")="runtest holt maintest tconv"
demodevtest("Developer1","Token2","testSetup","holt","65401,21987")=1
demodevtest("Developer1","Token2","testSetup","holt","t")="mtest"
demodevtest("Developer1","Token2","testSetup","holt","t","SendReport",65401,22073)=1
demodevtest("Developer1","Token2","testSetup","holt","t","tconv",65401,22025)=0
demodevtest("Developer2","Token3","testSetup")="runtest holt maintest tconv"
demodevtest("Developer2","Token3","testSetup","holt","65401,21987")=1
demodevtest("Developer2","Token3","testSetup","holt","t")="mtest"
demodevtest("Developer2","Token3","testSetup","holt","t","SendReport",65401,22073)=1
demodevtest("Developer2","Token3","testSetup","holt","t","tconv",65401,22025)=0
demodevtest("Developer3","Token4","testSetup")="runtest holt maintest tconv"
demodevtest("Developer3","Token4","testSetup","holt","65401,31987")=1
demodevtest("Developer3","Token4","testSetup","holt","t")="mtest"
demodevtest("Developer3","Token4","testSetup","holt","t","SendReport",65401,32073)=1
demodevtest("Developer3","Token4","testSetup","holt","t","tconv",65401,32025)=0

GTM>set glvn="demodevtest(""Developer2"")"

GTM>do ^%JSWRITE(glvn,"*") ; JS Object Strings: All descendants of
▶ demodevtest("Developer2")
{"demodevtest":{"Developer2":{"Token3":{"testSetup":"runtest holt maintest tconv"}}}}
{"demodevtest":{"Developer2":{"Token3":{"testSetup":{"holt":{"65401,21987":1}}}}}}
{"demodevtest":{"Developer2":{"Token3":{"testSetup":{"holt":{"t":"mtest"}}}}}}
{"demodevtest":{"Developer2":{"Token3":{"testSetup":{"holt":{"t":{"SendReport":{"65401":{"22073":1}}}}}}}}}
{"demodevtest":{"Developer2":{"Token3":{"testSetup":{"holt":{"t":{"tconv":{"65401":{"22025":0}}}}}}}}}

GTM>do ^%JSWRITE(glvn,"[*]") ; Array: All descendants of demodevtest("Developer2")
[{"demodevtest":{"Developer2":{"Token3":{"testSetup":"runtest holt maintest tconv"}}}},
{"demodevtest":{"Developer2":{"Token3":{"testSetup":{"holt":{"65401,21987":1}}}}}},
{"demodevtest":{"Developer2":{"Token3":{"testSetup":{"holt":{"t":"mtest"}}}}}},
```

```
{"demodevtest":{"Developer2":{"Token3":{"testSetup":{"holt":{"t":{"SendReport":{"65401":{"22073":1}}}}}}}},
{"demodevtest":{"Developer2":{"Token3":{"testSetup":{"holt":{"t":{"tconv":{"65401":{"22025":0}}}}}}}}]

GTM> do ^%JSWRITE(glvn,"#") ; JS Object Strings: All descendants of demodevtest starting from
▸ demodevtest("Developer2")
{"demodevtest":{"Developer2":{"Token3":{"testSetup":"runtest holt maintest tconv"}}}}
{"demodevtest":{"Developer2":{"Token3":{"testSetup":{"holt":{"65401,21987":1}}}}}}
{"demodevtest":{"Developer2":{"Token3":{"testSetup":{"holt":{"t":"mtest"}}}}}}
{"demodevtest":{"Developer2":{"Token3":{"testSetup":{"holt":{"t":{"SendReport":{"65401":{"22073":1}}}}}}}}
{"demodevtest":{"Developer2":{"Token3":{"testSetup":{"holt":{"t":{"tconv":{"65401":{"22025":0}}}}}}}}
{"demodevtest":{"Developer3":{"Token4":{"testSetup":"runtest holt maintest tconv"}}}}
{"demodevtest":{"Developer3":{"Token4":{"testSetup":{"holt":{"65401,31987":1}}}}}}
{"demodevtest":{"Developer3":{"Token4":{"testSetup":{"holt":{"t":"mtest"}}}}}}
{"demodevtest":{"Developer3":{"Token4":{"testSetup":{"holt":{"t":{"SendReport":{"65401":{"32073":1}}}}}}}}
{"demodevtest":{"Developer3":{"Token4":{"testSetup":{"holt":{"t":{"tconv":{"65401":{"32025":0}}}}}}}}

GTM>do ^%JSWRITE(glvn,"[#]") ; Array: All descendants of demodevtest starting from
▸ demodevtest("Developer2")
[{"demodevtest":{"Developer2":{"Token3":{"testSetup":"runtest holt maintest tconv"}}}},
{"demodevtest":{"Developer2":{"Token3":{"testSetup":{"holt":{"65401,21987":1}}}}}},
{"demodevtest":{"Developer2":{"Token3":{"testSetup":{"holt":{"t":"mtest"}}}}}},
{"demodevtest":{"Developer2":{"Token3":{"testSetup":{"holt":{"t":{"SendReport":{"65401":{"22073":1}}}}}}}},
{"demodevtest":{"Developer2":{"Token3":{"testSetup":{"holt":{"t":{"tconv":{"65401":{"22025":0}}}}}}}},
{"demodevtest":{"Developer3":{"Token4":{"testSetup":"runtest holt maintest tconv"}}}},
{"demodevtest":{"Developer3":{"Token4":{"testSetup":{"holt":{"65401,31987":1}}}}}},
{"demodevtest":{"Developer3":{"Token4":{"testSetup":{"holt":{"t":"mtest"}}}}}},
{"demodevtest":{"Developer3":{"Token4":{"testSetup":{"holt":{"t":{"SendReport":{"65401":{"32073":1}}}}}}}},
{"demodevtest":{"Developer3":{"Token4":{"testSetup":{"holt":{"t":{"tconv":{"65401":{"32025":0}}}}}}}}]

$ $gtm_dist/mumps -r %XCMD 'ZWRITE ^demodevtest' | $gtm_dist/mumps -r STDIN^%JSWRITE
[{"^demodevtest":{"Developer1":{"Token1":{"testSetup":"runtest holt maintest cpipe"}}}},
{"^demodevtest":{"Developer1":{"Token1":{"testSetup":{"65401,11987":1}}}}},
{"^demodevtest":{"Developer1":{"Token1":{"testSetup":{"holt":{"t":"mtest"}}}}}},
{"^demodevtest":{"Developer1":{"Token1":{"testSetup":{"holt":{"t":{"SendReport":{"65401":{"12073":1}}}}}}}},
{"^demodevtest":{"Developer1":{"Token1":{"testSetup":{"holt":{"t":{"cpipe":{"65401":{"12025":0}}}}}}}},
{"^demodevtest":{"Developer1":{"Token2":{"testSetup":"runtest holt maintest tconv"}}}},
{"^demodevtest":{"Developer1":{"Token2":{"testSetup":{"65401,21987":1}}}}},
{"^demodevtest":{"Developer1":{"Token2":{"testSetup":{"holt":{"t":"mtest"}}}}}},
{"^demodevtest":{"Developer1":{"Token2":{"testSetup":{"holt":{"t":{"SendReport":{"65401":{"22073":1}}}}}}}},
{"^demodevtest":{"Developer1":{"Token2":{"testSetup":{"holt":{"t":{"tconv":{"65401":{"22025":0}}}}}}}},
{"^demodevtest":{"Developer2":{"Token3":{"testSetup":"runtest holt maintest tconv"}}}},
{"^demodevtest":{"Developer2":{"Token3":{"testSetup":{"65401,21987":1}}}}},
{"^demodevtest":{"Developer2":{"Token3":{"testSetup":{"holt":{"t":"mtest"}}}}}},
{"^demodevtest":{"Developer2":{"Token3":{"testSetup":{"holt":{"t":{"SendReport":{"65401":{"22073":1}}}}}}}},
{"^demodevtest":{"Developer2":{"Token3":{"testSetup":{"holt":{"t":{"tconv":{"65401":{"22025":0}}}}}}}},
{"^demodevtest":{"Developer3":{"Token4":{"testSetup":"runtest holt maintest tconv"}}}},
{"^demodevtest":{"Developer3":{"Token4":{"testSetup":{"65401,31987":1}}}}},
{"^demodevtest":{"Developer3":{"Token4":{"testSetup":{"holt":{"t":"mtest"}}}}}},
{"^demodevtest":{"Developer3":{"Token4":{"testSetup":{"holt":{"t":{"SendReport":{"65401":{"32073":1}}}}}}}},

▸ {"^demodevtest":{"Developer3":{"Token4":{"testSetup":{"holt":{"t":{"tconv":{"65401":{"32025":0}}}}}}}}]
```

▼

Utility Label:

STDIN^%JSWRITE [singlesub]

With the STDIN, the %JSWRITE utility routine expects a valid SET @ argument (like the one from the ZWRITE command) as its standard input over a named/unnamed pipe device and returns an array of objects. This construct ensures that $ZUSEDSTOR remains consistently low even for processing large data for conversion. STDIN^%JSWRITE automatically terminates the process with a non-zero exit status when it does not receive a READ terminator for 120 seconds from standard input.

When "singlesub" is specified as an argument, ^%JSWRITE expects ZWRITE lines for single subscript glvns. Here ^%JSWRITE implicitly removes the unsubscripted glvn name and returns an array collection of objects in the form of **[{"key1":value,"key2":value,...},{"key1":value,"key2":value,...}]** where:

- **key1** is the subscript

- **value** is the right side of the =.

The subscript first received by STDIN^%JSWRITE singlesub denotes the start of the object. When ^%JSWRITE finds the same subscript, it ends the current object boundary and starts the boundary of a new object.

Example:

```
$ $gtm_dist/mumps -r ^RTN
abc("firstname")="John"
abc("lastname")="Doe"
abc("firstname")="Jane"
abc("lastname")="Doe"

$ $gtm_dist/mumps -r ^RTN | $gtm_dist/mumps -r STDIN^%JSWRITE singlesub
[{"firstname":"John","lastname":"Doe"},
{"firstname":"Jane","lastname":"Doe"}]
```

The fis-gtm-jswrite.tar.gz npm package

Overview

fis-gtm-jswrite.tar.gz is an npm package containing four ^%JSWRITE reference implementations and three examples for the ^%JSWRITE utility routine. It also includes a utility class called JSWRITE.js to help process the output of the ^%JSWRITE utility routine for use in a JavaScript runtime environment and convert JavaScript objects to SET @ arguments.

The fis-gtm-jswrite.tar.gz package is available only for supported GT.M customers from the FIS Client Portal (https://my.fisglobal.com/products/gtm). Being a supported GT.M customer entitles you to receive package patches and upgrades under the terms of your support agreement. Note that the license for the fis-gtm-jswrite.tar.gz does not permit redistribution. You may use the fis-gtm-jswrite.tar.gz package as-is or modify as appropriate to suit your needs. In both the cases, you must adequately test the reference implementations and the utility class before using them in a production environment. Please contact gtmsupport@fisglobal.com or your support channel for more information on obtaining this package.

The ^%JSWRITE reference implementations are:

1. Dynamic Journal File Progress Bar: GTMJSJNLCHAIN.m/gtmjsjnlchain.js

2. Global Buffers Dashboard: GTMJSACCESSBG.m/gtmjsaccessbg.js

3. Journal File Chain Report: GTMJSJNLCHAIN.m/gtmjsdbjnl.js

4. %YGBLSTATS Sparkline Chart and %YGBLSTATS to JSON: GTMJSGVSTATS.m/gtmjsgvstats.js

---

The examples are:

1. gtmjstree.js (renders ^sampletree in the form of a tree)

2. gtmjs-sql1.js (a simple example)

3. gtmjs-sql2.js (a simple example)

4. gtmjs-to-gtm.js (example of JSWRITE.js helper class)

Installation

Step 1: Install node.js and npm

```
curl -sL https://deb.nodesource.com/setup_13.x | sudo -E bash -
sudo apt-get install -y nodejs
curl -L https://www.npmjs.com/install.sh | sh
```

Step 2: Install the fis-gtm-jswrite npm package

```
cd $project_dir
npm install /path/to/fis-gtm-jswrite.tar.gz # This command installs the @fis-gtm/jswrite package and the relevant
▶ %JSWRITE reference implementation dependencies.
npm audit --fix
```

🔽

Step 3: Run the %JSWRITE reference implementations

```
# The following reference implementations use the "STDIN^%JSWRITE singlesub" entry point and require the gtmposix
▶ plugin.
export gtmroutines="node_modules/@fis-gtm/jswrite/reference_implementations $gtmroutines"      # include the
▶ reference_implementations directory in the search list
node node_modules/@fis-gtm/jswrite/reference_implementations/gtmjsjnlchain.js  # journal file chain status sheet
node node_modules/@fis-gtm/jswrite/reference_implementations/gtmjsaccessbg.js  # %dirty buffer dashboard
node node_modules/@fis-gtm/jswrite/reference_implementations/gtmjsdbjnl.js     # displays the size of the database
 and
▶ journal files
# Set the environment variable LC_ALL to a UTF8 locale (needed for the %JSWRITE Sparkline Chart
▶ (GTMJSGVSTATS.m/gtmjsgvstats.js) reference implementation)
node node_modules/@fis-gtm/jswrite/reference_implementations/gtmjsgvstats.js   # By default, this reference
▶ implementation displays a sparkline chart for SETs and GETs. For monitoring other statistics, open
 gtmjsgvstats.js and update the STAT array to include the list of statistics which require sparkline chart style
 monitoring.
```

🔽

Step 4: Run the examples

Create a sandbox environment for GT.M and run the following commands:

```
$gtm_dist/mupip set -key_size=1019 -region "*"
$gtm_dist/mupip load node_modules/@fis-gtm/jswrite/examples/sample.zwr
$gtm_dist/mupip load node_modules/@fis-gtm/jswrite/examples/sampletree.zwr
export "gtmroutines=node_modules/@fis-gtm/jswrite/examples/ $gtmroutines"
```

```
# The following example uses the do ^%JSWRITE("^sample","[*]") entry point. This uses the lodash library to merge
 the
▶ nodes coming from %JSWRITE.
npm install lodash@latest
node node_modules/@fis-gtm/jswrite/examples/gtmjstree.js     # loads sample.zwr in the tree format
# The following two examples use the STDIN^%JSWRITE entry point of the %JSWRITE utility routine and then use the
 getter
▶ methods of the JSWRITE.js helper class to create the SQL queries.
npm install alasql@latest
node node_modules/@fis-gtm/jswrite/examples/gtmjs-sql1.js
npm install sqlite3@latest
node node_modules/@fis-gtm/jswrite/examples/gtmjs-sql2.js
# The following example uses the @fis-gtm/jswrite helper class (JSWRITE.js) to send data back to GT.M in SET @
 argument
▶ form using shelljs.
npm install shelljs@latest
node node_modules/@fis-gtm/jswrite/examples/gtmjs-to-gtm.js
```

▼

Dynamic Journal File Progress Bar

The dynamic journal file progress bar displays the size progress of a journal file till it reaches the autoswitch limit. It also displays an estimation of time and speed at which a journal file reaches its autoswitch limit. This can be use for benchmarking purposes or as a tool for GT.M performance tuning.

This %JSWRITE reference implementation uses the STDIN^%JSWRITE singlesub entrypoint of the %JSWRITE utility routine.

Example:

```
$ node node_modules/@fis-gtm/jswrite/reference_implementations/gtmjsjnlprogress.js
Press <Ctrl> + c to terminate. refreshInterval is 3 seconds
DEFAULT [===------------------------------------] 7% | Size: 303.7 MiB | Autoswitch: 4095 MiB | ETA: 5m40s |
 Speed:
▶ 15.233 MiB/s
REGIONA [========-----------------------------] 21% | Size: 876 MiB | Autoswitch: 4095 MiB | ETA: 5m15s | Speed:
 14
▶ MiB/s
REGIONB [========-----------------------------] 23% | Size: 962 MiB | Autoswitch: 4095 MiB | ETA: 4m40s | Speed:
 15
▶ MiB/s
REGIONC [========-----------------------------] 21% | Size: 876.12 MiB | Autoswitch: 4095 MiB | ETA: 5m15s |
 Speed:
▶ 14.04 MiB/s
```

▼

This %JSWRITE reference implementation monitors the size of the journal files and displays an estimate of time and speed for reaching the autoswitch limit.

Journal File Chain Report

The journal file chain report display the journal file chain starting from the current journal file for reach region. This reference implementation also displays the "out-of-chain" journal files, that is, those journal files which are broken and do not participate

in journal recovery in normal circumstances. These out-of-chain journal files can be removed at the discretion of the GT.M database administrator to make space available.

This %JSWRITE reference implementation uses the STDIN^%JSWRITE singlesub entrypoint of the %JSWRITE utility routine.

Example:

```
$ node node_modules/@fis-gtm/jswrite/reference_implementations/gtmjsjnlchain.js
# Journal File Chain Report
| Region    | Journal File                        | Size (MiB) | Switch Date          | Back link

▶ | Chain status |
| :--------- | :-------------------------------- | ---------: | :------------------
 | :-------------------------
▶ | ------------ |
| GTMWIZARDS | /path/to/gtmwizards.mjl             |       0.07 | current              |
 gtmwizards.mjl_2020063121042
▶ | In chain   |
| GTMWIZARDS | /path/to/gtmwizards.mjl_2020063121042 |     0.07 | 03-MAR-2020 12:10:42 |
 gtmwizards.mjl_2020063121035
▶ | In chain   |
| GTMWIZARDS | /path/to/gtmwizards.mjl_2020063121035 |     0.07 | 03-MAR-2020 12:10:35 | -

▶ | In chain   |
| REGIONA    | /path/to/regiona.mjl                |       0.07 | current              |
 regiona.mjl_2020063121042
▶ | In chain   |
| REGIONA    | /path/to/regiona.mjl_2020063121042  |       0.07 | 03-MAR-2020 12:10:42 |
 regiona.mjl_2020063121035
▶ | In chain   |
| REGIONA    | /path/to/regiona.mjl_2020063121035  |       0.07 | 03-MAR-2020 12:10:35 | -

▶ | In chain   |
| REGIONB    | /path/to/regionb.mjl                |       0.07 | current              |
 regionb.mjl_2020063121042
▶ | In chain   |
| REGIONB    | /path/to/regionb.mjl_2020063121042  |       0.07 | 03-MAR-2020 12:10:42 |
 regionb.mjl_2020063121035
▶ | In chain   |
| REGIONB    | /path/to/regionb.mjl_2020063121035  |       0.07 | 03-MAR-2020 12:10:35 | -

▶ | In chain   |
| REGIONC    | /path/to/regionc.mjl                |       0.07 | current              |
 regionc.mjl_2020063121042
▶ | In chain   |
| REGIONC    | /path/to/regionc.mjl_2020063121042  |       0.07 | 03-MAR-2020 12:10:42 |
 regionc.mjl_2020063121035
▶ | In chain   |
| REGIONC    | /path/to/regionc.mjl_2020063121035  |       0.07 | 03-MAR-2020 12:10:35 | -

▶ | In chain   |
| N/A        | /path/to/gtmwizards.mjl_2020063120753 |     0.07 | 03-MAR-2020 12:07:53 | -

▶ | Out of chain |
| N/A        | /path/to/gtmwizards.mjl_2020063120800 |     0.07 | 03-MAR-2020 12:08:00 | -
```

```
▶ | Out of chain |
| N/A         | /path/to/gtmwizards.mjl_2020063120801 |     0.07 | 03-MAR-2020 12:08:01 |
gtmwizards.mjl_2020063120800
▶ | Out of chain |
| N/A         | /path/to/gtmwizards.mjl_2020063120812 |     0.07 | 03-MAR-2020 12:08:12 |
gtmwizards.mjl_2020063120801
▶ | Out of chain |
| N/A         | /path/to/gtmwizards.mjl_2020063120822 |     0.07 | 03-MAR-2020 12:08:22 |
gtmwizards.mjl_2020063120812
▶ | Out of chain |
| N/A         | /path/to/gtmwizards.mjl_2020063121026 |     0.07 | 03-MAR-2020 12:10:26 |
gtmwizards.mjl_2020063120822
▶ | Out of chain |
| N/A         | /path/to/regiona.mjl_2020063120753    |     0.07 | 03-MAR-2020 12:07:53 | -

▶ | Out of chain |
| N/A         | /path/to/regiona.mjl_2020063120800    |     0.07 | 03-MAR-2020 12:08:00 | -

▶ | Out of chain |
| N/A         | /path/to/regiona.mjl_2020063120801    |     0.07 | 03-MAR-2020 12:08:01 |
regiona.mjl_2020063120800
▶ | Out of chain |
| N/A         | /path/to/regiona.mjl_2020063120812    |     0.07 | 03-MAR-2020 12:08:12 |
regiona.mjl_2020063120801
▶ | Out of chain |
| N/A         | /path/to/regiona.mjl_2020063120822    |     0.07 | 03-MAR-2020 12:08:22 |
regiona.mjl_2020063120812
▶ | Out of chain |
| N/A         | /path/to/regiona.mjl_2020063121026    |     0.07 | 03-MAR-2020 12:10:26 |
regiona.mjl_2020063120822
▶ | Out of chain |
| N/A         | /path/to/regionb.mjl_2020063120753    |     0.07 | 03-MAR-2020 12:07:53 | -

▶ | Out of chain |
| N/A         | /path/to/regionb.mjl_2020063120800    |     0.07 | 03-MAR-2020 12:08:00 | -

▶ | Out of chain |
| N/A         | /path/to/regionb.mjl_2020063120801    |     0.07 | 03-MAR-2020 12:08:01 |
regionb.mjl_2020063120800
▶ | Out of chain |
| N/A         | /path/to/regionb.mjl_2020063120812    |     0.07 | 03-MAR-2020 12:08:12 |
regionb.mjl_2020063120801
▶ | Out of chain |
| N/A         | /path/to/regionb.mjl_2020063120822    |     0.07 | 03-MAR-2020 12:08:22 |
regionb.mjl_2020063120812
▶ | Out of chain |
| N/A         | /path/to/regionb.mjl_2020063121026    |     0.07 | 03-MAR-2020 12:10:26 |
regionb.mjl_2020063120822
▶ | Out of chain |
| N/A         | /path/to/regionc.mjl_2020063120753    |     0.07 | 03-MAR-2020 12:07:53 | -

▶ | Out of chain |
| N/A         | /path/to/regionc.mjl_2020063120800    |     0.07 | 03-MAR-2020 12:08:00 | -

▶ | Out of chain |
```

```
| N/A           | /path/to/regionc.mjl_2020063120801    |        0.07 | 03-MAR-2020 12:08:01 |
 regionc.mjl_2020063120800
▶ | Out of chain |
| N/A           | /path/to/regionc.mjl_2020063120812    |        0.07 | 03-MAR-2020 12:08:12 |
 regionc.mjl_2020063120801
▶ | Out of chain |
| N/A           | /path/to/regionc.mjl_2020063120822    |        0.07 | 03-MAR-2020 12:08:22 |
 regionc.mjl_2020063120812
▶ | Out of chain |
| N/A           | /path/to/regionc.mjl_2020063121026    |        0.07 | 03-MAR-2020 12:10:26 |
 regionc.mjl_2020063120822
▶ | Out of chain |
# Out of chain journal files:
 /path/to/gtmwizards.mjl_2020063120753 /path/to/gtmwizards.mjl_2020063120800 /path/to/gtmwizards.mjl_2020063120801
▶ /path/to/gtmwizards.mjl_2020063120812 /path/to/gtmwizards.mjl_2020063120822 /path/to/
gtmwizards.mjl_2020063121026 /path/to/regiona.mjl_2020063120753 /path/to/regiona.mjl_2020063120800 /path/
to/regiona.mjl_2020063120801 /path/to/regiona.mjl_2020063120812 /path/to/regiona.mjl_2020063120822 /path/
to/regiona.mjl_2020063121026 /path/to/regionb.mjl_2020063120753 /path/to/regionb.mjl_2020063120800 /path/
to/regionb.mjl_2020063120801 /path/to/regionb.mjl_2020063120812 /path/to/regionb.mjl_2020063120822 /path/
to/regionb.mjl_2020063121026 /path/to/regionc.mjl_2020063120753 /path/to/regionc.mjl_2020063120800 /path/
to/regionc.mjl_2020063120801 /path/to/regionc.mjl_2020063120812 /path/to/regionc.mjl_2020063120822 /path/to/
regionc.mjl_2020063121026
# Out of chain total       : 1.68 MiB
# Summary
| Region     | Size Total (MiB) | Recoverability up to |
| ---------- | ---------------- | -------------------- |
| GTMWIZARDS | 0.21             | 03-MAR-2020 12:10:35 |
| REGIONA    | 0.21             | 03-MAR-2020 12:10:35 |
| REGIONB    | 0.21             | 03-MAR-2020 12:10:35 |
| REGIONC    | 0.21             | 03-MAR-2020 12:10:35 |
```

Global Buffer Dashboard

The %dirty global buffer dashboard displays the region-wise percentage of global buffers vs dirty global buffers in a bar graph and the number of global buffers in a table. A large number of global buffers implies a large number of dirty global buffers to be flushed at an epoch. This dashboard along with the dynamic journal file chain can be used to determine an appropriate epoch interval / tapering settings for the application and help troubleshoot I/O spikes and process hangs during high loads.

This %JSWRITE reference implementation uses the STDIN^%JSWRITE singlesub entrypoint of the %JSWRITE utility routine.

Example:

```
$ node node_modules/@fis-gtm/jswrite/reference_implementations/gtmjsaccessbg.js
```

^%YGBLSTAT Sparkline Charts

%YGBLSTAT Sparkline shows region-wise sparkline charts for the specified collection of statistics. Typically, you would group statistics by their relevance, for example, TR0,TR1,TR2,TR3, and TR4 can be grouped together to visualize variations for TP Restarts in a condensed form.

This %JSWRITE reference implementation uses the STDIN^%JSWRITE singlesub entrypoint of the %JSWRITE utility routine.

Example:

```
# Set the environment variable LC_ALL to a UTF8 locale (needed for the %JSWRITE Sparkline Chart
▶ (GTMJSGVSTATS.m/gtmjsgvstats.js) reference implementation)
node node_modules/@fis-gtm/jswrite/reference_implementations/gtmjsgvstats.js   # By default, this reference
▶ implementation displays a sparkline chart for SETs and GETs. For monitoring other statistics, open
 gtmjsgvstats.js and update the STAT array to include the list of statistics which require sparkline chart style
 monitoring.
```

^%YGBLSTAT to JSON

GTMJSGVSTATS.m is routine used in the %YGBLSTAT sparkline chart reference implementation. You can use the same routine to create a JSON file which can be imported to spreadsheet software such as Excel. The following command creates gvstats.json which includes all statistics produced with the %YGBLSTAT utility for all regions which have opted for statistics sharing.

```
$gtm_dist/mumps -r ^GTMJSGVSTATS | $gtm_dist/mumps -r STDIN^%JSWRITE singlesub >
▶ /src/toExcel/gvstats.json
```

You can also choose the statistics that you want in the JSON file with the following:

```
$gtm_dist/mumps -r ^GTMJSGVSTATS TR0,TR1,TR2,TR3,TR4 | $gtm_dist/mumps -r STDIN^%JSWRITE singlesub >
▶ /src/toExcel/gvstats.json
```

You can import gvstats.json in Excel using the "Data->Get Data->From File->From JSON" option of Excel.

This %JSWRITE reference implementation uses the STDIN^%JSWRITE singlesub entrypoint of the %JSWRITE utility routine.

JSWRITE Utility Class

JSWRITE.js is a utility class for managing objects created with the ^%JSWRITE utility routine of GT.M. It helps with object mutation and cases when you need to round-trip data back to GT.M. The use of JSWRITE.js is optional and is provided as a convenience to JavaScript developers working on ^%JSWRITE utility routine.

The constructor of JSWRITE.js takes one argument which can be an array collection object from %JSWRITE(glvn,[#]|[*]), one object string %JSWRITE(glvn,#|*), or an empty object. After instantiation, the object has the following getter function.

Getter Method

getSubs	Returns an array containing the unsubscripted GT.M glvname and their subscripts. The first element of the array is always the glvname.
getValue	Returns the value of the GT.M glvname.
getZWRLines	Contains an array of ZWR lines with each element represents one valid argument for use with SET @ in GT.M.

For example, for following GT.M lv structure:

```
GTM>zwrite
A("One","Two","Three","Four")="Demo"
GTM>do ^%JSWRITE("A","*")
{"A":{"One":{"Two":{"Three":{"Four":"Demo"}}}}}
```

..instantiate an object as follows:

```
> var JSWRITE=require("@fis-gtm/jswrite")
> JSObj=new JSWRITE({"A":{"One":{"Two":{"Three":{"Four":"Demo"}}}}})
```

The getter functions return the following values:

JSObj.getSubs	['A', 'One', 'Two', 'Three', 'Four']
JSObj.getValue	'Demo'
JSObj.getZWRLines	['A("One","Two","Three","Four")="Demo"']

Other Methods

toZWR(JSONobj)	Attempts to traverse and convert the JSON paths and their values of JSONObj to a form that can be used with SET @. The argument must be an object. Specifying an array produces an error. Invoking the .toZWR(JSONObj) method appends the getZWRLines array.
customZWR(glvName, [subscripts], glvValue)	Returns an array containing the unsubscripted GT.M glvname and their subscripts. The first element of the array is always the glvname.
validateJSON(str)	Checks whether the string argument is a valid JSON object. If str is a valid JSON string, it returns the result of JSON.parse(str).

Examples:

.toZWR()

```
>var JSWRITE=require("@fis-gtm/jswrite")
undefined
> JSObj=new JSWRITE({})
undefined
> JSObj.toZWR({"A":{"A1":[1,2,{"C":"C3"},4]}})
undefined
> JSObj.getZWRLines
[
   'A("A1","0")="1"',
   'A("A1","1")="2"',
   'A("A1","2","C")="C3"',
   'A("A1","3")="4"'
]
>
```

.customZWR()

```
> JSObj.customZWR("A",[1,2,3,4,5],"ABCD\n")
undefined
> JSObj.getZWRLines
[ 'A(1,2,3,4,5)="ABCD"_$C(10)_""' ]
```

%LCASE

The %LCASE utility converts a string to all lower-case letters. If a routine uses this function repetitively, put the utility code directly into the M program.

Utility Labels

INT: Converts interactively a string to lower-case.

FUNC(s): Invokes %LCASE as an extrinsic function returning the lower-case form of the argument.

Prompts

String: Requests a string for conversion to lower case.

Input Variables

%S: As input, contains string to be converted to lower case.

Output Variables

%S: As output, contains the converted string in lower case.

Examples of %LCASE

Example:

```
GTM>DO INT^%LCASE
String: LABEL
Lower: label
```

This example invokes %LCASE in interactive mode using INT^%LCASE. %LCASE prompts for a string that it converts to all lower case.

Example:

```
GTM>SET %S="Hello"
GTM>do ^%LCASE
GTM>zwrite
%S="hello"
```

This example sets the variable %S to the string "Hello" and invokes %LCASE non-interactively to convert the string.

Example:

```
GTM>SET ^X="Hello"
```

```
GTM>WRITE $$FUNC^%LCASE(^X)
hello
```

This example sets the variable ^X to the string "Hello" and invokes %LCASE as an extrinsic function that returns "hello" in lower case.

%OD

The %OD utility converts numeric values from octal to decimal. %OD returns the decimal number in the read-write variable %OD. %OD rejects input numbers beginning with a minus (-) sign and returns null (""). The routine has entry points for interactive or non-interactive use.

Utility Labels

INT: Converts octal number entered interactively to decimal number.

FUNC(oct): Invokes %OD as an extrinsic function returning the decimal equivalent of the argument.

Prompts

Octal: Requests an octal number for conversion to decimal.

Input Variables

%OD: As input, contains input octal number.

Output Variables

%OD: As output, contains the converted number in decimal.

Examples of %OD

Example:

```
GTM>DO INT^%OD
Octal:14
GTM>ZWRITE
%OD=12
```

This example invokes INT^%OD to interactively convert the octal number entered. %OD prompts for an octal number that it converts to a decimal. %OD returns the converted value in the variable %OD.

Example:

```
GTM>SET %OD=14
GTM>DO ^%OD
GTM>ZWRITE
%OD=12
```

This example sets the read-write variable %OD to 14 and invokes %OD to convert the number non-interactively. ZWRITE displays the contents of the variables.

Example:

```
GTM>WRITE $$FUNC^%OD(14)
12
```

This example invokes %OD as an extrinsic function with the FUNC label. The argument specifies the number to be converted.

%OH

The %OH utility converts numeric values from octal to hexadecimal. %OH returns the hexadecimal number in the read-write variable %OH. %OH rejects input numbers beginning with a minus (-) sign. The routine has entry points for interactive or non-interactive use. In interactive mode, %OH rejects non-octal numbers with the following message, "Input must be an octal number". In non-interactive mode, %OH returns a null string ("") upon encountering a non-octal number.

Utility Labels

INT: Converts interactively octal number entered to hexadecimal number.

FUNC(oct): Invokes %OH as an extrinsic function returning the hexadecimal equivalent of the argument.

Prompts

Octal:Requests an octal number for conversion to hexadecimal.

Input Variables

%OH: As input, contains input octal number.

Output Variables

%OH: As output, contains the converted number in hexadecimal.

Examples of %OH

Example:

```
GTM>DO INT^%OH
Octal:16
GTM>ZWRITE
%OH="E"
```

This example invokes %OH in interactive mode using INT^%OH. %OH prompts for an octal number that it converts to a hexadecimal number. ZWRITE displays the contents of the variable.

Example:

```
GTM>SET %OH=16
GTM>DO ^%OH
GTM>ZWRITE
%OH="E"
```

This example sets the read-write variable %OH to 16 and invokes %OH to convert the value of %OH non-interactively. ZWRITE displays the contents of the variable.

Example:

```
GTM>WRITE $$FUNC^%OH(16)
E
```

This example invokes %OH as an extrinsic function with the FUNC label.

%UCASE

The %UCASE utility converts a string to all upper-case letters. If a routine uses this function repetitively, put the utility code directly into the M program.

Utility Labels

INT: Converts a string to upper case interactively.

FUNC(s): Invokes %UCASE as an extrinsic function, returning the upper-case form of the argument.

Prompts

String: Requests a string for conversion to upper case.

Input Variables

%S: As input, contains string to be converted to upper case.

Output Variables

%S: As output, contains the converted string in upper case.

Examples of %UCASE

Example:

```
GTM>DO INT^%UCASE
String: test
Upper: TEST
```

This example invokes %UCASE in interactive mode using INT^%UCASE. %UCASE prompts for a string that it converts to all upper case.

Example:

```
GTM>SET ^X="hello"
GTM>WRITE $$FUNC^%UCASE(^X)
HELLO
```

This example sets the variable X to the string "hello" and invokes %UCASE as an extrinsic function that returns "HELLO" in upper case.

Mathematic Utilities

The mathematic utilities are:

%EXP: Raises one number to the power of another number.

%SQROOT: Calculates the square root of a number.

The mathematic utilities can be invoked as extrinsic functions.

The "%" sign has been removed from the topic headings below, intentionally.

%EXP

The %EXP utility raises one number provided to the power of another number provided. While this utility provides an interactive interface for exponential calculations, most production code would perform inline calculation with the "**" operator. The routine has entry points for interactive or non-interactive use.

Utility Labels

INT: Calculates a number to the power of another number interactively.

FUNC(i,j): Invokes %EXP as an extrinsic function returning the first argument raised to the power of the second argument.

Prompts

Power: Requests an exponent or power.

Number: Requests a base number to raise by the power.

Input Variables

%I: As input, contains number to be raised to a power.

%J: Contains exponential power by which to raise %I.

Output Variables

%I: As output, contains the result of the exponential calculation.

Examples of %EXP

Example:

```
GTM>DO INT^%EXP
Power: 3
Number: 12
```

```
12 raised to 3 is 1728
```

This example invokes %EXP in interactive mode using INT^%EXP. %EXP prompts for an exponent (power) and a base number.

Example:

```
GTM>SET %I=2,%J=9
GTM>DO ^%EXP
GTM>ZWRITE
%I=512
%J=9
```

This example sets the read-write variable %I to 2, variable %J to 9, and invokes %EXP to calculate the result. ZWRITE displays the contents of the variables. %I contains the result.

Example:

```
GTM>WRITE $$FUNC^%EXP(2,9)
512
```

This example invokes %EXP as an extrinsic function with the label FUNC.

%SQROOT

The %SQROOT utility calculates the square root of a number provided. While this utility provides an interactive interface for taking square roots, most production code would perform inline calculation by raising a number to the .5 power (n**.5). The routine has entry points for interactive or non-interactive use.

Utility Labels

INT: Calculates the square root of a number interactively.

FUNC(s): Invokes %SQROOT as an extrinsic function returning the square root of the argument.

Prompts

The square root of: Requests a number.

Input Variables

%X: Contains the number for which to calculate the square root.

Output Variables

%Y: Contains the square root of %X.

Examples of %SQROOT

Example:

```
GTM>SET %X=81
```

```
GTM>DO ^%SQROOT
GTM>ZWRITE
%X=81
%Y=9
```

This example sets the variable %X to 81 and invokes %SQROOT to calculate the square root non-interactively. ZWRITE displays the contents of the variables.

Example:

```
GTM>DO INT^%SQROOT
The square root of: 81 is: 9
The square root of: <RETURN>
GTM>
```

This example invokes INT^%SQROOT interactively that prompts for a number. The square root of the number appears on the same line. %SQROOT then prompts for another number. Press <RETURN> to exit.

Example:

```
GTM>WRITE $$FUNC^%SQROOT(81)
9
```

This example invokes %SQROOT as an extrinsic function with the label FUNC.

String Utilities

%TRIM

%TRIM removes leading and trailing characters from a string. The format of the %TRIM utility function is:

```
$$FUNC|$$L|$$R^%TRIM(expr1[,expr2])
```

- The first expression specifies the string. The optional second expression specifies a list of trailing and leading characters to remove from expr1. When expr2 is not specified, ^%TRIM assumes expr2 as $char(9,32) which removes all trailing and leading whitespaces (spaces and tabs) from expr1. Note that ^%TRIM treats expr2 as a list of characters (not a substring).

- The $$FUNC label trims leading and trailing characters.

- The $$L label trims leading characters.

- The $$R label trims trailing characters.

You can also use %TRIM as a command line utility routine to read from STDIN and write to STDOUT in the following format:

```
echo "  string with leading and trailing spaces  " | $gtm_dist/mumps -r ^%TRIM
```

Example:

```
GTM>set strToTrim=$char(9,32)_"string with spaces and tabs"_$char(32,32,32) write $length(strToTrim)
32
GTM>write
```

```
▶  "strToTrim=",?24,"""",strToTrim,"""",!,"$$L^%TRIM(strToTrim)=",?24,"""",$$L^%TRIM(strToTrim),"""",!,"$
$R^%TRIM(strToTrim)=",?24,"""",$$R^%TRIM(strToTrim),"""",!,"$$FUNC^%TRIM(strToTrim)=",?24,"""",$$FUNC^
%TRIM(strToTrim),""""
strToTrim=                  "          string with spaces and tabs    "
$$L^%TRIM(strToTrim)=   "string with spaces and tabs    "
$$R^%TRIM(strToTrim)=   "          string with spaces and tabs"
$$FUNC^%TRIM(strToTrim)="string with spaces and tabs"
```

▼

This example invokes %TRIM as an extrinsic function and demonstrates the use of its $$L,$$R, and $$FUNC labels.

Example:

```
$ echo " GT.M Rocks! " | $gtm_dist/mumps -r ^%TRIM
GT.M Rocks!
$
```

This example invokes %TRIM as a command line utility which reads STDIN and writes the trimmed output to STDOUT.

%MPIECE

The %MPIECE utility replaces one or more consecutive occurrences of the second argument in the first argument with one occurrence of the third argument. This lets $PIECE operate on the resulting string like UNIX awk.

You can use the %MPIECE utility in Direct Mode or include it in a source application program in the following format:

```
$$^%MPIECE(str,expr1,expr2)
```

If expr1 and expr2 are not specified, %MPIECE assumes expr1 to be one or more consecutive occurrences of whitespaces and expr2 to be one space.

%MPIECE removes all leading occurrences of expr1 from the result.

Utility Labels

$$SPLIT^%MPIECE(str,expr1): Invokes %MPIECE as an extrinsic function that returns an alias local array of string divided into pieces by expr1. If expr1 is not specified, MPIECE assumes expr1 to be one or more consecutive occurrences of whitespaces.

Example:

```
GTM>set strToSplit=" please split this string into six"
GTM>set piecestring=$$^%MPIECE(strToSplit," ","|") zwrite strToSplit,piecestring write $length(piecestring,"|")
strToSplit=" please split this string into six"
piecestring="please|split|this|string|into|six"
6
GTM>set *fields=$$SPLIT^%MPIECE(strToSplit) zwrite fields
fields(1)="please"
fields(2)="split"
fields(3)="this"
fields(4)="string"
fields(5)="into"
fields(6)="six"
```

Global Utilities

The Global utilities are:

%G: Displays global variables and their values.

%GC: Copies a global or global sub-tree.

%GCE: Replaces a specified value or part of a value in a set of variables.

%GD: Displays existing globals in the current global directory without displaying their values or descendants.

%GED: Provides full-screen editing capabilities for global variables and values.

%GI: Loads global data from a sequential file into a GT.M database.

%GO: Extracts global data from a GT.M database into a sequential file.

%GSE: Displays global variables and their values when the values contain a specified string or number.

%GSEL: Selects globals.

%ZSHOWVTOLCL: Restores ZSHOW "V":gvn data into its original local variables.

The "%" sign has been removed from the topic headings below, intentionally.

%G

The %G utility displays names, descendants and values of globals currently existing in the database. Use %G to examine global variables and their values. Enter a question mark (?) at any prompt to display help information.

Prompts

Output Device: <terminal>:

Requests a destination device; defaults to the principal device.

List ^Requests the name, in ZWRITE format, of a global to display.

For descriptions of valid input to the List ^ prompt, see the following table.

Arguments for %G and %GED:

ITEM	DESCRIPTION	EXAMPLES
Global name	M name	SQL, %5
	M pattern form to match several globals	?1"A".E, ?1A1"TMP"
	asterisk to match all global names	*

ITEM	DESCRIPTION	EXAMPLES
	global directory lists request	?D
Subscripts following a global name in parentheses	M expr	"rick",599,X, $e(a,7)*10
	[expr]:[expr] for a range	1:10, "A":"F", :4, PNT:, :
	M pattern form to match certain subscripts	1"E"3N, ?1"%F".E
	* descendants	*

Examples of %G

Example:

```
GTM>do ^%G
Output Device: <terminal>: <RETURN>
List ^C
^C="CLASS"
^C(1)="MARY"
^C(1,2)="MATH"
^C(1,2,1)=80
^C(1,3)="BIO"
^C(1,3,1)=90
^C(2)="JOHN"
^C(3)="PETER"
List ^ <RETURN>
GTM>
```

This example lists the nodes of global ^C. %G displays the global and its descendants and values, if the node exists.

Example:

```
GTM>do ^%G
Output Device: <terminal>: <RETURN>
List ^C(1)
^C(1)="MARY"
```

This example lists only the node entered and its value.

Example:

```
GTM>do ^%G
Output Device: <terminal>: <RETURN>
List ^C(1,*)
^C(1)="MARY"
^C(1,2)="MATH"
^C(1,2,1)=80
^C(1,3)="BIO"
```

```
^C(1,3,1)=90
List ^ <RETURN>
GTM>
```

This example uses the asterisk (*) wildcard to list node ^C(1), its descendants and values.

Example:

```
GTM>do ^%G
Output Device: <terminal>: <RETURN>
List ^?D
Global Directory
Global ^ <RETURN>
^C ^D ^S ^Y ^a
Total of 5 globals.
List ^
GTM>
```

This example specifies "?D" as the global that invokes the %GD utility. %GD displays existing globals in the current global directory without displaying their values or descendants.

%GC

The %GC utility copies values of globals from one global to another. It is useful for testing and for moving misfiled data.

Prompts

Show copied nodes <Yes>?:

Asks whether to display the "source nodes" on the principal device.

From global: ^Requests a global variable name from which to copy variable and descendants.

To global: ^Request a global variable name to receive the copy.

Examples of %GC

Example:

```
GTM>do ^%GC
Global copy
Show copied nodes <Yes>? <RETURN>
From global ^b
To global ^g
^g(1)=1
^g(2)=2
^g(3)=3
Total 3 nodes copied.
From global ^<RETURN>
GTM>
```

This example makes a copy of the nodes and values of global ^b to global ^g.

%GCE

The %GCE utility changes every occurrence of a string within the data of selected global nodes to a replacement string. ^%GCE changes the string in each place it occurs, even if it forms part of a longer string. For example, changing the string 12 to 55 changes 312 to 355.

%GCE displays the name of each global as it is processed. You can suppress the output of the names of globals in which no changes are made by using the QUIET utility label.

Utility Labels

Prompts

Global^: Requests (using %GSEL) the name(s) of the globals to change; <RETURN> ends selection.

Old string: Requests an existing string to find.

New string: Requests the replacement string.

Show changed nodes <Yes>?:

Asks whether to display the before and after versions of modified nodes on the current device.

Output Device: <terminal>:

Requests a destination device; defaults to the principal device.

QUIET: Only displays the names of globals in which changes are made.

Examples of %GCE

Example:

```
GTM>DO ^%GCE
Global Change Every occurrence
Global ^a:^b
^a ^b
Current total of 2 globals.
Global ^ <RETURN>
Old String: hello
New String: good-bye
Show changed nodes <Yes>?: <RETURN>
Output Device: <terminal>: <RETURN>
^a
No changes made in total 1 nodes.
^b
^b(10)
Was : hello Adam
Now : good-bye Adam
1 changes made in total 25 nodes.
Global ^ <RETURN>
GTM>
```

This example searches a range of globals and its nodes for the old string value entered. GT.M searches each global and displays the changes and number of nodes changed and checked.

Example:

```
GTM>set ^b(12)=12
GTM>set ^b(122)=122
GTM>set ^b(30)=656
GTM>set ^b(45)=344
GTM>set ^b(1212)=012212
GTM>DO ^%GCE
Global Change Every occurrence
Global ^b
Current total of 1 global.
Global ^ <RETURN>
Old String: 12
New String: 35
Show changed nodes <Yes>?: <RETURN>
Output Device: <terminal>: <RETURN>
^b(12)
Was : 12
Now : 35
^b(122)
Was : 122
Now : 352
^b(1212)
Was : 12212
Now : 35235
5 changes made in total 5 nodes
Global ^ <RETURN>
GTM>DO ^%G
Output device: <terminal>: <RETURN>
List ^b
^b(12)=35
^b(30)=656
^b(45)=344
^b(122)=352
^b(1212)=35235
```

This example shows that executing %GCE replaces all occurrences of "12" in the data stored in the global ^b with "35" and displays the affected nodes before and after the change. Then the %G demonstrates that "12" as data was changed, while "12" in the subscripts remained untouched.

%GD

The %GD utility displays existing globals in the current global directory without displaying their values or descendants.

%GD prompts for a global name and redisplays the name if that global exists.

%GD interprets a percent sign (%) in the first position of a global name literally.

%GD allows the wildcard characters asterisk (*) and question mark (?). The wildcards carry their usual meanings, an asterisk (*) denotes a field or a portion of a field, and a question mark (?) denotes a single character.

A colon (:) between two globals specifies a range. %GD displays existing globals within that range.

After each selection %GD reports the number of globals selected by the input.

A question mark (?) entered at a prompt displays help information. Pressing <RETURN> exits %GD.

Prompts

Global^: Requests (using %GSEL) a global name with optional wildcards or a range of names; <RETURN> terminates %GD.

Examples of %GD

Example:

```
GTM>DO ^%GD
Global directory
Global ^k
^k
Total of 1 global.
Global ^ <RETURN>
GTM>
```

This example verifies that ^k exists in the global directory.

Example:

```
GTM>DO ^%GD
Global directory
Global ^C:S
^C ^D ^S
Total of 3 globals
Global ^ <RETURN>
GTM>
```

This example displays a range of globals that exist from ^C to ^S.

Example:

```
GTM>DO ^%GD Global directory
Global ^*
^C ^D ^S ^Y ^a
Total of 5 globals
Global ^ <RETURN>
GTM>
```

The asterisk (*) wildcard at the Global ^ prompt displays all globals in the global directory.

%GED

The %GED utility enables you to edit the globals in a full-screen editor environment. %GED invokes your default editor as specified by the EDITOR environment variable. When you finish the edit, use the [save and] exit command(s) of the editor you are using, to exit.

Prompts

Edit^: Requests the name, in ZWRITE format, of a global to edit.

Only one global can be edited at a time with %GED, see " Prompts" (page 483) for descriptions of valid input for subscripts.

Examples of %GED

Example:

```
GTM>DO ^%GED
edit ^ b
Beginning screen:
^b(1)="melons"
^b(2)="oranges"
^b(3)="bananas"
Screen with a change to ^b(1), elimination of ^b(3), and two new entries ^b(4) and ^b(5):
^b(1)="apples"
^b(2)="oranges"
^b(4)=pears
^b(5)="grapes"
%GED responds:
Invalid syntax: b(4)=pears
return to continue:
After screen:
^b(1)="apples"
^b(2)="oranges"
^b(4)="pears"
^b(5)="grapes"
%GED responds:
node: ^b
selected: 3
changed: 1
added: 2
killed: 1
Edit ^ <RETURN>
GTM>
```

This example shows the use of the full-screen editor to change, add, and delete (kill) nodes. When you exit from the editor, %GED checks the syntax and reports any problems. By pressing <RETURN>, return to the full-screen editor to fix the error. At the end of the session, %GED reports how many nodes were selected, changed, killed, and added.

%GI

%GI loads global variable names and their corresponding data values into a GT.M database from a sequential file. %GI uses the global directory to determine which database files to use. %GI may operate concurrently with normal GT.M database access. However, a %GI does not use M LOCKs and may produce application-level integrity problems if run concurrently with many applications.

In many ways, %GI is similar to MUPIP LOAD. The format of the input file (GO or ZWRITE) is automatically detected. Like MUPIP LOAD, %GI does not load GT.M trigger definitions. Unlike MUPIP LOAD, %GI invokes triggers just like any other M code, which may yield results other than those expected or intended.

^%GI loads records having up to 1MiB string length.

Prompts

Enter input file:

Requests name of a file; file should be in standard Global Output (GO) format or Zwrite (ZWR) format .

OK <Yes>?: Asks for confirmation.

Examples of %GI

Example:

```
GTM>DO ^%GI
Global Input Utility
Input device <terminal>: DATA.GBL
Saved from user's development area
GT.M 07-MAY-2010 14:14:09
OK <Yes>? <RETURN>
^IB ^INFO
Restored 10 nodes in 2 globals
GTM>
```

%GO

%GO copies specified globals from the current database to a sequential output file in either GO or ZWR format. Use %GO to back up specific globals or when extracting data from the database for use by another system. %GO uses the global directory to determine which database files to use. %GO may operate concurrently with normal GT.M database access. To ensure that a %GO reflects a consistent application state, suspend database updates to all regions involved in the extract.

In many ways, the %GO utility is similar to MUPIP EXTRACT (-FORMAT=GO or -FORMAT=ZWR). Like MUPIP EXTRACT, %GO does not extract and load GT.M trigger definitions.

Prompts

Global^: Requests (using %GSEL) the name(s) of the globals to search; <RETURN> ends selection.

Header label: Requests text describing contents of extract file.

Output Format: GO or ZWR:

Requests the format to output the data. Defaults to ZWR.

Output Device: <terminal>:

Requests destination device, which may be any legal filename.

Examples of %GO

Example:

```
GTM>DO ^%GO
Global Output Utility
Global ^A
^A
Current total of 1 global
Global ^<RETURN>
Header label: Revenues May, 2010
Output Format: GO or ZWR: ZWR
Output device: /usr/dev/out.go
^A
Total of 1 node in 1 global.
```

```
GTM>
```

%GSE

The %GSE utility finds occurrences of a string within the data values for selected global nodes and displays the variable name and data on a specified output device.

%GSE displays the name of each global as it is processed. You can suppress the output of the names of globals in which the search string is not found by using the QUIET utility label.

Prompts

Output Device: <terminal>:

Requests a destination device; defaults to the principal device.

Global^: Requests (using %GSEL) the name(s) of the globals to search; <RETURN> ends selection.

String: Requests a search string.

Utility Labels

QUIET: Only displays the names of globals in which the search string is found

Examples of %GSE

Example:

```
GTM>do ^%GSE
Global Search For Every Occurence
Output device: <terminal>: Test.dat
Global ^a <RETURN>
^a
Current total of 1 global.
Global ^ <RETURN>
String: Hello
^a
^a(10) Hello Adam
Total 1 matches found in 25 nodes.
Global ^ <RETURN>
GTM>
```

This example searches global ^a for the string "Hello" and displays all nodes that contain that string.

%GSEL

The %GSEL utility selects globals. %GSEL creates a variable %ZG that is a local array of the selected globals. After each selection %GSEL displays the number of globals in %ZG.

- %GSEL accepts the wildcard characters asterisk (*), percent sign (%) and question mark (?). The wildcards carry their usual meanings, asterisk (*) denoting a field or a portion of a field, and question mark (?) or percent sign (%) denoting a single character.

- The wildcards question mark (?) and percent sign (%) lose their meanings when in the first position of a global name:

  - When '%' is in the first position of a global name, %GSEL interprets it literally. For example, "%*" means all global names starting with '%'.

  - When you specify only '?' as a global name, %GSEL displays the online help.

  - When you specify a '?' followed by a 'D' or 'd', %GSEL displays the global names currently in the %ZG array.

  - %GSEL produces an error if there is '?' at the first position of a global name followed by any character other than 'D' or 'd'. For example, "?a" produces an error.

- A colon (:) between two globals specifies a range. %GSEL produces an error if you specify a '?' as the first character after a colon (:).

- A minus sign (-) or quotation mark (') as the first character will cause the search to remove the proceding global or range from the %ZG array.

Utility Labels

CALL: Runs %GSEL without reinitializing %ZG.

Output Variables

%ZG Contains array of all globals selected.

Prompts

Global^: Requests a global name with optional wildcards or a range of names.

Examples of %GSEL

Example:

```
GTM>DO ^%GSEL
Global ^C
^C
Current total of 1 global.
Global ^*
^S        ^Y        ^c        ^class
Current total of 5 globals.
Global ^-S
^S
Current total of 4 globals.
Global ^'Y
^Y
Current total of 3 globals.
Global ^?D
^C        ^c        ^class
Current total of 3 globals.
Global ^
GTM>ZWRITE
```

```
%ZG=3
%ZG("^C")=""
%ZG("^c")=""
%ZG("^class")=""
GTM>
```

This example adds and subtracts globals from the list of selected globals. "?D" displays all globals selected. ZWRITE displays the contents of the %ZG array.

Example:

```
GTM>DO ^%GSEL
Global ^a
^a
Current total of 1 global.
Global ^
GTM>ZWRITE
%ZG=1
%ZG("^a")=""
GTM>DO CALL^%GSEL
Global ^?d
^a
Current total of 1 global.
Global ^iv
^iv
Current total of 2 globals.
Global ^
GTM>ZWRITE
%ZG=2
%ZG("^a")=""
%ZG("^iv")=""
GTM>
```

This example uses CALL^%GSEL to add to an existing %ZG array of selected globals.

%ZSHOWVTOLCL

The %ZSHOWVTOLCL utility restores ZSHOW "V":gvn data into its original local variables. Invoke this utility with $ECODE set to the empty string. This utility facilitates automated restoration even of nodes exceeding the maximum record size of the global.

Input Variables

%ZSHOWvbase: The name of the global variable destination for ZSHOW "V". Note that %ZSHOWVTOLCL cannot restore a local variable with the name %ZSHOWvbase.

Routine Utilities

The routine utilities are:

%FL: Lists the comment lines at the beginning of source programs.

%RANDSTR: Generates a random string.

%RCE: Replaces every occurrence of a text string with another text string in a routine or a list of routines.

%RD: Lists routine names available through $ZROUTINES.

%RI: Loads routines from RO file to *.m files in GT.M format.

%RO: Writes M source code for one or more routines to a sequential device such as a terminal, or a disk file.

%RSE: Searches for every occurrence of a text string in a routine or a list of routines.

%RSEL: Selects M routines and places their directories and names in a local array.

The "%" sign has been removed from the topic headings below, intentionally.

%FL

The %FL utility lists the comment lines at the beginning of source programs. %FL writes the routines in alphabetical order to the specified device. If the output device is not the principal device, %FL displays the name of each routine on the principal device as it writes the routine to the output device.

%FL uses %RSEL to select routines. For more information, see " %RSEL" (page 502).

Prompts

Routine: Requests the name(s) of the routines (using %RSEL); <RETURN> ends the selection.

Output Device: <terminal>:

Requests a destination device; defaults to the principal device.

Examples of %FL

Example:

```
GTM>DO ^%FL
First Line Lister
Routine: %D
%D
Current total of 1 routine.
Routine: %GS*
%GSE %GSEL
Current total of 3 routines.
Routine: - %D
%D
Current total of 2 routines.
Routine: ?D
%GSE %GSEL
Routine: <RETURN>
Output Device: <RETURN>
Routine First Line Lister Utility
GT.M 21-MAR-2002 16:44:09
%GSE
%GSE;GT.M %GSE utility - global search
;
```

```
%GSEL;
%GSEL;GT.M %GSEL utility - global select into a local array
;
;invoke ^%GSEL to create %ZG - a local array of existing globals, interactively
;
Total 5 lines in of 2 routines.
GTM>
```

This example selects %D, then selects %GSE and %GSEL and deselects %D. Because the example enters <RETURN> at the Output Device: <terminal>: prompt, the output goes to the principal device.

%RANDSTR

%RANDSTR generates a random string. The format %RANDSTR is:

```
%RANDSTR (strlen,charranges,patcodes,charset)
```

The random string is of length strlen from an alphabet defined by charset or by charranges and patcodes.

strlen: the length of the random string.

charranges: Range of alphabets defined by charset. By default charranges is 1:1:127. charranges uses the same syntax used for FOR loop ranges, for example, 48:2:57 to select the even decimal digits or 48:1:57,65:1:70 to select hexadecimal digits.

patcodes: specifies pattern codes used to restrict the characters to those that match the selected codes. By default, patcodes is "AN".

charset: Specifies a string of non-zero length. If specified, %RANDSTR generates the random string using the characters in charset, otherwise it takes its alphabet as specified by charranges and patcodes. If charset is of zero length, and is passed by reference, %RANDSTR() initializes it to the alphabet of characters defined by charranges and patcodes. If not specified, strlen defaults to 8, charranges defaults to 1:1:127 and patcodes to "AN".

%RCE

The %RCE utility replaces every occurrence of a text string with another text string in a routine or a list of routines.

%RCE uses %RSEL to select routines. For more information, see " %RSEL" (page 502).

%RCE prompts for a text string to replace and its replacement. %RCE searches for text strings in a case-sensitive manner. %RCE issues a warning message if you specify a control character such as a <TAB> in the text string or its replacement. %RCE confirms your selection by displaying the text string and its replacement between a left and right arrow. The arrows highlight any blank spaces that you might have included in the text string or its replacement.

Regardless of whether you select a display of every change, %RCE displays the name of each routine as it is processed. You can suppress the output of the names of routines in which no changes are made by using the QUIET and QCALL utility labels. %RCE completes processing with a count of replacements and routines changed.

Prompts

Routine: Requests (using %RSEL) the name(s) of the routines to change; <RETURN> ends the selection.

Old string: Requests string to be replaced.

New string: Requests replacement string.

Show changed lines <Yes>?:

Asks whether to display the before and after versions of the modified lines on an output device.

Output Device: <terminal>:

Requests a destination device; defaults to the principal device.

Utility Labels

QUIET: Only displays the names of routines in which changes are made.

CALL: Works without user interaction unless %ZR is not defined.

QCALL: Works without user interaction unless %ZR is not defined. Only displays the names of routines in which changes are made.

Input Variables

The following input variables are only applicable when invoking CALL^%RCE.

%ZR: Contains an array of routines provided or generated with %RSEL.

%ZF: Contains string to find.

%ZN: Contains a replacement string.

%ZD: Identifies the device to display the change trail, defaults to principal device. Make sure you open the device if the device is not the principal device.

%ZC: Truth-value indicating whether to display the change trail, defaults to 0 (no).

Examples of %RCE

Example:

```
GTM>DO ^%RCE
Routine Change Every occurrence
Routine: BES*
BEST BEST2 BEST3 BEST4
Current total of 4 routines
Routine: <RETURN>
Old string:^NAME
New string:^STUDENT
Replace all occurrences of:
>^NAME<
With
>^STUDENT<
Show changed lines <Yes>?: <RETURN>
Output Device: <RETURN>
```

```
/usr/smith/work/BEST.m
Was: S ^NAME=SMITH
Now: S ^STUDENT=SMITH
Was: S ^NAME(1)=JOHN
Now: S ^STUDENT(1)=JOHN
/usr/smith/work/BEST2.m
/usr/smith/work/BEST3.m
Was: S ^NAME=X
Now: S ^STUDENT=X
Was: W ^NAME
Now: W ^STUDENT
/usr/smith/work/BEST4.m
Total of 4 routines parsed.
4 occurrences changed in 2 routines.
GTM>
```

This example selects a list of routines that change the string "^NAME" to the string "^STUDENT," and displays a trail of the changes.

Example:

```
GTM>DO ^%RCE
Routine Change Every occurrence
Routine: BES*
BEST BEST2 BEST3 BEST4
Current total of 4 routines
Routine: <RETURN>
Old String:<TAB>
The find string contains control characters
New string: <RETURN>
Replace all occurrences of:
><TAB><
With:
><
Show changed lines <Yes>?: N
BEST BEST2 BEST3 BEST4
Total 4 routines parsed.
4 occurrences changed in 2 routines.
GTM>
```

This example removes all occurrences of the <TAB> key from specified routines and suppresses the display trail of changes.

%RD

The %RD utility lists routine names accessible through the current $ZROUTINES. %RD calls %RSEL and displays any routines accessible through %RSEL. Use %RD to locate routines.

%RD accepts the wildcard characters asterisk (*) and question mark (?). The wildcards carry their usual meanings, an asterisk (*) denotes a field or a portion of a field, and a question mark (?) denotes a single character in positions other than the first.

A colon (:) between two routine names specifies a range of routines. %RD displays only those routine names accessible through the current $ZROUTINES.

After each selection %RD displays the total number of routines listed.

Pressing <RETURN> exits %RD.

Prompts

Routine: Requests (using %RSEL) the name(s) of the routines to list; <RETURN> ends the selection.

Utility Labels

OBJ: Lists object modules accessible through the current $ZROUTINES.

LIB: Lists percent (%) routines accessible through the current $ZROUTINES.

SRC: Lists the source modules accessible through the current $ZROUTINES (same as %RD).

Examples of %RD

Example:

```
GTM>DO ^%RD
Routine directory
Routine: TAXES
TAXES
Total of 1 routine
Routine:*
EMP FICA PAYROLL TAXES YTD
Total of 5 Routines
Routine: <RETURN>
GTM>
```

This example invokes %RD that prompts for routine TAXES and the wildcard (*). %RD lists five routines accessible through the current $ZROUTINES.

Example:

```
GTM>DO OBJ^%RD
Routine directory
Routine:*
EMP FICA
Total of 2 routines
Routine: <RETURN>
GTM>
```

This example invokes %RD with the label OBJ that lists only object modules accessible through the current $ZROUTINES.

Example:

```
GTM>DO LIB^%RD
Routine directory
%D %DATE %DH %G %GD %GSEL
GTM>
```

This example invokes %RD with the LIB label that lists all the % routines accessible through the current $ZROUTINES.

Example:

```
GTM>DO SRC^%RD
Routine directory
Routine:*
DATACHG
Total of 1 routines
Routine: <RETURN>
GTM>
```

This example invokes %RD with the label SRC that lists only source modules accessible through the current $ZROUTINES.

%RI

%RI transforms M routines in the sequential format described in the ANSI standard into individual .m files in GT.M format. ^ %RI handles code lines up to 1 MiB. Note that when a code line exceeds 8 KiB, the GT.M compiler automatically inserts a line separator and issues a warning. Use %RI to make M RO format accessible as GT.M routines.

Prompts

Formfeed delimited <No>?

Requests whether lines should be delimited by formfeed characters rather than carriage returns.

Input Device: <terminal>:

Requests name of RO file containing M routines.

Output Directory:

Requests name of directory to output M routines.

Examples of %RI

Example:

```
GTM>DO ^%RI
Routine Input utility - Converts RO file to *.m files
Formfeed delimited <No>? <RETURN>
Input device: <terminal>: file.ro
Files saved from FILEMAN directory
GT.M 07-MAY-2002 15:17:54
Output directory: /usr/smith/work/
DI DIA DIAO DIAI DIB DIBI
Restored 753 lines in 6 routines.
GTM>
```

%RO

The %RO utility writes M source code for one or more routines to a sequential device such as, a disk file or a printer. .

^%RO handles code lines up to 1 MiB. Note that when a code line exceeds 8 KiB, the GT.M compiler automatically inserts a line separator and issues a warning.

%RO uses %RSEL to select routines. For more information, see " %RSEL" (page 502).

%RO writes the routines in alphabetical order to the specified device. %RO displays the name of each routine as it writes the routine to the device.

Prompts

Routine: Requests (using %RSEL) the name(s) of the routines to output; <RETURN> ends selection.

Output device: <terminal>:

Requests a destination device; defaults to the principal device.

Header label: Requests text to place in the first of the two header records.

Strip comments <No>?:

Asks whether to remove all comment lines except those with two adjacent semicolons.

Utility Labels

CALL: Works without user interaction unless %ZR is not defined.

Input Variables

The following input variables are only applicable when invoking CALL^%RO.

%ZR: Contains an array of routines provided or generated with %RSEL.

%ZD: Identifies the device to display output, defaults to principal device.

Examples of %RO

Example:

```
GTM>DO ^%RO
Routine Output - Save selected routines into RO file.
Routine: %D
%D
Current total of 1 routines.
Routine: -%D
%D
Current total of 0 routines.
Routine: BEST*
BEST BEST1 BEST2
Current total of 3 routines.
Routine: ?D
BEST BEST1 BEST2
Routine: <RETURN>
Output Device: <terminal>: output.txt
Header Label: Source code for the BEST modules.
Strip comments <No>?:<RETURN>
```

```
BEST BEST1 BEST2
Total of 53 lines in 3 routines
GTM>
```

This example adds and subtracts %D from the selection, then adds all routines starting with "BEST" and confirms the current selection. The example sends output to the designated output file output.txt. %RO displays the label at the beginning of the output file. The first record of the header label is the text entered at the prompt. The second record of the header label consists of the word "GT.M" and the current date and time.

%RSE

The %RSE utility searches for every occurrence of a text string in a routine or a list of routines.

%RSE uses %RSEL to select routines. For more information, see " %RSEL" (page 502).

%RSE searches for text strings that are case-sensitive. %RSE issues a warning message if you specify a control character such as a <TAB> in the text string. %RSE confirms your selection by displaying the text string between a left and right arrow. The arrows display any blank spaces included in the text string.

%RSE displays the name of each routine as it is processed. You can suppress the output of the names of routines in which the search string is not found by using the QUIET and QCALL utility labels.

%RSE completes processing with a count of occurrences found.

Prompts

Routine: Requests (using %RSEL) the name(s) of the routines to search; <RETURN> ends selection.

Find string: Requests string for which to search.

Output device: <terminal>:

Requests a destination device; defaults to the principal device.

Utility Labels

QUIET: Only displays the names of routines in which the search string is found.

CALL: Works without user interaction unless %ZR is not defined.

QCALL: Works without user interaction unless %ZR is not defined. Only displays the names of routines in which the search string is found.

Input Variables

The following input variables are only applicable when invoking CALL^%RSE.

%ZR: Contains an array of routines provided or generated with %RSEL.

%ZF: Contains the string to find.

%ZD: Identifies the device to display the results, defaults to principal device. Make sure you open the device if the device is not the principal device.

Examples of %RSE

Example:

```
GTM>DO ^%RSE
Routine Search for Every occurrence
Routine: BES*
BEST BEST2 BEST3 BEST4
Current total of 4 routines
Routine: <RETURN>
Find string:^NAME
Find all occurrences of:
>^NAME<
Output device: <terminal>:
/usr/smith/work/BEST.m
S ^NAME=SMITH
S ^NAME(1)=JOHN
/usr/smith/work/BEST2.m
/usr/smith/work/BEST3.m
S ^NAME=X
W ^NAME
/usr/smith/work/BEST4.m
Total of 4 routines parsed.
4 occurrences found in 2 routines.
GTM>
```

This example invokes %RSE that searches and finds a given string. The output device specifies a terminal display of all lines where the text string occurs.

Example:

```
GTM>DO ^%RSE
Routine Search for Every occurrence
Routine: BEST
BEST
Current total of 1 routine
Routine: <RETURN>
Find string:^NAME
Find all occurrences of:
>^NAME<
Output Device: out.lis
BEST
GTM>
```

This example instructs ^%RSE to write all lines where the text string occurs to an output file, out.lis.

%RSEL

The %RSEL utility selects M routines. %RSEL selects routines using directories specified by the GT.M special variable $ZROUTINES. $ZROUTINES contains an ordered list of directories that certain GT.M functions use to locate source and object files. If $ZROUTINES is not defined, %RSEL searches only the current default directory. Other GT.M utilities call %RSEL.

%RSEL prompts for the name of a routine(s).

%RSEL accepts the wildcard characters asterisk (*) and question mark (?). The wildcards carry their usual meanings: an asterisk (*) denotes a field or a portion of a field, and a question mark (?) denotes a single character in positions other than the first.

A colon (:) between two routines specifies a range.

%RSEL creates a read-write variable %ZR, which is a local array of selected routines. After each selection, %RSEL reports the number of routines in %ZR. A minus sign (-) or an apostrophe (') character preceding a routine name removes that routine from the %ZR array. A question mark (?) provides online help, and "?D" displays M routines currently in the array.

> ## Note
>
> If a local variable %ZRSET is defined, %RSEL places the output information into a global variable (^%RSET) instead of the local variable %ZR.

Prompts

Routine: Requests the name(s) of the routines; <RETURN> ends selection.

Utility Labels

CALL: Performs %RSEL without reinitializing %ZR.

OBJ: Searches only object files.

SRC: Searches only source files (same as %RSEL).

SILENT: Provides non-interactive (batch) access to the functionality of %RSEL. The syntax is **SILENT^%RSEL(pattern,label)** where **pattern** is a string that specifies the routine names to be searched. label can be "OBJ", "SRC" or "CALL". The default is "SRC" value corresponds to ^%RSEL if invoked interactively.

Input Variables

The following input variables are only valid when invoking CALL^%RSEL:

%ZE: Contains the file extension, usually either .m for source files or .o for object files.

%ZR: As input, contains an existing list of routines to be modified.

%ZRSET: On being set, requests %RSEL to place the output in the global variable ^%RSET.

Output Variables

%ZR: As output, contains list of directories indexed by selected routine names.

^%RSET($JOB): The output global variable ^%RSET is used instead of the local variable %RD if the input variable %ZRSET is set. It is indexed by job number $JOB and the selected routine names.

Examples of %RSEL

Example:

```
GTM>DO ^%RSEL
Routine: TES*
TEST2 TEST3
Current total of 2 routines
Routine: <RETURN>
GTM>DO OBJ^%RSEL
Routine:TEST?
Current total of 0 routines
Routine: <RETURN>
GTM>ZWRITE
%ZR=0
```

This example selects two source routines starting with "TES" as the first three characters. Then, the example invokes %RSEL at the OBJ label to select object modules only. OBJ^%RSEL returns a %ZR=0 because object modules for the TEST routines do not exist.

Example:

```
GTM>DO ^%RSEL
Routine: BES*
BEST BEST2 BEST3 BEST4
Current total of 4 routines
Routine: - BEST
BEST
Current total of 3 routines
Routine: ?D
BEST2 BEST3 BEST4
Routine: 'BEST2
BEST2
Current total of 2 routines
Routine: ?D
BEST3 BEST4
Routine: <RETURN>
GTM>ZWRITE
%ZR=2
%ZR("BEST3")="/usr/smith/work/"
%ZR("BEST4")="/usr/smith/test/"
GTM>
```

This example selects the routines using the asterisk (*) wildcard and illustrates how to tailor your selection list. Note that %ZR contains two routines from different directories.

By default, %RSEL bases the contents of %ZR on source files that have a .m extension.

Example:

```
GTM>DO ^%RSEL
Routine:BEST*
BEST2 BEST3
Current total of 2 routines
Routine: <RETURN>
GTM>ZWRITE
```

```
%ZR=2
%ZR("BEST2")="/usr/smith/test/"
%ZR("BEST3")="/usr/smith/test/"
```

This example creates a %ZR array with BEST2 and BEST3.

Example:

```
GTM>DO ^%RSEL
Routine:LOCK
LOCK
Current total of 1 routine
Routine: <RETURN>
GTM>ZWRITE
%ZR=1
%ZR("LOCK")="/usr/smith/work/"
GTM>DO CALL^%RSEL
Routine:BEST*
BEST2 BEST3
Current total of 2 routines
Routine: <RETURN>
GTM>ZWRITE
%ZR=3
%ZR("BEST2")="/usr/smith/work/"
%ZR("BEST3")="/usr/smith/work/"
%ZR("LOCK")="/usr/smith/work/"
GTM>
```

This example creates a %ZR array with LOCK and adds to it using CALL%RSEL.

Example:

```
GTM>do SILENT^%RSEL("myroutine","OBJ")
```

```
GTM>ZWRITE
%ZR=1
%ZR("myroutine")="/usr/smith/work"
```

This example invokes %RSEL non-interactively and creates a %ZR array for myroutine using OBJ%RSEL.

Internationalization Utilities

These utilities are an integral part of the GT.M functionality that permits you to customize your applications for use with other languages. For a description of these utilities, refer to Chapter 12: "Internationalization" (page 543).

The internationalization utilities are:

%GBLDEF

The %GBLDEF utility manipulates the collation sequence assigned to a global other than the default M collation that is based on code value (ASCII for the codes 0 to 127) ordering except for numeric values, which order most negative to most positive before string values. As described in the Internationalization Chapter, alternative collation definition requires supplying routines that transform codes into desired collation order. All subscripted variables for a global must use the same collation sequence. A global collation sequence can only be changed when a global has no subscripted variables defined.

Utility Labels

`get^%GBLDEF(gname[,reg])`

get: returns a value associated with the current global variable collation in the form nct,act,ver, where:

- if nct is FALSE (0), it indicates the current global variable collation follows the M standard of collating numeric values before all non-empty strings, and if TRUE (1), the global uses only code order with no special treatment of numeric values

- act is an integer from zero (0) to 255, where zero indicates M standard collation, and other integers indicate alternative collation definitions that tie to environment variables of the form gtm_collate_act

- ver is an integer indicating the version of the act collation sequence definition

A "0" return from $$get^%GBLDEF(gname[,reg]) indicates that the global has no special characteristics and uses the region default collation, while a "0,0,0" return indicates that the global is explicitly defined to M collation.

`kill^%GBLDEF(gname)`

kill: returns a TRUE (1) if the current global variable collation now uses M standard collation that is based on code value (ASCII for the codes 0 to 127) ordering except for numeric values, which order most negative to most positive before string values;It returns FALSE (0) if:

- The global has an access method other than BG or MM - typically this means the global is remote and accessed using GT.CM - if so, maintain it on the system that holds it

- The global contains data with collation other than M; use this to function after removing all content

- The global's subscripts span multiple regions; use the global directory (GBLNAME object in GDE) to set collation characteristics for a global whose subscripts span multiple regions

- $TLEVEL is non-zero; always execute this function outside of a TSTART/TCOMMIT fence

`set^%GBLDEF(gname[,nct][,act])`

set: if possible, esablishes a global variable collation; returns a TRUE (1) if the current successful in establishing or matching global variable collation characteristics;returns FALSE (0) if:

- The global has an access method other than BG or MM - typically this means the global is remote and accessed using GT.CM - if so, maintain it on the system that holds it

- The global contains data with collation other than that specified; use this to function after removing all content

- The global's subscripts span multiple regions; use the global directory (GBLNAME object in GDE) to set collation characteristics for a global whose subscripts span multiple regions

- $TLEVEL is non-zero; always execute this function outside of a TSTART/TCOMMIT fence

Input Variables

Of the following input arguments only the second and third are valid for set, while the first is required for all three functions:

- first argument: name of the global; must be a legal M variable name, including the leading caret (^); if the global name appears as a literal, it must be enclosed in quotation marks (" ")

- Optional second argument: truth value where a FALSE (0) indicates numeric values collate before non-numeric values and a TRUE (1) value indicates all keys collate in code order.

- Optional third argument: integer value of a collation definition, where zero (0) selects the standard collation, which requires no special definition and other numbers n up to 255 select collations that tie to environment variables of the form gtm_collate_n

- Optional second argument for get: region name; the funtion determines the region name if you don't supply it

For more information and usage examples, refer to "Using the %GBLDEF Utility" (page 552) in the Internationalization Chapter of this manual.

%LCLCOL

The %LCLCOL utility deals with the adoption and reporting of a local variable collation other than the default M collation that is based on code value (ASCII for the codes 0 to 127) ordering except for numeric values, which order most negative to most positive before string values. As described in the Internationalization Chapter, alternative collation definition requires supplying routines that transform codes into desired collation order. All subscripted local variables for a process must use the same collation sequence. The collation sequence used by local variables can be established as a default or by this utility within the current process. The local collation sequence can only be changed when a process has no subscripted local variables defined.

To establish a default local collation sequence prior to invoking a GT.M process, provide a numeric value to the environment variable gtm_local_collate to select one of the collation tables.

Utility Labels

```
get^%LCLCOL[()]
```

get: returns an integer value associated with the current local variable collation.

```
getncol^%LCLCOL[()]
```

getncol: returns a FALSE (0) if the current local variable collation uses the original GT.M collation of an empty string ("NULL) value between numeric values and strings, and a TRUE (1) value indicates the proces is using the M standard collation where the empty string collates before all other values.

```
getnct^%LCLCOL[()]
```

getnct: returns a FALSE (0) if the current local variable collation follows the M standard of collating numeric values before all non-empty strings and a TRUE (1) if it uses only code order with no special treatment of numeric values.

```
set^%LCLCOL([colid][,ncol][,nct])
```

set: if possible, esablishes a local variable collation.

If the set invocation supplies no argument, there is no environment variable defined corresponding to the first argument, or the specification requests a change when subscripted local variables already exist, the extrinsic function returns a FALSE (0). If the function successfully modifies the local variable collation, or the specification matches the current settings, the function returns a TRUE (1).

Input Variables

The following input arguments are valid only for set:

- Optional first argument: integer value of a collation definition, where zero (0) selects the standard collation, which requires no special definition and other numbers n up to 255 select collations that tie to environment variables of the form gtm_collate_n

- Optional second argument: truth value where a FALSE (0) indicates numeric values collate before non-numeric values and a TRUE (1) value indicates all keys collate in code order.

- Optional third argument: truth value where a FALSE (0) indicates original GT.M collation of an empty string ("NULL) value between numeric values and strings and a TRUE (1) value indicates M standard collation where the empty string collates before all other values. Empty string use as a subscript in local variables is also subject to the VIEW and $VIEW() "LVNULLSUBS" keyword and its "NO" and "NEVER" variations.

For more information and usage examples, refer to "Establishing A Local Collation Sequence" (page 545)in the Internationalization Chapter of this manual.

%PATCODE

The %PATCODE utility is a wrapper for pattern code related VIEW "PATCODE":code" and $VIEW("PATCODE"), and has the following extrinsic entry points:

Utility Labels

```
get^%PATCODE[()]
```

get: returns the name of the process current pattern table; for example the default table is "M"

```
set^%PATCODE(tablename)
```

set: updates the current process pattern table to the named table

If the set invocation finds the character set is UTF-8, it returns FALSE (0) if the argument is missing or does not correspond to an existing file, the function returns the empty string; it successfuly changes the pattern table or matches the current table, the function returns TRUE (1).

Input Variables

The following input argument is valid only for set:

- First argument: name of a pattern table; except for "M", this can be either a file specification or the name of an evironment variable that contains the file specification

System Management Utilities

The System Management utilities are:

%DUMPFHEAD

The %DUMPFHEAD utility provides a programmatic interface to the functionality of MUPIP DUMPFHEAD. This routine reads the database file header directly, rather than opening it as a database and reading values mapped into memory. This means it is

lighter weight in some seneses than ^%PEEKBYNAME, but it also means that the information it retrieves is more limited, and possibly less current.

Utility Labels

getfields^%DUMPFHEAD(fldarray,dbname) : Retrieve the file header fields provided by the MUPIP DUMPFHEAD command for the database file specified by the second parameter into the array passed by reference to the first parameter.

The first parameter is a required pass-by-reference variable that the caller uses to retrieve data.

The second parameter is the path and name for the database file on which to report information.

The format of the output array is fdump(sgmnt_data.<FIELD NAME>)=<value>; refer to the ^%PEEKBYNAME and its documentation for additional information on the names and values.

The $ETRAP handler simply QUITs as it defers error handling to the caller. Application developers should define an appropriate $ETRAP prior to calling %DUMPFHEAD.

Example:

```
$gtm -run ^%XCMD 'do getfields^%DUMPFHEAD(.fields,"mumps.dat") zwrite fields'
```

%FREECNT

The %FREECNT utility displays the number of free and total blocks in the database files associated with all regions in the current global directory.

Example:

```
GTM>DO ^%FREECNT
Region          Free     Total          Database file
------          ----     -----          -------------
DEFAULT           81       100 ( 81.0%) /home/gtmuser1/.fis-gtm/V5.4-002B_x86/g/gtm.dat
GTM>
```

%XCMD

The ^%XCMD utility XECUTEs input from the shell command line and returns any error status (truncated to a single byte on UNIX) generated by that code.

LOOP%XCMD Utility Label

```
LOOP^%XCMD [--before=/<XECUTE_code>/] --xec=/<XECUTE_code>/
▶ [--after=/<XECUTE_code>/]
```

▼

LOOP^%XCMD: XECUTEs the arguments specified in --xec=/arg1/ as GT.M code for each line of standard input that it reads. The currently read line is stored in the variable %l; its line number is stored in %NR (starts from 1). It returns any error status (truncated to a single byte on UNIX) generated by that code.

--before=/arg0/ specifies the GT.M code that LOOP^%XCMD must XECUTE before executing --xec.

--after=/arg2/ specifies the GT.M code that LOOP^%XCMD must XECUTE after executing the last --xec.

For all qualifiers, always wrap GT.M code specified two forward slashes (/) to denote the start and end of the GT.M code. FIS strongly recommends enclosing the GT.M code within single quotation marks to prevent inappropriate expansion by the shell. LOOP^%XCMD's command line parser ignores these forward slashes.

Example:

```
/usr/lib/fis-gtm/V5.4-002B_x86/gtm -run %XCMD 'write "hello world",!'
```

produces the following output:

```
"hello world"
```

Example:

```
$ ps -ef | $gtm_exe/mumps -run LOOP^%XCMD --before='/set user=$ztrnlnm("USER") write "Number of processes owned by
▶ ",user," : "/' --xec='/if %l[user,$increment(x)/' --after='/write x,\!/'
Number of processed owned by jdoe: 5
$
$ cat somefile.txt | $gtm_exe/mumps -run LOOP^%XCMD --before='/write "Total number of lines : "/' --xec='/set
▶ total=$increment(x)/' --after='/write total,\!/'
Total number of lines: 9
$
$ cat somefile.txt | $gtm_exe/mumps -run LOOP^%XCMD --after='/write "Total number of lines : ",%NR,\!/'
Total number of lines: 9
$
$ $gtm_exe/mumps -run LOOP^%XCMD --before='/set f="somefile.txt" open f:readonly use f/' --after='/use $p write
 "Total
▶ number of lines in ",f,": ",%NR,\!/'
Total number of lines in somefile.txt: 9
$
```

▼

%PEEKBYNAME()

%PEEKBYNAME() provides a stable interface to $ZPEEK() that uses control structure field names. $ZPEEK() provides a read-only mechanism to access selected fields in selected control structures in the address space of a process, including process private memory, database shared memory segments and Journal Pools. Although application code can call $ZPEEK() directly, such direct access must use numeric arguments that can vary from release to release. Access by name using %PEEKBYNAME makes application code more stable across GT.M releases. For more information, refer to "$ZPEEK()" (page 275). FIS intends to maintain the stability of a name from release to release where that name refers to the same data item; however we may add or obsolete names, or modify the type and size associated with existing names at our discretion, to reflect changes in the implementation. The format of the %PEEKBYNAME() function is:

```
%PEEKBYNAME(field[,regindex][,format])
```

- The first expression specifies the memory location to access in the format: **CONTROL_BLOCK[.FIELD].*** (For example, **"gd_region.max_key_size"**).

- The optional second expression specifies a region name, structure index or a base address associated with the first (field name) argument. The choice is governed by the following rules applied in the following order:

  1. If the value is a hex value in the form of **0xhhhhhhhh[hhhhhhhh]**, then PEEKBYNAME uses it as the base address of the data to fetch. Also in this case, the offset, length, and type are taken from the field specified in the first expression (field). For more information, see the description of the "PEEK" mnemonic in "$ZPEEK()" (page 275).

2. If the first expression refers to one of the region-related structures supported by the $ZPEEK() function, PEEKBYNAME treats this second expression as a region name.

3. If the first expression refers to one of the replication related structures supported by the $ZPEEK() function that are indexed, PEEKBYNAME treats this second expression as a numerical (base 10) index value.

4. For those structures supported by the $ZPEEK() function that do not accept an argument, this second expression must be NULL or not specified.

- The optional third expression specifies the output format in one character as defined in the "format" argument in the $ZPEEK() documentation. This argument overrides the automatic format detection by the %PEEKBYNAME utility.

Examples:

```
; Print the maximum key size for the DEFAULT region
GTM>write $$^%PEEKBYNAME("gd_region.max_key_size","DEFAULT")
64
; Print the journaling state (non-zero value means it is on)
GTM>write $$^%PEEKBYNAME("gd_region.jnl_state","DEFAULT")
0
; Save the base address of the database file header
GTM>set base=$$^%PEEKBYNAME("sgmnt_addrs.hdr","DEFAULT")
; Print the file header label
GTM>write $$^%PEEKBYNAME("sgmnt_data.label",base)
GDSDYNUNX03
```

LISTALL^%PEEKBYNAME

Prints all the field mnemonics acceptable as the first argument to %PEEKBYNAME().

LIST^%PEEKBYNAME(.output)

Populates output variable with the type and size information indexed by the field mnemonics for all %PEEKBYNAME()-acceptable fields. For example, **output("gd_region.jnl_file_name")="unsigned-char,256"**.

Labels for Selected Fields

Below are selected fields for which you may find %PEEKBYNAME to be a useful alternative to running a DSE or MUPIP command in a PIPE device, and parsing the output. If there is a field that you wish to access using %PEEKBYNAME, please send questions to your GT.M support channel. We will get you an answer, and if it seems to us to be of general interest, we will add it to the %PEEKBYNAME user documentation.

Region Parameters

Calls to %PEEKBYNAME with the listed string as value of the first parameter, and the region name as the value of the second parameter, return the value. For example:

```
GTM>write $$^%PEEKBYNAME("sgmnt_data.n_bts","DEFAULT") ; How many global buffers there are
1000
GTM>write $$^%PEEKBYNAME("node_local.wcs_active_lvl","DEFAULT") ; How many of them are dirty
0
GTM>for i=1:1:10000 set ^x($$^%RANDSTR(8))=$$^%RANDSTR(64)
```

```
GTM>write $$^%PEEKBYNAME("node_local.wcs_active_lvl","DEFAULT") ; And now, how many of them are dirty
377
GTM>
```

When using the following, remember to write code that allows for values other than those listed, e.g., if writing code to check whether before image journaling is in use, make sure it can deal with values other than 0 and 1, because a future release of GT.M can potentially introduce a new return value for a field.

Parameter	^%PEEKBYNAME() Parameter	Value
Asynchronous I/O	**"sgmnt_data.asyncio"**	TRUE (1) if the region has asynchronous I/O enabled and FALSE (0) if it does not
Block size	**"sgmnt_data.blk_size"**	Integer number of bytes
Commit wait spin count	**"sgmnt_data.wcs_phase2_commit_wait_spincnt"**	Integer count
Current transaction	**"sgmnt_data.trans_hist.curr_tn"**	Integer count
Defer allocate	**"sgmnt_data.defer_allocate"**	Integer - 1 means DEFER_ALLOCATE, 0 means NODEFER_ALLOCATE
Encryption key hash	**"sgmnt_data.encryption_hash"**	String of binary bytes
Extension size	**"sgmnt_data.extension_size"**	Integer number of blocks
Flush trigger	**"sgmnt_data.flush_trigger"**	Integer number of blocks (not meaningful for MM)
Journal align size	**"sgmnt_data.alignsize"**	Integer number of bytes
Journal autoswitch limit	**"sgmnt_data.autoswitchlimit"**	Integer number of bytes for maximum size of each journal file
Journal before imaging	**"sgmnt_data.jnl_before_image"**	Integer - 1 means BEFORE image journaling, 0 means NOBEFORE (meaningful only if journaling is on)
Journal buffer size	**"sgmnt_data.jnl_buffer_size"**	Integer number of journal buffers
Journal epoch interval	**"sgmnt_data.epoch_interval"**	Integer number of seconds
Journal next write offset	**"jnl_buffer.dskaddr"**	Integer number of bytes from beginning of journal file
Journal next epoch time	**"jnl_buffer.next_epoch_time"**	Integer number of seconds since January1, 1970 00:00:00 UTC

Parameter	^%PEEKBYNAME() Parameter	Value
Journal state	**"sgmnt_data.jnl_state"**	Integer 0 means disabled, 1 means enabled but off, 2 means on
Journal SYNCIO	**"sgmnt_data.jnl_sync_io"**	Integer - 1 means SYNC_IO, 0 means NOSYNC_IO
Journal yield limit	**"sgmnt_data.yield_lmt"**	Integer count
Lock space	**"sgmnt_data.lock_space_size"**	Integer number of bytes
Maximum key size	**"sgmnt_data.max_key_size"**	Integer number of bytes
Maximum record size	**"sgmnt_data.max_rec_size"**	Integer number of bytes
Mutex hard spin count	**"sgmnt_data.mutex_spin_parms.mutex_hard_spin_count"**	Integer count
Mutex sleep spin count	**"sgmnt_data.mutex_spin_parms.mutex_sleep_spin_count"**	Integer count
Number of global buffers (dirty)	**"node_local.wcs_active_lvl"**	Integer count
Number of global buffers (total)	**"sgmnt_data.n_bts"**	Integer count
Number of processes accessing the database	**"node_local.ref_cnt"**	Integer count (always greater than zero, because the process running %PEEKBYNAME has the database open)
Region open	**"gd_region.open"**	Boolean
Region replication sequence number	**"sgmnt_data.reg_seqno"**	Integer count
Spanning nodes absent	**"sgmnt_data.span_node_absent"**	Integer - 1 means that no global variable nodes span multiple database blocks, 0 means GT.M does not know (in the past, at least one global variable node spanned multiple blocks, but it may since have been overwritten or KILL'd)
Write errors	**"sgmnt_data.wcs_wterror_invoked_cntr"**	Integer count of errors trying to write database blocks - barring problems with the storage subsystem, hitting an asynchronous write limit constitute the primary (probably only) cause

Parameter	^%PEEKBYNAME() Parameter	Value
Writes in progress	**"node_local.wcs_wip_lvl"**	Integer count of of blocks for which GT.M has issued writes that have not yet been recognized as complete
Writes per flush	**"sgmnt_data.n_wrt_per_flu"**	Integer count of database blocks written each time a process encounters a circumstance calling for a flush

Replication Parameters

Calls to %PEEKBYNAME with the listed parameter as the first or only parameter return replication fields as described. For example:

```
GTM>write $$^%PEEKBYNAME("repl_inst_hdr.inst_info.this_instname") ; Name of this instance
Collegeville
GTM>write $$^%PEEKBYNAME("gtmsource_local_struct.secondary_instname",0) ; Name of instance in slot 0 of
 replication
▸ instance file
Malvern
GTM>set x=$$^%PEEKBYNAME("jnlpool_ctl_struct.jnl_seqno") ; Sequence number in Journal Pool of Collegeville
GTM>set y=$$^%PEEKBYNAME("gtmsource_local_struct.read_jnl_seqno",0) ; Next sequence number to send to Malvern
GTM>write x-y ; Current replication backlog from Collegeville to Malvern
2
GTM>
```

▼

Replication Parameter	^%PEEKBYNAME() Parameter	Value
Instance Freeze	**"jnlpool_ctl_struct.instfreeze_environ_inited"**	Integer 1 means Instance Freeze is enabled; 0 means Instance Freeze is disabled
Journal sequence number	**"jnlpool_ctl_struct.jnl_seqno"**	Integer
Journal sequence number to send to receiving instance in replication file slot	**"gtmsource_local_struct.read_jnl_seqno",***i* where *i* is the slot number in the replication instance file	Integer
Name of receiving instance in replication instance file slot	**"gtmsource_local_struct.secondary_instname",***i* where *i* is the slot number in the replication instance file	String of text
Name of this instance	**"repl_inst_hdr.inst_info.this_instname"**	String of text
Supplementary Replication	**"repl_inst_hdr.is_supplementary"**	Integer - 1 means supplementary instance; 0 means not supplementary instance

Replication Parameter	^%PEEKBYNAME() Parameter	Value
Updates disabled	**"jnlpool_ctl_struct.upd_disabled"**	Integer - 1 means updates disabled; 0 means updates permitted

> **Note**
>
> %PEEKBYNAME opens the -READ_ONLY help database as part of its operation, and in so doing causes each process using it to create a private semaphore. In addition, the first process to access a database creates an ftok related semaphore, which in the case of a -READ_ONLY database remains until deleted by a MUPIP RUNDOWN.

%YGBLSTAT()

Labels in the ^%YGBLSTAT utility program gather and report statistics, offering both a high level API and a low level API. While we intend to preserve backward compatibility of the high level API in future GT.M releases, we may change the low level API if and, when, we change the underlying implementation. A call to a label in ^%YGBLSTAT does not in any way slow the execution of other processes. Because the gathering of statistics is not instantaneous, and processes concurrently open database files as well as close them on exit and may turn their participation in statistics monitoring on and off, statistics typically do not show a single moment in time, as they change during the short time interval over which they are gathered.

In the following, an omitted response or argument is equivalent to "*".

The high level API implemented by **$$STAT^%YGBLSTAT(expr1[,expr2[,expr3[,expr4]]])** reports global variable statistics and has arguments as follows:

- expr1 (treated as an intexpr - coercing an expr to an integer is equivalent to +(expr)) specifies the PID of a process on which to report; if such a process does not exist, has not opted in, or no database file mapped by expr3 and expr4 includes statistics for such a process, the function returns an empty string. Specifying "*" as the value of expr1 returns the aggregate statistic(s) specified by expr2 for all processes whose statistics are included in the database file(s) of the region(s) specified by expr4 within the global directory specified by expr3, or the empty string if there are no statistics to report for any process.

- expr2 specifies the statistic(s) to report as follows:

  - If expr2 is a single statistic, for example, "LKF", the function returns the requested value as an integer

  - If expr2 is a series of comma-separated names of statistics, for example., "GET,DTA", the function returns a string with each requested statistic in ZSHOW "G" order, for example, "GET:3289,DTA:598...", rather than in the order in which they appear within the specifying argument.

  - If expr2 is omitted, or consists of the string "*", the return value reports all statistics formatted like the ZSHOW "G" statistics for a single region, for example, "SET:563,KIL:39,GET:3289,DTA:598...

  - The ^%YGBLSTAT utility uses the local variable structure %YGS. You application code may hide %YGS with NEW but must not modify nodes within it.

- expr3 specifies a global directory file name (producing a ZGBLDIRACC error if such a global directory is not accessible); if unspecified, the utility defaults this value to $ZGBLDIR of the invoking process.

- expr4 specifies the name of a region (producing a NOREGION error if no such region exists in the global directory expr3); if expr4 is unspecified, or the string "*", the function returns statistics for the process or processes summed across all regions of the global directory explicitly or implicitly specified by expr3.

When invoked as an interactive utility program using DO, ^%YGBLSTAT, prompts for:

- the process id (respond * for all processes).

- a comma separated list of the statistics desired (respond * for all statistics).

- the global directory to use.

- region (respond * to report statistics summed across all regions).

When invoked from a shell, the command line is:

```
$ mumps -run %YGBLSTAT [--help] [--pid pidlist] [--reg reglist]
▶ [--stat statlist]
```

▼

where:

- pidlist is a single pid, or "*" (quoted to protect it from expansion by the shell) for all processes currently sharing statistics.

- reglist is a single region name in the global directory specified by $gtmgbldir, or "*" to report statistics summed across all regions.

- statlist is one or more comma separated statistics, or "*".

- When statlist specifies a list of statistics, %YGBLSTAT reports them in the same order in which ZSHOW "G" reports those statistics, rather than in the order in which they appear within the specifying argument.

$$IN^%YGBLSTAT(expr1[,expr2[,expr3]]) reports whether a process that has opted in. There is an assumption that only GT.M processes opt in. Its arguments are as follows:

- expr1 coerced to an intexpr specifies a PID. If the PID is invalid or has not opted in, the function returns an empty string, which coerces to FALSE if treated as a Boolean.

- expr2 specifies a global directory file name (producing a ZGBLDIRACC error if such a global directory is not accessible); if unspecified or the empty string, the utility defaults this value to the $ZGBLDIR of the invoking process.

- expr3 evaluates to the name of a region (producing a NOREGION error if no such region exists in the global directory specified by expr2); if expr3 is unspecified, or the string "*", the function returns whether the process has opted in to any region of the global directory specified by expr2.

$$ORDERPID^%YGBLSTAT(expr1[,expr2[,expr3]]) reports PIDs of processes that have opted in and recorded statistics. Its arguments are as follows:

- expr1 coerced to an intexpr specifies a PID such that the function returns the next PID after expr1 of a process that has opted in to be monitored and which has recorded statistics in any region(s) specified by expr3 from the global directory specified by expr2, or the empty string if expr1 is the last PID. A value of the empty string ("") for expr1 returns the first monitored PID meeting the specifications in expr2 and expr3.

- expr2 specifies a global directory file name (producing a ZGBLDIRACC error if such a global directory is not accessible); if unspecified or the empty string, the utility defaults this value to the $ZGBLDIR of the invoking process.

- expr3 evaluates to the name of a region (producing a NOREGION error if no such region exists in the global directory specified by expr2); of expr3 is unspecified, or the string "*", the function returns the PID for the next process after expr1 for any region of the global directory specified by expr2.

- Applications should not rely on GT.M returning the PIDs in a sorted or other predictable order: the order in which PIDs are returned is at the discretion of the implementation, and may change from release to release.

The low level API implemented by $$SHOW^%YGBLSTAT(glvn[,strexp]) reports raw statistics of a process and has arguments as follows:

- glvn specifies a node containing raw statistics for a process

- the raw data is stored in uniquely managed database files as ^%YGS(expr1,expr2) where:

  - expr1 evaluates to the name of a region in the current global directory (or the global directory of an extended reference), producing an UNDEF error, or, in NOUNDEF mode, an empty string, if no such region exists

  - expr2 coerced to an intexpr is a PID.

  - The data in the node is a series of binary bytes which are the raw statistics shared by a process

- strexp specifies statistics to report with the same interpretation as the expr2 parameter of $$STAT^%YGBLSTAT.

- $$SHOW^%YGBLSTAT() reports a zero value for any statistic whose name is unrecognized. This facilitates application code written for a version of GT.M that includes a statistic, but which also needs to run on an earlier version without that statistic

- Because a process sharing statistics can exit, deleting its node, between the time a monitoring process decides to access its statistics, e.g., finding it using $$ORDERPID^%YGBLSTAT() or $ORDER(^%YGS()), and the time the monitoring process performs the database access, any direct access to ^%YGBLSTAT should be wrapped in $GET().

- As raw statistics are binary data, processes in UTF-8 mode that gather and monitor statistics should use code with appropriate BADCHAR handling. Note that processes sharing statistics and processes gathering statistics for monitoring and reporting need not run in the same UTF-8/M mode. Statistics sharing by processes is identical in M and UTF-8 modes. FIS suggests that processes gathering statistics run in M mode

FIS strongly recommends that except as documented here for sharing and gathering statistics, you not access statistics database files except under the direction of your GT.M support channel.

As they do for unshared statistics, shared statistics reflect all database actions for a TP transaction, including those during RESTARTs. Because the sharing of statistics is not a database operation that modifies the relative time stamps GT.M uses to maintain serialized operation preserving the Consistency and Isolation aspects of ACID operation, statistics generated by a sharing process inside a transaction (TSTART/TCOMMIT) do not cause transaction restarts as a consequence of updates to shared statistics by other processes.

Example:

```
GTM>view "statshare"
GTM>for i=1:1:10 set ^A(i)=i
GTM>write $$STAT^%YGBLSTAT("*","","","DEFAULT") ; returns all statistics for the DEFAULT region

 SET:10,KIL:0,GET:1,DTA:0,ORD:0,ZPR:0,QRY:0,LKS:0,LKF:0,CTN:100030,DRD:5,DWT:4,NTW:10,NTR:1,NBW:13,NBR:22,NR0:0,NR1:0,
GTM>
```

UTF-8 Mode Utility Routines

The %UTF2HEX and %HEX2UTF M utility routines provide conversions between UTF-8 and hexadecimal code-point representations. Both these utilities run in only in UTF-8 mode; in M mode, they both trigger a run-time error.

%UTF2HEX

The GT.M %UTF2HEX utility returns the hexadecimal notation of the internal byte encoding of a UTF-8 encoded GT.M character string. This routine has entry points for both interactive and non-interactive use.

DO ^%UTF2HEX converts the string stored in %S to the hexadecimal byte notation and stores the result in %U.

DO INT^%UTF2HEX converts the interactively entered string to the hexadecimal byte notation and stores the result in %U.

$$FUNC^%UTF2HEX(s) returns the hexadecimal byte representation of the character strings.

Example:

```
GTM>write $zchset
UTF-8
GTM>SET %S=$CHAR($$FUNC^%HD("0905"))_$CHAR($$FUNC^%HD("091A"))_$CHAR($$FUNC^%HD(
"094D"))_$CHAR($$FUNC^%HD("091B"))_$CHAR($$FUNC^%HD("0940"))
GTM>zwrite
%S="अच्छी"
GTM>DO ^%UTF2HEX
GTM>zwrite
%S="अच्छी"
%U="E0A485E0A49AE0A58DE0A49BE0A580"
GTM>write $$FUNC^%UTF2HEX("ABC")
414243
GTM>
```

Note that %UTF2HEX provides functionality similar to the UNIX binary dump utility (od -x).

%HEX2UTF

The GT.M %HEX2UTF utility returns the GT.M encoded character string from the given bytestream in hexadecimal notation. This routine has entry points for both interactive and non-interactive use.

DO ^%HEX2UTF converts the hexadecimal byte stream stored in %U into a GT.M character string and stores the result in %S.

DO INT^%HEX2UTF converts the interactively entered hexadecimal byte stream into a GT.M character string and stores the result in %S.

$$FUNC^%HEX2UTF(s) returns the GT.M character string specified by the hexadecimal character values in s (each character is specified by its Unicode® code point).

Example:

```
GTM>set u="E0A485" write $$FUNC^%HEX2UTF(u)
अ
GTM>set u="40E0A485" write $$FUNC^%HEX2UTF(u)
@अ
GTM>
```

GT.M Utilities Summary Table

GT.M Utilities Summary	
UTILITY NAME	**DESCRIPTION**
%D	Displays the current date in [d]d-mmm-[yy]yy format.
%DATE	Converts input date to $HOROLOG format.
%DH	Converts decimal numbers to hexadecimal.
%DO	Converts decimal numbers to octal.
%EXP	Raises number to the power of another number.
%FL	Lists comment lines at the beginning of the source programs.
%FREECNT	Displays the number of free blocks in the database files associated with the current global directory.
%G	Displays global variables and their values.
%GBLDEF	Manipulates the collation sequence assigned to a global.
%GC	Copies a global or global sub-tree.
%GCE	Replaces a specified value or part of a value in a set of global variables.
%GD	Displays existing globals in the current global directory without displaying their values or descendants.
%GED	Provides full-screen editing capabilities for global variables and values.
%GI	Enters global variables and their values from a sequential file into a database.
%GO	Copies globals from the current database to a sequential output file.
%GSE	Displays global variables and their values when the values contain a specified string or number.
%GSEL	Selects globals by name.
%H	Converts date and time to and from $HOROLOG format.
%HD	Converts hexadecimal numbers to decimal.
%HEX2UTF	Converts the given bytestream in hexadecimal notation to GT.M encoded character string.
%HO	Converts hexadecimal numbers to octal.
%JSWRITE	Converts a GT.M glv structure to a string of JS objects.
%LCASE	Converts a string to all lower case.
%LCLCOL	Manipulates the collation sequence assigned to local variables.
%OD	Converts octal numbers to decimal.
%OH	Converts octal numbers to hexadecimal.
%PATCODE	Loads pattern definition files for use within an active database.
%RCE	Replaces every occurrence of a text string with another string in a routine or list of routines.
%RD	Lists routine names available through your $ZROUTINES search list.

GT.M Utilities Summary	
UTILITY NAME	**DESCRIPTION**
%RI	Transfers routines from ANSI sequential format into individual .m files in GT.M format.
%RO	Writes M routines in ANSI transfer format.
%RSE	Searches for every occurrence of a text string in a routine or a list of routines.
%RSEL	Selects M routines and places their directories and names in a local array.
%SQROOT	Calculates the square root of a number.
%T	Displays the current time in [h]h:mm AM/PM format.
%TI	Converts time to $HOROLOG format.
%TO	Converts the current time from $HOROLOG format to [h]h:mm AM/PM format.
%UCASE	Converts a string to all upper case.
%UTF2HEX	Converts UTF-8 encoded GT.M character string to bytestream in hexadecimal notation.

Chapter 11. Integrating External Routines

Revision History		
Revision V6.3-014	06 October 2020	• In "Examples of Using External Calls" (page 531), reorder prototypes to match order in the header
Revision V6.3-012	08 April 2020	• In "gtmxc_types.h" (page 534), Note about 1 MiB limit on strings
Revision V6.3-009	27 June 2019	• In "Using External Calls" (page 523), Add null address and negative length handling
Revision V6.3-006	26 October 2018	• In "Using External Calls" (page 523), added SIGSAFE keyword • In "Pre-allocation of Output Parameters" (page 528), specify that when an external call exceeds its specified preallocation (gtm_string_t * or gtm_char_t * output), GT.M produces an EXCEEDSPREALLOC error. • In "Rules to Follow in Call-Ins" (page 540), specify that using gtm_malloc is mandatory to allocate returns of pointer types to prevent memory leaks.
Revision V6.3-005	29 June 2018	• In "Nested Call-Ins" (page 540), tweak ZGOTO description for GTMCI behavior
Revision V6.3-001	20 March 2017	• In "Building Standalone Programs" (page 539), removed instructions for platforms that are no longer supported. • In "Creating a Shareable Library" (page 523), removed examples for currently unsupported platforms. • In "Using External Calls" (page 523), removed Tru64 and fixed formatting • In "gtmxc_types.h" (page 534), removed Tru64 • In "Relevant files for Call-Ins" (page 534), removed HP-UX
Revision V6.1-000	28 August 2014	• In "Access to Non-M Routines" (page 522), added information about the external call arguments that do not specify a value and corrected the examples. • In "Using External Calls" (page 523), added a note about how arguments that do not specify a value translate to default values in C.

Revision V6.0-003	24 February 2014	• Improved the description of the "Pre-allocation of Output Parameters" [528] section.
		• In "Rules to Follow in Call-Ins" [540], added a recommendation to not mix GT.M device input using read() system service with buffered input services in fgets() family.
		• Added a section called "Type Limits for Call-ins and Call-outs" [541].
Revision V6.0-001	21 March 2013	In "Using External Calls" (page 523) and "gtmxc_types.h" (page 534), added information about V6.0-001 enhancements for gtmxc_types.h.

Introduction

Application code written in M can call application code written in C (or which uses a C compatible call) and vice versa.

> **Note**
>
> This C code shares the process address space with the GT.M run-time library and M application code. Bugs in C code may result in difficult to diagnose failures to occur in places not obviously related to the cause of the failure.

Access to Non-M Routines

In GT.M, calls to C language routines may be made with the following syntax:

```
DO &[packagename.]name[^name][parameter-list]
```

or as an expression element,

```
$&[packagename.]name[^name][parameter-list]
```

Where packagename, like the name elements is a valid M name. Because of the parsing conventions of M, the identifier between the ampersand (&) and the optional parameter-list has precisely constrained punctuation - a later section describes how to transform this into a more richly punctuated name should that be appropriate for the called function. While the intent of the syntax is to permit the name^name to match an M labelref, there is no semantic implication to any use of the up-arrow (^). For more information on M names, labelrefs and parameter-lists, refer toChapter 5: "General Language Features of M" (page 68).

Example:

```
;Call external routine rtn1
DO &rtn1
;Call int^exp in package "mathpak" with one parameter: the expression val/2
DO &mathpak.int^exp(val/2)
;Call the routine sqrt with the value "2"
WRITE $&sqrt(2)
;Call the routine get parms, with the parameter "INPUT" and the variable "inval", passed by reference.
DO &getparms("INPUT",.inval)
```

```
;Call program increment in package "mathpak" without specifying a value for the first argument and the variable
 "outval" passed by reference as the second argument. All arguments which do not specify a value translate to
 default values in the increment program.
Do &mathpak.increment(,.outval)
```

The called routines follow the C calling conventions. They must be compiled as position independent code and linked as a shareable library.

Creating a Shareable Library

The method of creating a shareable library varies by the operating system. The following examples illustrate the commands on an IBM pSeries (formerly RS/6000) AIX system.

Example:

```
$ cat increment.c
int increment(int count, float *invar, float *outvar)
{
    *outvar=*invar+1.0;
    return 0;
}
$ cat decrement.c
int decrement(int count, float *invar, float *outvar)
{
    *outvar=*invar-1.0;
    return 0;
}
```

On IBM pSeries AIX:

Example:

```
$ cc -c -I$gtm_dist increment.c decrement.c
$ ld -o libcrement.so increment.o decrement.o -G -bexpall -bnoentry -bh:4 -lc
```

> **Note**
>
> Refer to the AIX V4.2 documentation of the ld(1) AIX command for information on shareable libraries under AIX V4.2.

On Linux x86:

Example:

```
% gcc -c -fPIC -I$gtm_dist increment.c decrement.c
% gcc -o libcrement.so -shared increment.o decrement.o
```

UsingExternal Calls

The functions in programs increment and decrement are now available to GT.M through the shareable library libcrement.sl or libcrement.so, or though the DLL as libcrement.dll, depending on the specific platform. The suffix .sl is used throughout the following examples to represent .sl, .so, or .dll. Be sure to use the appropriate suffix for your platform.

GT.M uses an "external call table" to map the typeless data of M into the typed data of C, and vice versa. The external call table has a first line containing the pathname of the shareable library file followed by one or more specification lines in the following format:

```
entryref: return-value routine-name (parameter, parameter, ... ) [: SIGSAFE]
```

The optional case-insensitive keyword SIGSAFE following the parameter list specifies the external call does not create its own signal handlers; this allows GT.M to avoid burdensome signal handler coordination for the external call. By default, GT.M saves and restores signal setups for external calls.

entryref is an M entryref, return-value is gtm_long_t, gtm_status_t, or void, and parameters are in the format:

```
direction:type [num]
```

where [num] indicates a pre-allocation value explained later in this chapter.

Legal directions are I, O, or IO for input, output, or input/output, respectively.

The following table describes the legal types defined in the C header file $gtm_dist/gtmxc_types.h:

Type : Usage

void: Specifies that the function does not return a value.

gtm_status_t : Type int. If the function returns zero (0), then the call was successful. If it returns a non-zero value, GT.M will signal an error upon returning to M.

gtm_long_t : 32-bit signed integer on 32-bit platforms and 64-bit signed integer on 64-bit platforms.

gtm_ulong_t : 32-bit unsigned integer on 32-bit platforms and 64-bit signed integer on 64-bit platforms.

gtm_long_t* : For passing a pointer to long [integers].

gtm_float_t* : For passing a pointer to floating point numbers.

gtm_double_t* : Same as above, but double precision.

gtm_char_t*: For passing a "C" style string - null terminated.

gtm_char_t** : For passing a pointer to a "C" style string.

gtm_string_t* : For passing a structure in the form {int length;char *address}. Useful for moving blocks of memory to or from GT.M.

gtm_pointertofunc_t : For passing callback function pointers. For details see "Callback Mechanism" (page 529).

> **Note**
>
> If an external call's function argument is defined in the external call table, GT.M allows invoking that function without specifying a value of the argument. All non-trailing and output-only arguments arguments which do not specify a value translate to the following default values in C:
>
> - All numeric types: 0

- gtm_char_t * and gtm_char_t **: Empty string

- gtm_string_t *: A structure with 'length' field matching the preallocation size and 'address' field being a NULL pointer.

In the mathpak package example, the following invocation translate inval to the default value, that is, 0.

```
GTM>do &mathpak.increment(,.outval)
```

If an external call's function argument is defined in the external call table and that function is invoked without specifying the argument, ensure that the external call function appropriately handles the missing argument. As a good programming practice, always ensure that count of arguments defined in the external call table matches the function invocation.

To protect the process, GT.M turns any return values containing a null pointer to an empty string value and, for the first occurrence in a process, sends one XCRETNULLREF syslog message. If an external call sets a gtm_string length to a negative value, to protect the process, GT.M turns any return with a negative length to an empty string value and, for the first occurrence in a process, sends one XCCONVERT syslog message.

gtmxc_types.h also includes definitions for the following entry points exported from libgtmshr:

```
void gtm_hiber_start(gtm_uint_t mssleep);
void gtm_hiber_start_wait_any(gtm_uint_t mssleep)
void gtm_start_timer(gtm_tid_t tid, gtm_int_t time_to_expir, void (*handler)(), gtm_int_t hdata_len, void
  \*hdata);
void gtm_cancel_timer(gtm_tid_t tid);
```

where:

- mssleep - milliseconds to sleep

- tid - unique timer id value

- time_to_expir - milliseconds until timer drives given handler

- handler - function pointer to handler to be driven

- hdata_len - 0 or length of data to pass to handler as a parameter

- hdata - NULL or address of data to pass to handler as a parameter

gtm_hiber_start() always sleeps until the time expires; gtm_hiber_start_wait_any() sleeps until the time expires or an interrupt by any signal (including another timer). gtm_start_timer() starts a timer but returns immediately (no sleeping) and drives the given handler when time expires unless the timer is canceled.

⚠️ Important

GT.M continues to support xc_* equivalent types of gtm_* for upward compatibility. gtmxc_types.h explicitly marks the xc_* equivalent types as deprecated.

- Parameter-types that interface GT.M with non-M code using C calling conventions must match the data-types on their target platforms. Note that most addresses on 64-bit platforms are 8 bytes long and require 8 byte alignment in structures whereas all addresses on 32-bit platforms are 4 bytes long and require 4-byte alignment in structures.

- Though strings with embedded zeroes are sent as input to external routines, embedded zeroes in output (or return value) strings of type gtm_char_t may cause string truncation because they are treated as terminator.

- If your interface uses gtm_long_t or gtm_ulong_t types but your interface code uses int or signed int types, failure to revise the types so they match on a 64-bit platform will cause the code to fail in unpleasant, potentially dangerous and hard to diagnose ways.

The first parameter of each called routine is an int (for example, int argc in decrement.c and increment.c) that specifies the number of parameters passed. This parameter is implicit and only appears in the called routine. It does not appear in the call table specification, or in the M invocation. If there are no explicit parameters, the call table specification will have a zero (0) value because this value does not include itself in the count. If there are fewer actual parameters than formal parameters, the call is determined from the parameters specified by the values supplied by the M program. The remaining parameters are undefined. If there are more actual parameters than formal parameters, GT.M reports an error.

There may be only a single occurrence of the type gtm_status_t for each entryref.

Database Encryption Extensions to the GT.M External Interface

To support Database Encryption, GT.M provides a reference implementation which resides in $gtm_dist/plugin/gtmcrypt.

The reference implementation includes:

- A $gtm_dist/plugin/gtmcrypt sub-directory with all source files and scripts. The scripts include those needed to build/install libgtmcrypt.so and "helper" scripts, for example, add_db_key.sh (see below).

- The plugin interface that GT.M expects is defined in gtmcrypt_interface.h. Never modify this file - it defines the interface that the plugin must provide.

- $gtm_dist/plugin/libgtmcrypt.so is the shared library containing the executables which is dynamically linked by GT.M and which in turn calls the encryption packages. If the $gtm_dist/utf8 directory exists, then it should contain a symbolic link to ../plugin.

- Source code is provided in the file $gtm_dist/plugin/gtmcrypt/source.tar which includes build.sh and install.sh scripts to respectively compile and install libgtmcrypt.so from the source code.

To support the implementation of a reference implementation, GT.M provides additional C structure types (in the gtmxc_types.h file):

- gtmcrypt_key_t - a datatype that is a handle to a key. The GT.M database engine itself does not manipulate keys. The plug-in keeps the keys, and provides handles to keys that the GT.M database engine uses to refer to keys.

- xc_fileid_ptr_t - a pointer to a structure maintained by GT.M to uniquely identify a file. Note that a file may have multiple names - not only as a consequence of absolute and relative path names, but also because of symbolic links and also because a file system can be mounted at more than one place in the file name hierarchy. GT.M needs to be able to uniquely identify files.

Although not required to be used by a customized plugin implementation, GT.M provides (and the reference implementation uses) the following functions for uniquely identifying files:

- xc_status_t gtm_filename_to_id(xc_string_t *filename, xc_fileid_ptr_t *fileid) - function that takes a file name and provides the file id structure for that file.

- xc_status_t gtm_is_file_identical(xc_fileid_ptr_t fileid1, xc_fileid_ptr_t fileid2) - function that determines whether two file ids map to the same file.

- gtm_xcfileid_free(xc_fileid_ptr_t fileid) - function to release a file id structure.

Mumps, MUPIP and DSE processes dynamically link to the plugin interface functions that reside in the shared library. The functions serve as software "shims" to interface with an encryption library such as libmcrypt or libgpgme / libgcrypt.

The plugin interface functions are:

- gtmcrypt_init()

- gtmcrypt_getkey_by_name()

- gtmcrypt_getkey_by_hash()

- gtmcrypt_hash_gen()

- gtmcrypt_encode()

- gtmcrypt_decode()

- gtmcrypt_close()

- and gtmcrypt_strerror()

A GT.M database consists of multiple database files, each of which has its own encryption key, although you can use the same key for multiple files. Thus, the gtmcrypt* functions are capable of managing multiple keys for multiple database files. Prototypes for these functions are in gtmcrypt_interface.h.

The core plugin interface functions, all of which return a value of type gtm_status_t are:

- gtmcrypt_init() performs initialization. If the environment variable $gtm_passwd exists and has an empty string value, GT.M calls gtmcrypt_init() before the first M program is loaded; otherwise it calls gtmcrypt_init() when it attempts the first operation on an encrypted database file.

- Generally, gtmcrypt_getkey_by_hash or, for MUPIP CREATE, gtmcrypt_getkey_by_name perform key acquisition, and place the keys where gtmcrypt_decode() and gtmcrypt_encode() can find them when they are called.

- Whenever GT.M needs to decode a block of bytes, it calls gtmcrypt_decode() to decode the encrypted data. At the level at which GT.M database encryption operates, it does not matter what the data is - numeric data, string data whether in M or UTF-8 mode and whether or not modified by a collation algorithm. Encryption and decryption simply operate on a series of bytes.

- Whenever GT.M needs to encode a block of bytes, it calls gtmcrypt_encode() to encode the data.

- If encryption has been used (if gtmcrypt_init() was previously called and returned success), GT.M calls gtmcrypt_close() at process exit and before generating a core file. gtmcrypt_close() must erase keys in memory to ensure that no cleartext keys are visible in the core file.

More detailed descriptions follow.

- gtmcrypt_key_t *gtmcrypt_getkey_by_name(gtm_string_t *filename) - MUPIP CREATE uses this function to get the key for a database file. This function searches for the given filename in the memory key ring and returns a handle to its symmetric cipher key. If there is more than one entry for the given filename , the reference implementation returns the entry matching the last occurrence of that filename in the master key file.

- gtm_status_t gtmcrypt_hash_gen(gtmcrypt_key_t *key, gtm_string_t *hash) - MUPIP CREATE uses this function to generate a hash from the key then copies that hash into the database file header. The first parameter is a handle to the key and the second parameter points to 256 byte buffer. In the event the hash algorithm used provides hashes smaller than 256 bytes, gtmcrypt_hash_gen() must fill any unused space in the 256 byte buffer with zeros.

- gtmcrypt_key_t *gtmcrypt_getkey_by_hash(gtm_string_t *hash) - GT.M uses this function at database file open time to obtain the correct key using its hash from the database file header. This function searches for the given hash in the memory key ring and returns a handle to the matching symmetric cipher key. MUPIP LOAD, MUPIP RESTORE, MUPIP EXTRACT, MUPIP JOURNAL and MUPIP BACKUP -BYTESTREAM all use this to find keys corresponding to the current or prior databases from which the files they use for input were derived.

- gtm_status_t gtmcrypt_encode(gtmcrypt_key_t *key, gtm_string_t *inbuf, gtm_string_t *outbuf) and gtm_status_t gtmcrypt_decode(gtmcrypt_key_t *key, gtm_string_t *inbuf, gtm_string_t *outbuf)- GT.M uses these functions to encode and decode data. The first parameter is a handle to the symmetric cipher key, the second a pointer to the block of data to encode or decode, and the third a pointer to the resulting block of encoded or decoded data. Using the appropriate key (same key for a symmetric cipher), gtmcrypt_decode() must be able to decode any data buffer encoded by gtmcrypt_encode(), otherwise the encrypted data is rendered unrecoverable.7 As discussed earlier, GT.M requires the encrypted and cleartext versions of a string to have the same length.

- char *gtmcrypt_strerror() - GT.M uses this function to retrieve addtional error context from the plug-in after the plug-in returns an error status. This function returns a pointer to additional text related to the last error that occurred. GT.M displays this text as part of an error report. In a case where an error has no additional context or description, this function returns a null string.

The complete source code for reference implementations of these functions is provided, licensed under the same terms as GT.M. You are at liberty to modify them to suit your specific GT.M database encryption needs. Check your GT.M license if you wish to consider redistributing your changes to others.

For more information and examples, refer to the Database Encryption Technical Bulletin.

Pre-allocation of Output Parameters

The definition of parameters passed by reference with direction output can include specification of a pre-allocation value. This is the number of units of memory that the user wants GT.M to allocate before passing the parameter to the external routine. For example, in the case of type gtm_char_t *, the pre-allocation value would be the number of bytes to be allocated before the call to the external routine.

Specification of a pre-allocation value should follow these rules:

- Pre-allocation is an unsigned integer value specifying the number of bytes to be allocated on the system heap with a pointer passed into the external call.

- Pre-allocating on a type with a direction of input or input/output results in a GT.M error.

- Pre-allocation is meaningful only on types gtm_char_t * and gtm_string_t *. On all other types the pre-allocation value specified will be ignored and the parameter will be allocated a default value for that type. With gtm_string_t * arguments

make sure to set the 'length' field appropriately before returning control to GT.M. On return from the external call, GT.M uses the value in the length field as the length of the returned value, in bytes.

- If the user does not specify any value, then the default pre-allocation value would be assigned to the parameter.

- Specification of pre-allocation for "scalar" types (parameters which are passed by value) is an error.

> ### ⚠️ Important
>
> Pre-allocation is optional for all output-only parameters except gtm_string_t * and gtm_char_t *. Pre-allocation yields better management of memory for the external call. When an external call exceeds its specified preallocation (gtm_string_t * or gtm_char_t * output), GT.M produces the EXCEEDSPREALLOC error. In the case the user allocates space for the character pointer inside a gtm_string_t * type output parameter, a length field longer than the specified preallocate size for the output parameter does not cause an EXCEEDSPREALLOC error.

Callback Mechanism

GT.M exposes certain functions that are internal to the GT.M runtime library for the external calls via a callback mechanism. While making an external call, GT.M populates and exposes a table of function pointers containing addresses to call-back functions.

Index	Function	Argument	Type	Description
0	hiber_start			sleep for a specified time
		slp_time	integer	milliseconds to sleep
1	hiber_start_wait_any			sleep for a specified time or until any interrupt, whichever comes first
		slp_time	integer	milliseconds to sleep
2	start_timer			start a timer and invoke a handler function when the timer expires
		tid	integer	unique user specified identifier for this timer
		time_to_expire	integer	milliseconds before handler is invoked
		handler	pointer to function	specifies the entry of the handler function to invoke
		hlen	integer	length of data to be passed via the hdata argument
		hdata	pointer to char	data (if any) to pass to the handler function
3	cancel_timer			stop a timer previously started with start_timer(), if it has not yet expired
		tid	integer	unique user specified identifier of the timer to cancel

Index	Function	Argument	Type	Description
4	gtm_malloc			allocates process memory from the heap
		<return-value>	pointer to void	address of the allocated space
		space_needed	32-bit platforms: 32-bit unsigned integer 64-bit platforms: 64-bit unsigned integer	bytes of space to allocate. This has the same signature as the system malloc() call.
5	gtm_free			return memory previously allocated with gtm_malloc()
		free_address	pointer to void	address of the previously allocated space

The external routine can access and invoke a call-back function in any of the following mechanisms:

- While making an external call, GT.M sets the environment variable GTM_CALLIN_START to point to a string containing the start address (decimal integer value) of the table described above. The external routine needs to read this environment variable, convert the string into an integer value and should index into the appropriate entry to call the appropriate GT.M function.

- GT.M also provides an input-only parameter type gtm_pointertofunc_t that can be used to obtain call-back function pointers via parameters in the external routine. If a parameter is specified as I:gtm_pointertofunc_t and if a numeric value (between 0-5) is passed for this parameter in M, GT.M interprets this value as the index into the callback table and passes the appropriate callback function pointer to the external routine.

> **Note**
>
> FIS strongly discourages the use of signals, especially SIGALARM, in user written C functions. GT.M assumes that it has complete control over any signals that occur and depends on that behavior for recovery if anything should go wrong. The use of exposed timer APIs should be considered for timer needs.

Limitations on the External Program

Since both GT.M runtime environment and the external C functions execute in the same process space, the following restrictions apply to the external functions:

1. GT.M is designed to use signals and has signal handlers that must function for GT.M to operate properly. The timer related call-backs should be used in place of any library or system call which uses SIGALRM such as sleep(). Use of signals by external call code may cause GT.M to fail.

2. Use of the GT.M provided malloc and free, creates an integrated heap management system, which has a number of debugging tools. FIS recommends the usage of gtm_malloc/gtm_free in the external functions that provides better debugging capability in case memory management problems occur with external calls.

3. Use of exit system call in external functions is strongly discouraged. Since GT.M uses exit handlers to properly shutdown runtime environment and any active resources, the system call _exit should never be used in external functions.

4. GT.M uses timer signals so often that the likelihood of a system call being interrupted is high. So, all system calls in the external program can return EINTR if interrupted by a signal.

5. Handler functions invoked with start_timer must not invoke services that are identified by the Operating System documentation as unsafe for signal handlers (or not identified as safe) - consult the system documentation or man pages for this information. Such services cause non-deterministic failures when they are interrupted by a function that then attempts to call them, wrongly assuming they are reentrant.

Examples of Using External Calls

```
foo: void bar (I:gtm_float_t*, O:gtm_float_t*)
```

There is one external call table for each package. The environment variable "GTMXC" must name the external call table file for the default package. External call table files for packages other than the default must be identified by environment variables of the form "GTMXC_name".

The first of the external call tables is the location of the shareable library. The location can include environment variable names.

Example:

```
% echo $GTMXC_mathpak
/user/joe/mathpak.xc
% echo lib /usr/
% cat mathpak.xc
$lib/mathpak.sl
exp: gtm_status_t xexp(I:gtm_float_t*, O:gtm_float_t*)
% cat exp.c
...
int xexp(count, invar, outvar)
int count;
float *invar;
float *outvar;
      {
        ...
      }
% gtm
...
GTM>d &mathpak.exp(inval,.outval)
GTM>
```

Example : For preallocation:

```
% echo $GTMXC_extcall
/usr/joe/extcall.xc
% cat extcall.xc
/usr/lib/extcall.sl
prealloc: void gtm_pre_alloc_a(O:gtm_char_t *[12])
% cat extcall.c
#include <stdio.h>
#include <string.h>
#include "gtmxc_types.h"
void gtm_pre_alloc_a (int count, char *arg_prealloca)
{
```

```
    strcpy(arg_prealloca, "New Message");
    return;
}
```

Example : for call-back mechanism

```
% echo $GTMXC
/usr/joe/callback.xc
% cat /usr/joe/callback.xc
$MYLIB/callback.sl
init:     void    init_callbacks()
tstslp:  void    tst_sleep(I:gtm_long_t)
strtmr: void     start_timer(I:gtm_long_t, I:gtm_long_t)
% cat /usr/joe/callback.c
#include <stdio.h>
#include <stdlib.h>

#include "gtmxc_types.h"

void **functable;
void (*sleep_uninterrupted)(int );
void (*sleep_interrupted)(int );
void (*setup_timer)(int , int , void (*)() , int , char *);
void (*cancel_timer)(int );
void* (*malloc_fn)(int);
void (*free_fn)(void*);

void  init_callbacks (int count)
{
   char *start_address;

   start_address = (char *)getenv("GTM_CALLIN_START");

   if (start_address == (char *)0)
   {
          fprintf(stderr,"GTM_CALLIN_START is not set\n");
     return;
   }
   functable = (void **)atoi(start_address);
   if (functable == (void **)0)
   {
     perror("atoi : ");
     fprintf(stderr,"addresses defined by GTM_CALLIN_START not a number\n");
     return;
   }
   sleep_uninterrupted = (void (*)(int )) functable[0];
   sleep_interrupted = (void (*)(int )) functable[1];
   setup_timer = (void (*)(int , int, void (*)(), int, char *)) functable[2];
   cancel_timer = (void (*)(int )) functable[3];

   malloc_fn = (void* (*)(int)) functable[4];
   free_fn = (void (*)(void*)) functable[5];

   return;
}
```

```
void  sleep (int count, int time)
{
   (*sleep_uninterrupted)(time);
}

void timer_handler ()
{
   fprintf(stderr,"Timer Handler called\n");
   /* Do something */
}

void  start_timer (int count, int time_to_int, int time_to_sleep)
{
   (*setup_timer)((int )start_timer, time_to_int, timer_handler, 0, 0);
   return;
}
void* xmalloc (int count)
{
return (*malloc_fn)(count);
}

void  xfree(void* ptr)
{
   (*free_fn)(ptr);
}
```

Example:gtm_malloc/gtm_free callbacks using gtm_pointertofunc_t

```
% echo $GTMXC
/usr/joe/callback.xc
% cat /usr/joe/callback.xc
/usr/lib/callback.sl
init: void init_callbacks(I:gtm_pointertofunc_t, I:gtm_pointertofunc_t)
% gtm
GTM> do &.init(4,5)
GTM>
% cat /usr/joe/callback.c
#include <stdio.h>
#include <stdlib.h>
#include "gtmxc_types.h"
void* (*malloc_fn)(int);
void (*free_fn)(void*);
void init_callbacks(int count, void* (*m)(int), void (*f)(void*))
{
    malloc_fn = m;
    free_fn = f;
}
```

Calls from External Routines:Call-Ins

Call-In is a framework supported by GT.M that allows a C/C++ program to invoke an M routine within the same process context. GT.M provides a well-defined Call-In interface packaged as a run-time shared library that can be linked into an external C/C++ program.

Relevant files for Call-Ins

To facilitate Call-Ins to M routines, the GT.M distribution directory ($gtm_dist) contains the following files:

1. libgtmshr.so - A shared library that implements the GT.M run-time system, including the Call-In API. If Call-Ins are used from a standalone C/C++ program, this library needs to be explicitly linked into the program. See "Building Standalone Programs" (page 539), which describes the necessary linker options on each supported platforms.

> **Note**
>
> .so is the recognized shared library file extension on most UNIX platforms.

2. mumps - The GT.M startup program that dynamically links with libgtmshr.so.

3. gtmxc_types.h - A C-header file containing the declarations of Call-In API.

The following sections describe the files relevant to using Call-Ins.

gtmxc_types.h

The header file provides signatures of all Call-In interface functions and definitions of those valid data types that can be passed from C to M. FIS strongly recommends that these types be used instead of native types (int, char, float, and so on), to avoid possible mismatch problems during parameter passing.

gtmxc_types.h defines the following types that can be used in Call-Ins.

Type	Usage
void	Used to express that there is no function return value
gtm_int_t	gtm_int_t has 32-bit length on all platforms.
gtm_uint_t	gtm_uint_t has 32-bit length on all platforms
gtm_long_t	gtm_long_t has 32-bit length on 32-bit platforms and 64-bit length on 64-bit platforms. It is much the same as the C language long type.
gtm_ulong_t	gtm_ulong_t is much the same as the C language unsigned long type.
gtm_float_t	floating point number
gtm_double_t	Same as above but double precision.
gtm_status_t	type int. If it returns zero then the call was successful. If it is non-zero, when control returns to GT.M, it issues a trappable error.
gtm_long_t*	Pointer to gtm_long_t. Good for returning integers.
gtm_ulong_t*	Pointer to gtm_ulong_t. Good for returning unsigned integers.

```
typedef struct {
    gtm_long_t length;
    gtm_char_t* address;
```

```
} gtm_string_t;
```

The pointer types defined above are 32-bit addresses on all 32-bit platforms. For 64-bit platforms, gtm_string_t* is a 64-bit address.

gtmxc_types.h also provides an input-only parameter type gtm_pointertofunc_t that can be used to obtain call-back function pointers via parameters in the external routine. If a parameter is specified as I:gtm_pointertofunc_t and if a numeric value (between 0-5) is passed for this parameter in M, GT.M interprets this value as the index into the callback table and passes the appropriate callback function pointer to the external routine.

> **Note**
>
> GT.M represents values that fit in 18 digits as numeric values, and values that require more than 18 digits as strings.

> **Note**
>
> GT.M MUMPS language runtime supports a maximum of 1MiB strings. Please take care to use 1MiB buffers for Output-only and Input-Output variables. Failure to do so could cause a segmentation violation if the called M routine writes more data into the supplied buffer than the caller allocated.

gtmxc_types.h also includes definitions for the following entry points exported from libgtmshr:

```
void gtm_hiber_start(gtm_uint_t mssleep);
void gtm_hiber_start_wait_any(gtm_uint_t mssleep)
void gtm_start_timer(gtm_tid_t tid, gtm_int_t time_to_expir, void (*handler)(), gtm_int_t hdata_len, void \
*hdata);
void gtm_cancel_timer(gtm_tid_t tid);
```

where:

- mssleep - milliseconds to sleep

- tid - unique timer id value

- time_to_expir - milliseconds until timer drives given handler

- handler - function pointer to handler to be driven

- hdata_len - 0 or length of data to pass to handler as a parameter

- hdata - NULL or address of data to pass to handler as a parameter

gtm_hiber_start() always sleeps until the time expires; gtm_hiber_start_wait_any() sleeps until the time expires or an interrupt by any signal (including another timer). gtm_start_timer() starts a timer but returns immediately (no sleeping) and drives the given handler when time expires unless the timer is canceled.

> **Important**
>
> GT.M continues to support xc_* equivalent types of gtm_* for upward compatibility. gtmxc_types.h explicitly marks the xc_* equivalent types as deprecated.

Call-In Table

The Call-In table file is a text file that contains the signatures of all M label references that get called from C. In order to pass the typed C arguments to the type-less M formallist, the enviroment variable GTMCI must be defined to point to the Call-In table file path. Each signature must be specified separately in a single line. GT.M reads this file and interprets each line according to the following convention (specifications withint box brackets "[]", are optional):

```
<c-call-name> : <ret-type> <label-ref> ([<direction>:<param-type>,...])
```

where,

<label-ref>: is the entry point (that is a valid label reference) at which GT.M starts executing the M routine being called-in

<c-call-name>: is a unique C identifier that is actually used within C to refer to <label-ref>

<direction>: is either I (input-only), O (output-only), or IO (input-output)

<ret-type>: is the return type of <label-ref>

> **Note**
>
> Since the return type is considered as an output-only (O) parameter, the only types allowed are pointer types and void. Void cannot be specified as parameter.

<param-type>: is a valid parameter type. Empty parentheses must be specified if no argument is passed to <label-ref>

The <direction> indicates the type of operation that GT.M performs on the parameter read-only (I), write-only (O), or read-write (IO). All O and IO parameters must be passed by reference, that is as pointers since GT.M writes to these locations. All pointers that are being passed to GT.M must be pre-allocated. The following table details valid type specifications for each direction.

Directions	Allowed Parameter types
I	gtm_long_t, gtm_ulong_t, gtm_float_t, gtm_double_t,_gtm_long_t*, gtm_ulong_t*, gtm_float_t*, gtm_double_t*,_gtm_char_t*, gtm_string_t*
O/IO	gtm_long_t*, gtm_ulong_t*, gtm_float_t*, gtm_double_t*,_gtm_char_t*, gtm_string_t*

Here is an example of Call-In table (calltab.ci) for piece.m (see "Example: Calling GT.M from a C Program" (page 538)):

```
print     :void            display^piece()
getpiece  :gtm_char_t*     get^piece(I:gtm_char_t*, I:gtm_char_t*, I:gtm_long_t)
setpiece  :void            set^piece(IO:gtm_char_t*, I:gtm_char_t*, I:gtm_long_t, I:gtm_char_t*)
pow       :gtm_double_t*   pow^piece(I:gtm_double_t, I:gtm_long_t)
powequal  :void            powequal^piece(IO:gtm_double_t*, I:gtm_long_t)
piece     :gtm_double_t*   pow^piece(I:gtm_double_t, I:gtm_long_t)
```

> **Note**
>
> The same entryref can be called by different C call names (for example, pow, and piece). However, if there are multiple lines with the same call name, only the first entry will be used by GT.M. GT.M ignores all subsequent entries using a call name. Also, note that the second and third entries, although shown here as wrapped across lines, must be specified as a single line in the file.

Call-InInterface

This section is further broken down into 6 subsections for an easy understanding of the Call-In interface. The section is concluded with an elaborate example.

Initialize GT.M

```
gtm_status_t gtm_init(void);
```

If the base program is not an M routine but a standalone C program, gtm_init() must be called (before calling any GT.M functions), to initialize the GT.M run-time system.

gtm_init() returns zero (0) on success. On failure, it returns the GT.M error status code whose message can be read into a buffer by immediately calling gtm_zstatus() (see "Print Error Messages" (page 539)). Duplicate invocations of gtm_init() are ignored by GT.M.

If Call-Ins are used from an external call function (that is, a C function that has itself been called from M code), gtm_init() is not needed, because GT.M is initialized before the External Call. All gtm_init() calls from External Calls functions are ignored by GT.M.

Call an M Routinefrom C

GT.M provides 2 interfaces for calling a M routine from C. These are:

- gtm_cip

- gtm_ci

gtm_cip offers better performance on calls after the first one.

gtm_cip

```
gtm_status_t gtm_cip(ci_name_descriptor *ci_info, ...);
```

The variable argument function gtm_cip() is the interface that invokes the specified M routine and returns the results via parameters.

ci_name_descriptor has the following structure:

```
typedef struct
{
  gtm_string_t rtn_name;
  void* handle;
} ci_name_descriptor;
```

rtn_name is a C character string indicating the corresponding <lab-ref> entry in the Call-In table.

The handle is GT.M private information initialized by GT.M on the first call-in and to be provided unmodified to GT.M on subsequent calls. If application code modifies it, it will corrupt the address space of the process, and potentially cause just about any bad behavior that it is possible for the process to cause, including but not limited to process death, database damage and security violations.

The gtm_cip() call must follow the following format:

```
status = gtm_cip(<ci_name_descriptor> [, ret_val] [, arg1] ...);
```

First argument: ci_name_descriptor, a null-terminated C character string indicating the alias name for the corresponding <lab-ref> entry in the Call-In table.

Optional second argument: ret_val, a pre-allocated pointer through which GT.M returns the value of QUIT argument from the (extrinsic) M routine. ret_val must be the same type as specified for <ret-type> in the Call-In table entry. The ret_val argument is needed if and only if <ret-type> is not void.

Optional list of arguments to be passed to the M routine's formallist: the number of arguments and the type of each argument must match the number of parameters, and parameter types specified in the corresponding Call-In table entry. All pointer arguments must be pre-allocated. GT.M assumes that any pointer, which is passed for O/IO-parameter points to valid write-able memory.

The status value returned by gtm_cip() indicates the GT.M status code; zero (0), if successful, or a non-zero; $ZSTATUS error code on failure. The $ZSTATUS message of the failure can be read into a buffer by immediately calling gtm_zstatus() (for details, see "Print Error Messages" (page 539)).

gtm_ci

```
gtm_status_t gtm_ci(const gtm_char_t* c_call_name, ...);
```

The variable argument function gtm_ci() is the interface that actually invokes the specified M routine and returns the results via parameters. The gtm_ci() call must be in the following format:

```
status = gtm_ci(<c_call_name> [, ret_val] [, arg1] ...);
```

First argument: c_call_name, a null-terminated C character string indicating the alias name for the corresponding <lab-ref> entry in the Call-In table.

Optional second argument: ret_val, a pre-allocated pointer through which GT.M returns the value of QUIT argument from the (extrinsic) M routine. ret_val must be the same type as specified for <ret-type> in the Call-In table entry. The ret_val argument is needed if and only if <ret-type> is not void.

Optional list of arguments to be passed to the M routine's formallist: the number of arguments and the type of each argument must match the number of parameters, and parameter types specified in the corresponding Call-In table entry. All pointer arguments must be pre-allocated. GT.M assumes that any pointer, which is passed for O/IO-parameter points to valid write-able memory.

The status value returned by gtm_ci() indicates the GT.M status code; zero (0), if successful, or a non-zero; $ZSTATUS error code on failure. The $ZSTATUS message of the failure can be read into a buffer by immediately calling gtm_zstatus(). For more details, see "Print Error Messages" (page 539).

Example: Calling GT.M from a C Program

Here are some working examples of C programs that use call-ins to invoke GT.M. Each example is packaged in a zip file which contains a C program, a call-in table, and a GT.M API. To run an example, click and follow the compiling and linking instructions in the comments of the C program.

Example	Download information
gtmaccess.c (gtm_ci interface)	Click to download or open directly from http://tinco.pair.com/bhaskar/gtm/doc/books/pg/UNIX_manual/gtmci_gtmaccess.zip

Example	Download information
gtmaccess.c (gtm_cip interface)	Click ▼ to download or open directly from http://tinco.pair.com/bhaskar/gtm/doc/books/pg/UNIX_manual/gtmcip_gtmaccess.zip
cpiece.c (gtm_ci interface)	Click ▼ to download or open directly from http://tinco.pair.com/bhaskar/gtm/doc/books/pg/UNIX_manual/gtmci_cpiece.zip

PrintError Messages

```
void gtm_zstatus (gtm_char_t* msg_buffer, gtm_long_t buf_len);
```

This function returns the null-terminated $ZSTATUS message of the last failure via the buffer pointed by msg_buffer of size buf_len. The message is truncated to size buf_len if it does not fit into the buffer. gtm_zstatus() is useful if the external application needs the text message corresponding to the last GT.M failure. A buffer of 2048 is sufficient to fit in any GT.M message.

Exit from GT.M

```
gtm_status_t  gtm_exit (void);
```

gtm_exit() can be used to shut down all databases and exit from the GT.M environment that was created by a previous gtm_init().

Note that gtm_init() creates various GT.M resources and keeps them open across multiple invocations of gtm_ci() until gtm_exit() is called to close all such resources. On successful exit, gtm_exit() returns zero (0), else it returns the $ZSTATUS error code.

gtm_exit() cannot be called from an external call function. GT.M reports the INVGTMEXIT error if an external call function invokes gtm_exit(). Since the GT.M run-time system must be operational even after the external call function returns, gtm_exit() is meant to be called only once during a process lifetime, and only from the base C/C++ program when GT.M functions are no longer required by the program.

BuildingStandalone Programs

All external C functions that use call-ins should include the header file gtmxc_types.h that defines various types and provides signatures of call-in functions. To avoid potential size mismatches with the parameter types, FIS strongly recommends that gtm *t types defined in gtmxc_types.h be used instead of the native types (int, float, char, etc).

To use call-ins from a standalone C program, it is necessary that the GT.M runtime library (libgtmshr.so) is explicitly linked into the program. If call-ins are used from an External Call function (which in turn was called from GT.M through the existing external call mechanism), the External Call library does not need to be linked explicitly with libgtmshr.so since GT.M would have already loaded it.

The following sections describe compiler and linker options that must be used on each platform for call-ins to work from a standalone C/C++ program.

IBM pSeries (RS/6000) AIX

• Compiler: -I$gtm_dist

- Linker: -L$gtm_dist -lgtmshr

X86 GNU/Linux

- Compiler: -I$gtm_dist

- Linker: -L$gtm_dist -lgtmshr -rpath $gtm_dist

- FIS advises that the C/C++ compiler front-end be used as the Linker to avoid specifying the system startup routines on the ld command. The compile can pass linker options to ld using -W option (for example, cc -W1, -R, $gtm_dist). For more details on these options, refer to the appropriate system's manual on the respective platforms.

Nested Call-Ins

Call-ins can be nested by making an external call function in-turn call back into GT.M. Each gtm_ci() called from an External Call library creates a call-in base frame at $ZLEVEL 1 and executes the M routine at $ZLEVEL 2. The nested call-in stack unwinds automatically when the External Call function returns to GT.M.

GT.M currently allows up to 10 levels of nesting, if TP is not used, and less than 10 if GT.M supports call-ins from a transaction (see "Rules to Follow in Call-Ins" (page 540)). GT.M reports the CIMAXLEVELS error when the nesting reaches its limit.

Following are the GT.M commands, Intrinsic Special Variables, and functions whose behavior changes in the context of every new nested call-in environment.

ZGOTO 0 (zero) returns to the processing of the invoking non-M routine as does ZGOTO 1 (one) with no entryref, while ZGOTO 1:entryref replaces the originally invoked M routine and continues M execution.

$ZTRAP/$ETRAP NEW'd at level 1 (in GTM$CI frame).

$ZLEVEL initializes to one (1) in GTM$CI frame, and increments for every new stack level.

$STACK initializes to zero (0) in GTM$CI frame, and increments for every new stack level.

$ESTACK NEW'd at level one (1) in GTM$CI frame.

$ECODE/$STACK() initialized to null at level one (1) in GTM$CI frame.

> **Note**
>
> After a nested call-in environment exits and the external call C function returns to M, the above ISVs and Functions restore their old values.
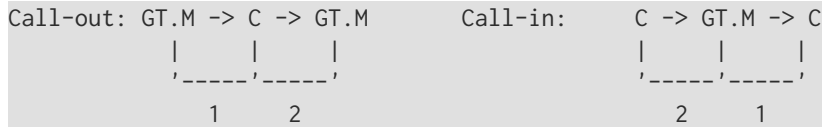
Rules to Follow in Call-Ins

1. External calls must not be fenced with TSTART/TCOMMIT if the external routine calls back into mumps using call-in mechanism. GT.M reports the CITPNESTED error if nested call-ins are invoked within a TP fence since GT.M currently does not handle TP support across multiple call-in invocations.

2. The external application should never call exit() unless it has called gtm_exit() previously. GT.M internally installs an exit handler that should never be bypassed.

3. The external application should never use any signals when GT.M is active since GT.M reserves them for its internal use. GT.M provides the ability to handle SIGUSR1 within M (see "$ZINTerrupt" (page 309) for more information). An interface is provided by GT.M for timers.

4. FIS recommends the use of gtm_malloc() and gtm_free() for memory management by C code that executes in a GT.M process space for enhanced performance and improved debugging. Always use gtm_malloc to allocate returns for pointer types to prevent memory leaks.

5. GT.M performs device input using the read() system service. UNIX documentation recommends against mixing this type of input with buffered input services in the fgets() family and ignoring this recommendation is likely to cause loss of input that is difficult to diagnose and understand.

Type Limits for Call-ins and Call-outs

Depending on the direction (I, O, or IO) of a particular type, both call-ins and call-outs may transfer a value in two directions as follows:

```
Call-out: GT.M -> C -> GT.M        Call-in:     C -> GT.M -> C
            |     |     |                        |     |     |
            '-----'-----'                        '-----'-----'
               1     2                              2     1
```

In the following table, the GT.M->C limit applies to 1 and the C->GT.M limit applies to 2. In other words, GT.M->C applies to I direction for call-outs and O direction for call-ins and C->GT.M applies to I direction for call-ins and O direction for call-outs.

Type	GTM->C		C->GT.M	
	Precision	**Range**	**Precision**	**Range**
gtm_int_t, gtm_int_t *	Full	[-2^31+1, 2^31-1]	Full	[-2^31, 2^31-1]
gtm_uint_t, gtm_uint_t *	Full	[0, 2^32-1]	Full	[0, 2^32-1]
gtm_long_t, gtm_long_t * (64-bit)	18 digits	[-2^63+1, 2^63-1]	18 digits	[-2^63, 2^63-1]
gtm_long_t, gtm_long_t * (32-bit)	Full	[-2^31+1, 2^31-1]	Full	[-2^31, 2^31-1]
gtm_ulong_t, gtm_ulong_t * (64-bit)	18 digits	[0, 2^64-1]	18 digits	[0, 2^64-1]
gtm_ulong_t, gtm_ulong_t * (32-bit)	Full	[0, 2^32-1]	Full	[0, 2^32-1]
gtm_float_t, gtm_float_t *	6-9 digits	[1E-43, 3.4028235E38]	6 digits	[1E-43, 3.4028235E38]
gtm_double_t, gtm_double_t *	15-17 digits	[1E-43, 1E47]	15 digits	[1E-43, 1E47]
gtm_char_t *	N/A	["", 1MiB]	N/A	["", 1MiB]
gtm_char_t **	N/A	["", 1MiB]	N/A	["", 1MiB]
gtm_string_t *	N/A	["", 1MiB]	N/A	["", 1MiB]

Notes

- gtm_char_t ** is not supported for call-ins but they are included for IO and O direction usage with call-outs.

- For call-out use of gtm_char_t * and gtm_string_t *, the specification in the interface definition for preallocation sets the range for IO and O, with a maximum of 1MiB.

Chapter 12. Internationalization

Revision History		
Revision V6.3-011	20 December 2019	• In "Establishing A Local Collation Sequence" (page 545), Move some material to the Utilities chapter • In "Examining Global Collation Characteristics" (page 553), Move some material to Utilities chapter • In "Using the %GBLDEF Utility" (page 552), Move some material to the Utilities chapter
Revision V6.3-010	31 October 2019	• In "Transform Utility Routine (gtm_ac_xutil)" (page 550), add new section • In "Transformation Routine (gtm_ac_xform_1 or gtm_ac_xform)" (page 547), correct section titles for input and output arguments; fix typos
Revision V6.3-006	26 October 2018	• In "Implementing an Alternative Collation Sequence for Unicode® characters" (page 558), minor corrections. • In "Pattern Code Definition" (page 559), UTF-8 tweaks
Revision V6.3-001	20 March 2017	• In "Pattern Code Selection" (page 560), added a compiler warning.
Revision V6.1-000	28 August 2014	In "Using the %GBLDEF Utility" (page 552), added changes for global spaning regions.

This chapter describes GT.M facilities for applications using characters encoded in other than eight-bit bytes (octets). Before continuing with use of UTF-8 features, you will need to ensure that your system has installed and configured the needed infrastructure for languages you wish to support, including International Components for Unicode (ICU / libicu), UTF-8 locale(s), and terminal emulators with appropriate fonts. This chapter addresses the specific issues of defining alternative collation sequences, and defining unique patterns for use with the pattern match operator.

Alternative collation sequences (or an alternative ordering of strings) can be defined for global and local variable subscripts. They can be established for specified globals or for an entire database. The alternative sequences are defined by a series of routines in an executable file pointed to by an environment variable. As the collation sequence is implemented by a user-supplied program, virtually any collation policy may be implemented. Detailed information on establishing alternative collation sequences and defining the environment variable is provided in "Collation Sequence Definitions" (page 544).

M has defined pattern classes that serve as arguments to the pattern match operator. GT.M supports user definition of additional pattern classes as well as redefinition of the standard pattern classes. Specific patterns are defined in a text file that is pointed to by an environment variable. Pattern classes may be re-defined dynamically. The details of defining these pattern classes and the environment variable are described in the section called "Matching Alternative Patterns" (page 558).

For some languages (such as Chinese), the ordering of strings according to Unicode® code-points (character values) may or may not be the linguistically or culturally correct ordering. Supporting applications in such languages requires development of collation modules - GT.M natively supports M collation, but does not include pre-built collation modules for any specific natural language. Therefore, applications that use characters in Unicode may need to implement their own collation functions. For more information on developing a collation module for Unicode, refer to "Implementing an Alternative Collation Sequence for Unicode® characters" (page 558).

Collation Sequence Definitions

Normally, GT.M orders data with numeric values first, followed by strings sequenced by ASCII values. To use an alternative collating sequence the following items must be provided at GT.M process intialization.

- A shared library containing the routines for each alternative collation sequence

- An environment variable of the form gtm_collate_n, specifying the shared library containing the routines for alternative collation sequence n.

Creating the Shared Library holding the alternative sequencing routines

A shared library for an alternative collation sequence must contain the following four routines:

1. gtm_ac_xform_1: Transforms subscripts up to the maximum supported string length to the alternative collation sequence, or

   gtm_ac_xform: Transforms subscripts up to 32,767 bytes to the alternative collation sequence.

2. gtm_ac_xback_1: Use with gtm_ac_xform_1 to transform the alternative collation keys back to the original subscript representation, or

   gtm_ac_xback: Use with gtm_ac_xform to transforms the alternative collation keys back to the original subscript representation.

3. gtm_ac_version: Returns a numeric version identifier for the "currently active" set of collation routines.

4. gtm_ac_verify: Returns the success (odd) or failure (even) in matching a collation sequence with a given version number.

GT.M searches the shared library for the gtm_ac_xform_1 and gtm_ac_xback_1 before searching for the gtm_ac_xform and gtm_ac_xback routines. If the shared library contains gtm_ac_xform_1, GT.M ignores gtm_ac_xform even if it is present. If GT.M finds gtm_ac_xform_1 but does not find gtm_ac_xback_1, it reports a COLLATIONUNDEF error with an additional mismatch COLLFNMISSING warning.

If the application does not use strings longer than 32,767 bytes, the alternative collation library need not contain the gtm_ac_xform_1 and gtm_ac_xback_1 routines. On the other hand, if the application passes strings greater than 32,767 bytes (but less than the maximum support string length) and does not provide gtm_xc_xform_1 and gtm_xc_xback_1, GT.M issues the COLLARGLONG run-time error.

Note that database key sizes are much more restricted by GT.M than local key sizes, and may be restricted further by user configuration.

Defining theEnvironment Variable

GT.M locates the alternative collation sequences through the environment variable gtm_collate_n where n is an integer from 1 to 255 that identifies the collation sequence, and pathname identifies the shared library containing the routines for that collation sequence, for example:

```
$ gtm_collate_1=/opt/fis-gtm/collation
$ export gtm_collate_1
```

Multiple alternative collation sequence definitions can co-exist.

Considerations in Establishing Alternative Collations

Alternative collation sequences for a global must be set when the global contains no data. When the global is defined the collation sequence is stored in the global. This ensures the future integrity of the global's collation. If it becomes necessary to change the collation sequence of a global containing data, you must copy the data to a temporary repository, delete the global, modify the variable's collation sequence by reinitializing the global either in a region that has the desired collation or with %GBLDEF, and restore the data from the temporary repository.

Be careful when creating the transformation and inverse transformation routines. The transformation routine must unambiguously and reliably encode every possible input value. The inverse routine must faithfully return the original value in every case. Errors in these routines can produce delayed symptoms that could be hard to debug. These routines may not be written in M.

Defining a Default DatabaseCollation Method

GT.M lets you define an alternative collation sequence as the default when creating a new database. Subsequently, this default is applied when each new global is created.

This default collation sequence is set as a GDE qualifier for the ADD, CHANGE, and TEMPLATE commands using the following example with CHANGE:

```
GDE>CHANGE -REGION DEFAULT -COLLATION_DEFAULT=<0-255>
```

This qualifier always applies to regions, and takes effect when a database is created with MUPIP CREATE. The output of GDE SHOW displays this value, and DSE DUMP -FILEHEADER also includes this information. In the absence of an alternative default collations sequence, the default used is 0, or ASCII.

The value cannot be changed once a database file is created, and will be in effect for the life of the database file. The same restriction applies to the version of the collation sequence. The version of a collation sequence implementation is also stored in the database fileheader and cannot be modified except by recreating the file.

If the code of the collation sequence changes, making it incompatible with the collation sequence in use when the database was created, use the following procedure to ensure the continued validity of the database. MUPIP EXTRACT the database using the older compatible collation routines, then recreate and MUPIP LOAD using the newer collation routines.

Establishing A Local Collation Sequence

All subscripted local variables for a process must use the same collation sequence. The collation sequence used by local variables can be established as a default or in the current process. The local collation sequence can only be changed when a process has no subscripted local variables defined.

To establish a default local collation sequence provide a numeric value to the environment variable gtm_local_collate to select one of the collation tables, for example:

```
$ gtm_local_collate=n
$ export gtm_local_collate
```

where n is the number of a collation sequence that matches a valid collation number defined by an environment variable in the form gtm_collate_n.

An active process can use the %LCLCOL utility to define the collation sequence for subscripts of local variables.

For more information, refer to "%LCLCOL " (page 507)in the Utilities Chapter of this manual.

set^%LCLCOL(n)changes the local collation to the type specified by n.

Example:

```
IF '$$set^%LCLCOL(3) D
. Write "local collation sequence not changed",! Break
```

This piece of code illustrates $$set^LCLCOL used as an extrinsic. It would write an error message and BREAK if the local collation sequence was not set to 3.

set^%LCLCOL(n,ncol) determines the null collation type to be used with the collation type n.

- If the truth value of ncol is FALSE(0), local variables use the GT.M standard null collation.

- If the truth value of ncol is TRUE(1), local variables use the M standard null collation.

With set^%LCLCOL(,ncol), the null collation order can be changed while keeping the alternate collation order unchanged. If subscripted local variables exist, null collation order cannot be changed. In this case, GT.M issues the COLLDATAEXISTS error.

get^%LCLCOL returns the current local type.

Example:

```
GTM>Write $$get^%LCLCOL
0
```

This example uses $$get^%LCLCOL as an extrinsic that returns 0, indicating that the effective local collation sequence is the standard M collation sequence.

If set^%LCLCOL is not specified and gtm_local_collate is not defined, or is invalid, the process uses M standard collation. The following would be considered invalid values:

- A value less than 0

- A value greater than 255

- A legal collation sequence that is inaccessible to the process

Inaccessibility could be caused by a missing environment variable, a missing image, or security denial of access.

Creating the Alternate Collation Routines

Each alternative collation sequence requires a set of four user-created routines--gtm_ac_xform_1 (or gtm_ac_xform), gtm_ac_xback_1 (or gtm_ac_xback), gtm_ac_version, and gtm_ac_verify. The original and transformed strings are passed

---

between GT.M and the user-created routines using parameters of type gtm_descriptor or gtm32_descriptor. An "include file" gtm_descript.h, located in the GT.M distribution directory, defines gtm_descriptor (used with gtm_ac_xform and gtm_ac_xback) as:

```
typedef struct
{
    short len;
    short type;
    void *val;
} gtm_descriptor;
```

> **Note**
>
> On 64-bit UNIX platforms, gtm_descriptor may grow by up to eight (8) additional bytes as a result of compiler padding to meet platform alignment requirements.

gtm_descript.h defines gtm32_descriptor (used with gtm_xc_xform_1 and gtm_xc_xback_2) as:

```
typedef struct
{
    unsigned int len;
    unsigned int type;
    void *val;
} gtm32_descriptor;
```

where len is the length of the data, type is set to DSC_K_DTYPE_T (indicating that this is an M string), and val points to the text of the string.

The interface to each routine is described below.

Transformation Routine (gtm_ac_xform_1 or gtm_ac_xform)

gtm_ac_xform_1 or gtm_ac_xform routines transforms subscripts to the alternative collation sequence.

This routine returns altered keys to the original subscripts. The syntax of this routine is:

```
#include "gtm_descript.h"
long gtm_ac_xback(gtm_descriptor *in, int level, gtm_descriptor *out, int *outlen)
```

If the application uses subscripted lvns longer than 32,767 bytes (but less than 1,048,576 bytes), the alternative collation library must contain the gtm_ac_xform_1 and gtm_ac_xback_1 routines. Otherwise, the alternative collation library can contain gtm_ac_xform and gtm_ac_xback.

The syntax of this routine is:

```
#include "gtm_descript.h"
int gtm_ac_xform_1(gtm32_descriptor* in, int level, gtm32_descriptor* out, int* outlen);
```

Input Arguments for gtm_ac_xform1

The input arguments for gtm_ac_xform1 are:

in: a gtm32_descriptor containing the string to be transformed.

level: an integer; this is not used currently, but is reserved for future facilities.

out: a gtm32_descriptor to be filled with the transformed key.

Output Arguments for gtm_ac_xform1

The output arguments for gtm_ac_xform1 are:

return value: A long word status code.

out: A transformed subscript in the string buffer, passed by gtm32_descriptor.

outlen: A 32-bit signed integer, passed by reference, returning the actual length of the transformed key.

The syntax of gtm_ac_xform routine is:

```
#include "gtm_descript.h"
long gtm_ac_xform(gtm_descriptor *in, int level, gtm_descriptor *out, int *outlen)
```

Input Arguments for gtm_ac_xform

The input arguments for gtm_ac_xform are:

in: a gtm_descriptor containing the string to be transformed.

level: an integer; this is not used currently, but is reserved for future facilities.

out: a gtm_descriptor to be filled with the transformed key.

Output Arguments for gtm_ac_xform

The output arguments for gtm_ac_xform are:

return value: a long result providing a status code; it indicates the success (zero) or failure (non-zero) of the transformation.

out: a gtm_descriptor containing the transformed key.

outlen: an unsigned long, passed by reference, giving the actual length of the output key.

Example:

```
#include "gtm_descript.h"
#define MYAPP_SUBSC2LONG 12345678
static unsigned char xform_table[256] =
{
  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63,
 64, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93,
 95, 97, 99,101,103,105,107,109,111,113,115,117,118,119,120,121,
122, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94,
 96, 98,100,102,104,106,108,110,112,114,116,123,124,125,126,127,
128,129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,
144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,
```

```
160,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,
176,177,178,179,180,181,182,183,184,185,186,187,188,189,190,191,
192,193,194,195,196,197,198,199,200,201,202,203,204,205,206,207,
208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223,
224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239,
240,241,242,243,244,245,246,247,248,249,250,251,252,253,254,255
};
long
gtm_ac_xform (in, level, out, outlen)
     gtm_descriptor *in;    /* the input string */
     int level;              /* the subscript level */
     gtm_descriptor *out;    /* the output buffer */
     int *outlen;          /* the length of the output string */
{
  int n;
  unsigned char *cp, *cout;
/* Ensure space in the output buffer for the string. */
  n = in->len;
  if (n > out->len)
    return MYAPP_SUBSC2LONG;
/* There is space, copy the string, transforming, if necessary */
  cp = in->val;               /* Address of first byte of input string */
  cout = out->val;         /* Address of first byte of output buffer */
  while (n-- > 0)
    *cout++ = xform_table[*cp++];
  *outlen = in->len;
  return 0;
}
```

Transformation Routine Characteristics

The input and output values may contain <NUL> (hex code 00) characters.

The collation transformation routine may concatenate a sentinel, such as <NUL>, followed by the original subscript on the end of the transformed key. If key length is not an issue, this permits the inverse transformation routine to simply retrieve the original subscript rather than calculating its value based on the transformed key.

If there are reasons not to append the entire original subscript, GT.M allows you to concatenate a sentinel plus a predefined code so the original subscript can be easily retrieved by the inverse transformation routine, but still assures a reformatted key that is unique.

Inverse Transformation Routine (gtm_ac_xback or gtm_ac_xback_1)

This routine returns altered keys to the original subscripts. The syntax of this routine is:

```
#include "gtm_descript.h"
long gtm_ac_xback(gtm_descriptor *in, int level, gtm_descriptor *out, int *outlen)
```

The arguments of gtm_ac_xback are identical to those of gtm_ac_xform.

The syntax of gtm_ac_xback_1 is:

```
#include "gtm_descript.h"
long gtm_ac_xback_1(gtm32_descriptor *src, int level, gtm32_descriptor *dst, int *dstlen)
```

The arguments of gtm_ac_xback_1 are identical to those of gtm_ac_xform_1.

Example:

```
#include "gtm_descript.h"
#define MYAPP_SUBSC2LONG 12345678
static unsigned char inverse_table[256] =
{
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63,
64, 65, 97, 66, 98, 67, 99, 68,100, 69,101, 70,102, 71,103, 72,
104, 73,105, 74,106, 75,107, 76,108, 77,109, 78,110, 79,111, 80,
112, 81,113, 82,114, 83,115, 84,116, 85,117, 86,118, 87,119, 88,
120, 89,121, 90,122, 91, 92, 93, 94, 95, 96,123,124,125,126,127,
128,129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,
144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,
160,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,
176,177,178,179,180,181,182,183,184,185,186,187,188,189,190,191,
192,193,194,195,196,197,198,199,200,201,202,203,204,205,206,207,
208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223,
224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239,
240,241,242,243,244,245,246,247,248,249,250,251,252,253,254,255
};
long gtm_ac_xback (in, level, out, outlen)
    gtm_descriptor *in;      /* the input string */
    int level;               /* the subscript level */
    gtm_descriptor *out;     /* output buffer */
    int *outlen;             /* the length of the output string */
{
  int n;
  unsigned char *cp, *cout;
/* Ensure space in the output buffer for the string. */
  n = in->len;
  if (n > out->len)
    return MYAPP_SUBSC2LONG;
/* There is enough space, copy the string, transforming, if necessary */
  cp = in->val;               /* Address of first byte of input string */
  cout = out->val;            /* Address of first byte of output buffer */
  while (n-- > 0)
    *cout++ = inverse_table[*cp++];
  *outlen = in->len;
  return 0;
}
```

Transform Utility Routine (gtm_ac_xutil)

This routine returns a next or previous character in the collation sequence. The syntax of this routine is:

```
#include "gtm_descript.h"
long gtm_ac_xutil (gtm32_descriptor *in, int level, gtm32_descriptor *out, int *outlen, int op, int
▶ honor_numeric)
```

Input Arguments

The input arguments of gtm_ac_xutil are:

- in: Specifies the input string; gtm_ac_xutil considers the first character of the input string.

- level: Currently unused and should not be examined or changed.

- honor_numeric: Boolean variable to specify whether to use standard GT.M collation for digits.

  - TRUE: use standard GT.M collation for digits before any other character

  - FALSE: treat digits the same as all other characters

Output Arguments

The output arguments of gtm_ac_xutil are:

- out: Supplies the one (1) character result string produced by applying the collation operation if a result was possible.

- outlen: Supplies to the caller the length of the returned string- 0 or 1.

- op: Supplies the collation operation as follows:

  - 0: collation value of the given character

  - 1: character collating before the given character if it exists

  - 2: character collating after the given character if it exists

The gtm_ac_xutil function returns 0 on success and -1 on failure.

Version Control Routines (gtm_ac_version and gtm_ac_verify)

Two user-defined version control routines provide a safety mechanism to guard against a collation routine being used on the wrong global, or an attempt being made to modify a collation routine for an existing global. Either of these situations could cause incorrect collation or damage to subscripts.

When a global is assigned an alternative collation sequence, GT.M invokes a user-supplied routine that returns a numeric version identifier for the set of collation routines, which was stored with the global. The first time a process accesses the global, GT.M determines the assigned collation sequence, then invokes another user-supplied routine. The second routine matches the collation sequence and version identifier assigned to the global with those of the current set of collation routines.

When you write the code that matches the type and version, you can decide whether to modify the version identifier and whether to allow support of globals created using a previous version of the routine.

Version Identifier Routine (gtm_ac_version)

This routine returns an integer identifier between 0 and 255. This integer provides a mechanism to enforce compatibility as a collation sequence potentially evolves. When GT.M first uses an alternate collation sequence for a database or global, it captures the version and if it finds the version has changed it at some later startup, it generates an error. The syntax is:

```
int gtm_ac_version()
```

Example:

```
int gtm_ac_version()
{
    return 1;
}
```

Verification Routine (gtm_ac_verify)

This routine verifies that the type and version associated with a global are compatible with the active set of routines. Both the type and version are unsigned characters passed by value. The syntax is:

```
#include "gtm_descript.h"
int gtm_ac_verify(unsigned char type, unsigned char ver)
```

Example:

```
Example:
#include "gtm_descript.h"
#define MYAPP_WRONGVERSION 20406080    /* User condition */
gtm_ac_verify (type, ver)
     unsigned char type, ver;
{
  if (type == 3)
    {
      if (ver > 2)         /* version checking may be more complex */
      {
        return 0;
      }
}
  return MYAPP_WRONGVERSION;
}
```

Using the %GBLDEF Utility

Use the %GBLDEF utility to get, set, or kill the collation sequence of a global variable mapped by the current global directory. %GBLDEF cannot modify the collation sequence for either a global containing data or a global whose subscripts span multiple regions. To change the collation sequence for a global variable that contains data, extract the data, KILL the variable, change the collation sequence, and reload the data. Use GDE to modify the collation sequence of a global variable that spans regions.

For more information, refer to "%GBLDEF " (page 505)in the Utilities Chapter of this manual.

Assigning the Collation Sequence

To assign a collation sequence to an individual global use the extrinsic entry point:

```
set^%GBLDEF(gname,nct,act)
```

Example:

```
GTM>kill ^G
GTM>write $select($$set^%GBLDEF("^G",0,3):"ok",1:"failed")
ok
```

```
GTM>
```

This deletes the global variable ^G, then uses the $$set%GBLDEF as an extrinsic to set ^G to the collation sequence number 3 with numeric subscripts collating before strings. Using $$set%GBLDEF as an argument to $SELECT provides a return value as to whether or not the set was successful. $SELECT will return a "FAILED" message if the collation sequence requested is undefined.

Examining Global Collation Characteristics

To examine the collation characteristics currently assigned to a global use the extrinsic entry point:

```
get^%GBLDEF(gname[,reg])
```

> **Note**
>
> get^%GBLDEF(gname) returns global specific characteristics, which can differ from collation characteristics defined for the database file at MUPIP CREATE time from settings in the global directory.
>
> DSE DUMP -FILEHEADER command displays region collation whenever the collation is other than M standard collation.

Example:

```
GTM>Write $$get^%GBLDEF("^G")
1,3,1
```

This example returns the collation sequence information currently assigned to the global ^G.

Deleting Global Collation Characteristics

To delete the collation characteristics currently assigned to a global, use the extrinsic entry point:

```
kill^%GBLDEF(gname)
```

Example of Upper and Lower Case Alphabetic Collation Sequence

This example is create an alternate collation sequence that collates upper and lower case alphabetic characters in such a way that the set of keys "du Pont," "Friendly," "le Blanc," and "Madrid" collates as:

- du Pont

- Friendly

- le Blanc

- Madrid

This is in contrast to the standard M collation that orders them as:

- Friendly

- Madrid

- du Pont

- le Blanc

> ⚠️ **Important**
>
> No claim of copyright is made with respect to the code used in this example. Please do not use the code as-is in a production environment.

Please ensure that you have a correctly configured GT.M installation, correctly configured environment variables, with appropriate directories and files.

Seasoned GT.M users may want download polish.c used in this example and proceed directly to Step 5 for compiling and linking instructions. First time users may want to start from Step 1.

1. Create a new file called polish.c and put the following code:

```
#include <stdio.h>
#include "gtm_descript.h"
#define COLLATION_TABLE_SIZE     256
#define MYAPPS_SUBSC2LONG        12345678
#define SUCCESS     0
#define FAILURE     1
#define VERSION     0
static unsigned char xform_table[COLLATION_TABLE_SIZE] =
        {
        0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
        16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
        32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
        48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63,
        64, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83, 85, 87, 89, 91, 93,
        95, 97, 99,101,103,105,107,109,111,113,115,117,118,119,120,121,
        122, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94,
        96, 98,100,102,104,106,108,110,112,114,116,123,124,125,126,127,
        128,129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,
        144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,
        160,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,
        176,177,178,179,180,181,182,183,184,185,186,187,188,189,190,191,
        192,193,194,195,196,197,198,199,200,201,202,203,204,205,206,207,
        208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223,
        224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239,
        240,241,242,243,244,245,246,247,248,249,250,251,252,253,254,255
        };
static unsigned char inverse_table[COLLATION_TABLE_SIZE] =
        {
        0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
        16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
        32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47,
        48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63,
        64, 65, 97, 66, 98, 67, 99, 68,100, 69,101, 70,102, 71,103, 72,
        104, 73,105, 74,106, 75,107, 76,108, 77,109, 78,110, 79,111, 80,
        112, 81,113, 82,114, 83,115, 84,116, 85,117, 86,118, 87,119, 88,
        120, 89,121, 90,122, 91, 92, 93, 94, 95, 96,123,124,125,126,127,
        128,129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,
```

```
        144,145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,
        160,161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,
        176,177,178,179,180,181,182,183,184,185,186,187,188,189,190,191,
        192,193,194,195,196,197,198,199,200,201,202,203,204,205,206,207,
        208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223,
        224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239,
        240,241,242,243,244,245,246,247,248,249,250,251,252,253,254,255
        };
```

Elements in xform_table represent input order for transform. Elements in inverse_table represent reverse transform for x_form_table.

2. Add the following code for the gtm_ac_xform transformation routine:

```
long gtm_ac_xform ( gtm_descriptor *src, int level, gtm_descriptor *dst, int *dstlen)
    {
        int n;
        unsigned char  *cp, *cpout;
#ifdef DEBUG
        char input[COLLATION_TABLE_SIZE], output[COLLATION_TABLE_SIZE];
#endif
        n = src->len;
        if ( n > dst->len)
            return MYAPPS_SUBSC2LONG;
        cp  = (unsigned char *)src->val;
#ifdef DEBUG
        memcpy(input, cp, src->len);
        input[src->len] = '\0';
#endif
        cpout = (unsigned char *)dst->val;
        while ( n-- > 0 )
            *cpout++ = xform_table[*cp++];
        *cpout = '\0';
        *dstlen = src->len;
#ifdef DEBUG
        memcpy(output, dst->val, dst->len);
        output[dst->len] = '\0';
        fprintf(stderr, "\nInput = \n");
        for (n = 0; n < *dstlen; n++ ) fprintf(stderr," %d ",(int )input[n]);
        fprintf(stderr, "\nOutput = \n");
        for (n = 0; n < *dstlen; n++ ) fprintf(stderr," %d ",(int )output[n]);
#endif
        return SUCCESS;
    }
  3. Add the following code for the gtm_ac_xback reverse transformation routine:
    long gtm_ac_xback ( gtm_descriptor *src, int level, gtm_descriptor *dst, int *dstlen)
    {
        int n;
        unsigned char  *cp, *cpout;
#ifdef DEBUG
        char input[256], output[256];
#endif
        n = src->len;
        if ( n > dst->len)
        return MYAPPS_SUBSC2LONG;
```

```
        cp   = (unsigned char *)src->val;
        cpout = (unsigned char *)dst->val;
        while ( n-- > 0 )
            *cpout++ = inverse_table[*cp++];
        *cpout = '\0';
        *dstlen = src->len;
#ifdef DEBUG
        memcpy(input, src->val, src->len);
        input[src->len] = '\';
        memcpy(output, dst->val, dst->len);
        output[dst->len] = '\0';
        fprintf(stderr, "Input = %s, Output = %s\n",input, output);
#endif
        return SUCCESS;
    }
```

3. Add code for the version identifier routine (gtm_ac_version) or the verification routine (gtm_ac_verify):

```
int gtm_ac_version ()
    {
        return VERSION;
    }
    int gtm_ac_verify (unsigned char type, unsigned char ver)
    {
            return !(ver == VERSION);
    }
```

4. Save and compile polish.c. On x86 GNU/Linux (64-bit Ubuntu 10.10), execute a command like the following:

```
gcc -c polish.c -I$gtm_dist
```

> **Note**
>
> The -I$gtm_dist option includes gtmxc_types.h.

5. Create a new shared library or add the above routines to an existing one. The following command adds these alternative sequence routines to a shared library called altcoll.so on x86 GNU/Linux (64-bit Ubuntu 10.10).

```
gcc -o altcoll.so -shared polish.o
```

6. Set $gtm_collate_1 to point to the location of altcoll.so.

7. At the GTM> prompt execute the following command:

```
GTM>Write $SELECT($$set^%GBLDEF("^G",0,1):"OK",1:"FAILED")
    OK
```

This deletes the global variable ^G, then sets ^G to the collation sequence number 1 with numeric subscripts collating before strings.

8. Assign the following value to ^G.

```
GTM>Set ^G("du Pont")=1
GTM>Set ^G("Friendly")=1
GTM>Set ^G("le Blanc")=1
GTM>Set ^G("Madrid")=1
```

9. See how the subscript of ^G order according to the alternative collation sequence:

```
GTM>ZWRite ^G
^G("du Pont")=1
^G("Friendly")=1
^G("le Blanc")=1
^G("Madrid")=1
```

Example of Collating Alphabets in Reverse Order using gtm_ac_xform_1 and gtm_ac_xback_1

This example creates an alternate collation sequence that collates alphabets in reverse order. This is in contrast to the standard M collation that collates alphabets in ascending order.

> ### ⚠ Important
>
> No claim of copyright is made with respect to the code used in this example. Please do not use the code as-is in a production environment.

Please ensure that you have a correctly configured GT.M installation, correctly configured environment variables, with appropriate directories and files.

1. Download col_reverse_32.c from http://tinco.pair.com/bhaskar/gtm/doc/books/pg/UNIX_manual/col_reverse_32.c. It contain code for transformation routine (gtm_ac_xform_1), reverse transformation routine (gtm_ac_xback_1) and version control routines (gtm_ac_version and gtm_ac_verify).

2. Save and compile col_reverse_32.c. On x86 GNU/Linux (64-bit Ubuntu 10.10), execute a command like the following:

```
gcc -c col_reverse_32.c -I$gtm_dist
```

> ### 📝 Note
>
> The -I$gtm_dist option includes gtmxc_types.h.

3. Create a new shared library or add the routines to an existing one. The following command adds these alternative sequence routines to a shared library called altcoll.so on x86 GNU/Linux (64-bit Ubuntu 10.10).

```
gcc -o revcol.so -shared col_reverse_32.o
```

4. Set the environment variable gtm_collate_2 to point to the location of revcol.so. To set the local variable collation to this alternative collation sequence, set the environment variable gtm_local_collate to 2.

5. At the GTM prompt, execute the following command:

```
GTM>Write $SELECT($$set^%GBLDEF("^E",0,2):"OK",1:"FAILED")
OK
```

6. Assign the following value to ^E.

```
GTM>Set ^E("du Pont")=1
GTM>Set ^E("Friendly")=1
GTM>Set ^E("le Blanc")=1
```

```
GTM>Set ^E("Madrid")=1
```

7. Notice how the subscript of ^E sort in reverse order:

```
GTM>zwrite ^E
^G("le Blanc")=1
^G("du Pont")=1
^G("Madrid")=1
^G("Friendly")=1
```

Implementing an Alternative Collation Sequence for Unicode® characters

By default, GT.M sorts string subscripts in the default order of the Unicode® UTF-8 numeric code-point values. Since this implied ordering may or may not be linguistically or culturally correct for a specific application, an implementation of an algorithm such as the Unicode® Collation Algorithm (UCA) may be required. Note that implementation of collation in GT.M requires the implementation of two functions, f(x) and g(y). f(x) transforms each input sequence of bytes into an alternative sequence of bytes for storage. Within the GT.M database engine, M nodes are retrieved according to the byte order in which they are stored. For each y that can be generated by f(x), g(y) is an inverse function that provides the original sequence of bytes; in other words, g(f(x)) must be equal to x for all x that the application processes. For example, for the People's Republic of China, it may be appropriate to convert from UTF-8 to Guojia Biaozhun (国家标准), the GB18030 standard, for example, using the libiconv library. The following requirements are important:

- *Unambiguous transformation routines*: The transform and its inverse must convert each input string to a unique sequence of bytes for storage, and convert each sequence of bytes stored back to the original string.

- *Collation sequence for all expected character sequences in subscripts*: GT.M does not validate the subscript strings passed to/from the collation routines. If the application design allows illegal UTF-8 character sequences to be stored in the database, the collation functions must appropriately transform, and inverse transform, these as well.

- *Handle different string lengths for before and after transformation*: If the lengths of the input string and transformed string differ, and, for local variables, if the output buffer passed by GT.M is not sufficient, follow the procedure described below:

  - *Global Collation Routines*: The transformed key must not exceed the lesser of the maximum key size configuration or 1019 bytes, the maximum GDS key size. GT.M allocates a temporary buffer of size 1019 bytes in the output string descriptor (of type DSC_K_DTYPE_T) and passes it to the collation routine to return the transformed key.

  - *Local Collation Routines*: GT.M allocates a temporary buffer in the output string descriptor based on the size of the input string. Both transformation and inverse transformation must check the buffer size, and if it is not sufficient, the transformation must allocate sufficient memory, set the output descriptor value (val field of the descriptor) to point to the new memory , and return the transformed key successfully. Since GT.M copies the key from the output descriptor into its internal structures, it is important that the memory allocated remain available even after the collation routines return. Collation routines are typically called throughout the process lifetime, therefore, GT.M expects the collation libraries to define a large static buffer sufficient to hold all key sizes in the application. Alternatively, the collation transform can use a large heap buffer (allocated by the system malloc() or GT.M gtm_malloc()). Application developers must choose the method best suited to their needs.

Matching Alternative Patterns

GT.M allows the definition of unique patterns for use with the pattern match operator, in place of, or in addition to, the standard C, N, U, L, and P. You can redefine existing pattern codes (patcodes), or add new ones. These codes are defined in a specification file. The format is described in the next section.

Pattern Code Definition

This section explains the requirements for specifying alternative pattern codes. These specifications are created as a table in a file which GT.M loads at run time.

Use the following keywords to construct your text file. Each keyword must:

• Appear as the first non-whitespace entry on a line.

• Be upper case.

The table names also must be uppercase. The patcodes are not case-sensitive.

PATSTART indicates the beginning of the definition text and must appear before the first table definition.

PATTABLE indicates the beginning of the table definition. The keyword PATTABLE is followed by whitespace, then the table name. The text file can contain multiple PATTABLEs.

PATCODE indicates the beginning of a patcode definition. The keyword PATCODE is followed by whitespace, then the patcode identifying character. On the next line enter a comma-delimited list of integer codes that satisfy the patcode. A PATCODE definition is always included in the most recently named PATTABLE. A PATTABLE can contain multiple PATCODEs.

PATEND indicates the end of the definition text; it must appear after the last table definition.

To continue the comma-delimited list on multiple lines, place a dash (-) at the end of each line that is not the last one in the sequence. To enter comments in the file, begin the line with a semi-colon (;).

The following example illustrates a possible patcode table called "NEWLANGUAGE," The example has definitions for patcodes "S," which would be a non-standard pattern character, and "L," which would substitute alternative definitions for the standard "L" (or lower case) pattern characters.

Example:

```
PATSTART
  PATTABLE NEWLANGUAGE
  PATCODE S
    144,145,146,147,148,149,150
  PATCODE L
    230,231,232,233,234,235,236,237,238,239,240,241-,242,243,244,245,246,247,248,249,250,251,252,253,254,255
PATEND
```

Be mindful of the following items as you define your patcode table.

• GT.M loads a table name can only be loaded once during an invocation of a process. Changes a loaded table do not apply to running processes that have already reference that table.

• The table name "M" is a reserved designation for standard M, which is included in the GT.M run-time library.

• Standard patcodes A and E cannot be explicitly redefined. A is always the union of codes U and L; E always designates the set of all characters.

• The C pattern code you define is used by GT.M to determine those characters which are to be treated as unprintable. All characters not defined as C are treated as printable.

• In UTF-8 mode, M standard patcodes (A,C,L,U,N,P,E) work with Unicode® characters. Application developers can neither change their default classification nor define the non-standard patcodes ((B,D,F-K,M,O,Q-T,V-X) beyond the ASCII subset. This means that the pattern tables cannot contain characters with codes greater than the maximum ASCII code 127.

PatternCode Selection

To establish a default patcode table for a database define the environment variable:

```
$ gtm_pattern_file=pathname
$ export gtm_pattern_file
```

where filename is the text file containing the patcode table definition, and

```
$ gtm_pattern_table=tablename
$ export gtm_pattern_table
```

where tablename is the name of the patcode table within the file pointed to by gtm_pattern_file.

> ⚠️ **Warning**
>
> GT.M performs operations on literals at compile time and the pattern codes settings may have an impact on such operations. Therefore, it is safest to either always compile with the same pattern code settings as those used at runtime. If changes to pattern codes are required at run time, "hide" any patterns used on literal expressions from the compiler (which are uncommon) using XECUTE commands or indirection.

Within an active process, the patcode table is established using the M VIEW command and the %PATCODE utility. Before invoking the %PATCODE utility, you may use VIEW to load pattern definition files for GT.M. The required keyword and value are:

```
VIEW "PATLOAD":"pathname"
```

This allows you to use the %PATCODE utility or the VIEW command to set current patcode table. The format of the VIEW command to set the patcode table is:

```
VIEW "PATCODE":"tablename"
```

This is equivalent to set ^%PATCODE explained below.

For more information, refer to "%PATCODE " (page 508)in the Utilities Chapter of this manual.

%PATCODE has the following extrinsic entry points:

```
set^%PATCODE(tn)
```

sets the current patcode table to the one having the name specified by tn, in the defined file specification.

Example:

```
GTM>Write $$set^%PATCODE("NEWLANGUAGE")
1
```

If there is no table with that name, the function returns a false (0) and does not modify the current patcode table.

```
get^%PATCODE
```

returns the current patcode table name.

Example:

```
GTM>Write $$get^%PATCODE
```

NEWLANGUAGE

Chapter 13. Error Processing

Revision History		
Revision V7.1-004	27 June 2024	• In "Program to Record Information on an Error using $ZTRAP" (page 586), Added WFR,BUS,BTS,STG,KTG,ZTG,DEXA,GLB,JNL,MLK,PI to the ZWR output
Revision V7.1-002	19 September 2023	• In "Program to Record Information on an Error using $ZTRAP" (page 586), Add the new $ZICUVER ISV
Revision V7.1-001	26 June 2023	• In "Program Handling of Errors" (page 566), add ZLINK and auto-ZLINK to the $ZCSTATUS entry
Revision V7.0-002	23 March 2022	• In "Input/Output Errors" (page 573), Clarify CTRAP usage
Revision V6.3-007	04 February 2019	• In "Program to Record Information on an Error using $ZTRAP" (page 586), add $ZTIMEOUT and $ZAUDIT to $zjobexam() output
Revision V6.3-005	29 June 2018	• In "Program to Record Information on an Error using $ZTRAP" (page 586), update ^EP13 and ^ERR examples for V6.3-005.
Revision V6.3-003	12 December 2017	• In "Processing Run-time Errors" (page 564), add paragraph on the precompilation of traps
Revision V6.3-001	20 March 2017	• In "Pattern Code Selection" (page 560), added a compiler warning. • In "Program to Record Information on an Error using $ZTRAP" (page 586), Update output
Revision V6.0-003	24 February 2014	In "Run-time Errors Outside of Direct Mode" (page 565), added a note about the gtm_etrap environment variable.
Revision V6.0-001	21 March 2013	In "Choosing $ETRAP or $ZTRAP" (page 571), added a note about handling non-fatal errors.

This chapter describes GT.M features and techniques for handling errors. Errors in programs may be classified as "predictable" meaning foreseen, or "unpredictable" meaning unforeseen.

M programs may attempt to recover from predictable errors. Device errors that can be remedied by an operator are the most common class of errors for which recovery provides a large benefit. Predictable errors from which the program does not attempt to recover are generally treated the same as unpredictable errors.

A typical application handles unpredictable errors by logging as much information about the error as the designer considers useful, then terminating or restarting the application from a known point.

Because GT.M invokes error handling when things are not normal, careful design and implementation of error handling are required to minimize the impact of errors and the cost of subsequent prevention.

The GT.M compiler detects and reports syntax errors at:

- Compile time while producing the object module from a source file.

- Run time while compiling code for M indirection and XECUTEs.

- Run time when the user is working in Direct Mode.

The GT.M run-time system:

- Recognizes and reports execution errors when they occur.

- Reports errors flagged by the compiler when they fall in the execution path.

Compile Time Error Message Format

To understand the compile-time error message format, consider this incorrect source line:

```
S=B+C
```

If this were line 7 of a source file ADD2.m, the compiler reports the compile-time error with the message:

```
S=B+C
    ^-----
At column 4, line 7, source module ADD2
 Variable expected in this context
```

The compile-time error message format consists of three lines. The first two lines tell you the line and location where the error occurred. The last line describes the M syntax error. The positioning accuracy of the carat pointing to the location of an issue in a source line depends on your terminal settings, particularly in UTF-8 mode where character widths are not uniform. If the line exceeds the terminal width as understood by GT.M, it replaces the carat line with an ARROWNTDSP error.

If you requested a listing file, it contains the same information and looks as follows:

```
.
.
6 .   .   .
7 S=B+C
    ^-----
 Variable expected in this context
8 . . .
.
.
```

Processing Compile TimeErrors

At compile-time, the compiler stops processing a routine line as soon as it detects the first error on that line. By default, the compiler displays the line in error on stderr, and also in a listing file when the compiler options include -list. By default, the compiler processes the remaining source lines until it exceeds the maximum error count of 127.

The compile-time error message format displays the line containing the error and the location of the error on the line. The error message also indicates what was incorrect about the M statement. For more information on the error message format, refer to the GT.M Message and Recovery Procedures Reference Manual.

You may correct compile-time errors immediately by activating an editor and entering the correct syntax in the source program. Because several errors may occur on a line, examine the line carefully to avoid compiling the routine several times.

The MUMPS command qualifier -ignore, which is the default, instructs GT.M to produce an object file even if the compiler detects errors in the source code. As long as the execution path does not encounter the compile-time errors, the GT.M run-time system executes the compiled-as-written routine. You may take advantage of this feature to exercise some parts of your program before correcting errors detected by the compiler.

Run-time Error Message Format

To understand the run-time error message format, consider this short program printsum.m:

```
     SET A=17
GO   SET B=21
     WRITE A+C
```

When you try to execute this program, the last statement causes an error since the variable C is undefined. If $ETRAP="B", GT.M displays the run-time error message:

```
$ mumps -run printsum
%GTM-E-UNDEF, Undefined local variable: C
At MUMPS source location GO+1^printsum
GTM>
```

GT.M informs you of the error (Undefined local variable) and where in the routine the error occurred (GO+1). Note that the run-time system displays the GTM> prompt, indicating that the process has entered Direct Mode. GT.M places run time error information in the intrinsic special variables $ECODE and $ZSTATUS.

Compile-time error messages may appear at run time. This is because errors in indirection and the compile-as-written feature leave errors that are subsequently reported at run time.

The GT.M utilities use portions of the run-time system and therefore may issue run-time errors as well as their own unique errors.

Processing Run-time Errors

GT.M does not detect certain types of errors associated with indirection, the functioning of I/O devices, and program logic until run-time. Also, the compile-as-written feature may leave errors which GT.M reports at run-time when it encounters them in the execution path. At run-time, GT.M reports any error encountered to stderr. The run-time system suspends normal execution of the routine as soon as it detects an error.

GT.M responds to errors differently depending on whether it encounters them in Direct Mode (at the command line) or during normal program execution.

When an executing GT.M image encounters an error:

• if Direct Mode is active at the top of the invocation stack, GT.M stays in Direct Mode.

- otherwise, if the error comes from a device that has an EXCEPTION, GT.M executes the EXCEPTION string.

- otherwise, if $ETRAP'="" GT.M transfers control to the code defined by $ETRAP as if it had been inserted at the point of the error, unless $ECODE'="", in which case it executes a TROLLBACK:$TLEVEL followed by a QUIT:$QUIT "" QUIT.

- otherwise, if $ZTRAP'="" GT.M executes $ZTRAP.

- otherwise, GT.M performs a QUIT:$QUIT "" QUIT and reissues the error at the new stack level, if no other error traps ($ETRAP or $ZTRAP) are uncovered by decending the stack, GT.M reports the error on the principal device and terminates the image.

After the action, if any, invoked by $ETRAP, $ZTRAP or EXCEPTION:

- if the process ends in Direct Mode – as a result either of performing a BREAK in the executed string or of starting in Direct Mode – GT.M reports the error on the principal device.

- otherwise, if the executed string contains an unstacked transfer of control, the only implicit behavior is that as long as $ECODE'="" and $ZTRAP'="" an attempt to QUIT from the level of the current error causes that error to be reissued at the new stack level.

- otherwise, if $ETRAP'="" GT.M performs a QUIT:$QUIT "" QUIT and reissues the error at the new stack level.

- otherwise, $ZTRAP must contain code and GT.M retries the line of M on which the error occurred.

GT.M checks the syntax of code assigned to $ETRAP, $ZSTEP, $ZTRAP, and EXCEPTION at the time they are specified. Note that $ZTRAP and EXCEPTION are subject to gtm_ztrap_form, and, if that specifies entryref or adaptive, GT.M does not check the syntax. Also, the environment variables $gtm_etrap, $gtm_trigger_etrap, and $gtm_zstep provide ways of setting some of the ISVs, so their values are verified at process initiation. Further, a SET $ETRAP uses a temporary default value of "IF $ZJOBEXAM" when shifting from $ZTRAP to $ETRAP in case the specified value has compilation errors.

Run-time Errors inDirect Mode

When GT.M detects an error in Direct Mode, it reports the error with a message and leaves the process at the GTM> prompt.

Example:

```
GTM>ZW
ZW
^_____
%GTM-E-INVCMD, Invalid command keyword encountered
GTM>
```

In Direct Mode, GT.M provides access to the RECALL command. RECALL allows you to retrieve a Direct Mode command line with a minimum of typing. The GT.M line editor allows you to make quick changes or corrections to the command line. For more information on RECALL and the line editor, see Chapter 4: "Operating and Debugging in Direct Mode" (page 50).

Run-time Errors Outside of Direct Mode

If GT.M encounters an error outside of code entered in Direct Mode, GT.M executes the $ETRAP or $ZTRAP special variable, if either of them have a length greater than zero, which only one can have at a given point in time.

The $ETRAP and $ZTRAP special variables specifiy an action that GT.M should perform when an error occurs during routine execution. $ETRAP and $ZTRAP can establish one or more error handling "actions".

> **Note**
>
> The environment variable gtm_etrap specifies an initial value of $ETRAP to override the default value of "B" for $ZTRAP as the base level error handler. The gtmprofile script sets gtm_etrap to "Write:(0=$STACK) ""Error occurred: "",$ZStatus,!" which you can customize to suit your needs. For more information, refer to "Processing Errors from Direct Mode and Shell" (page 48).

Program Handling of Errors

GT.M provides the error handling facilities described in the M standard. In addition, GT.M provides a number of extensions for error handling. Both are discussed in the following sections. The following table summarizes some of the tools, which are then described in more detail within the context of various techniques and examples.

Summary of GT.M Error-Handling Facilities	
EXTENSION	**EXPLANATION**
OPEN/USE/CLOSE EXCEPTION	Provides a deviceparameter specifying an XECUTE string or entryref that GT.M invokes upon encountering a device-related exception condition.
MUMPS -list ZLINK :"-list"	Creates a listing file of all the errors detected by the compiler and detects syntax errors. Useful in the process of re-editing program to correct errors.
ZGoto	Provides for removing multiple levels from the M invocation stack.
ZMESSAGE	Creates or emulates arbitrary errors.
$STACK	Contains the current level of M execution stack depth.
$STACK()	Returns values describing aspects of the execution environment.
$ECODE	Contains a list of error codes for "active" errors; these are the errors that have occurred, but have not yet been cleared.
$ESTACK	Contains an integer count of M virtual machine stack levels that have been activated and not removed, since the last time $ESTACK was NEW'd.
$ETRAP	Contains a string value that GT.M invokes when an error occurs during routine execution.
$QUIT	Indicates whether the current block of code was called as an extrinsic function or a subroutine.
$ZCSTATUS	Holds the value of the status code for the last compilation performed by a ZCOMPILE, ZLINK or auto-ZLINK.
$ZEDIT	Holds the value of the status code for the last edit session invoked by a ZEDIT command.
$ZEOF	Holds the value '1' (TRUE) if the last READ on the current device reached end-of-file, otherwise holds a '0' (FALSE).
$ZERROR	Contains a string supplied by the application, typically one generated by the code specified in $ZYERROR.

Summary of GT.M Error-Handling Facilities	
EXTENSION	**EXPLANATION**
$ZLEVEL	Contains current level of DO/EXECUTE nesting ($STACK+1).
$ZMESSAGE()	Translates a UNIX/GT.M condition code into text form.
$ZSTATUS	Contains the error condition code and location of last exception condition occurring during routine execution.
$ZTRAP	Contains an XECUTE string or entryref that GT.M invokes upon encountering an exception condition.
$ZYERROR	Contains an entryref to invoke when an error occurs; typically used to maintain $ZERROR.

$ECODE

The value of $ECODE is a string that may reflect multiple error conditions. As long as no error has occured, the value of $ECODE is equal to the empty string.

$ECODE contains a list of errors codes for "active" errors - the error conditions which are not yet resolved. If there are no active errors, $ECODE contains the empty string. The value of $ECODE can be SET.

The most recent error in $ECODE appears first, the oldest last. If the error is defined by the M standard, the code starts with an "M", GT.M error codes including those provided by OS services start with "Z", and application defined codes must start with "U". Every code is separated by a coma (,) and there is always a coma at the beginning and at the end of a list. GT.M provided codes are those reported in $ZSTATUS, interpreted by $ZMESSAGE() and recognized as arguments to ZMESSAGE command. When GT.M supplies a standard error code in $ECODE, it also supplies a corresponding 'Z' code.

> **Note**
>
> See "$ECode" (page 294) for a detailed description of $ECODE.

Example (setting $ECODE):

```
SET $ECODE="" ;sets $ECODE to the empty string
SET $ECODE=",M20," ;an ANSI M standardized error code
SET $ECODE=",U14," ;user defined error code
SET $PIECE($ECODE,",",2)="Z3," ;insert a non-ANSI error code
SET $PIECE($ECODE,",",$LENGTH($ECODE,",")+1)="An..," ;append
```

Standard Error processing affects the flow of control in the following manner. Detection of an error causes GOTO implicit subroutine. When $ECODE="", the implicit subroutine is $ETRAP and QUIT:$QUIT "" QUIT. Otherwise the implicit subroutine is $ETRAP followed by TROLLBACK:$TLEVEL and then QUIT:$QUIT "" QUIT.

The QUIT command behaves in a special fashion while the value of $ECODE is non-empty. If a QUIT command is executed that returns control to a less nested level than the one where the error occurred, and the value of $ECODE is still non-empty, first all normal activity related to the QUIT command occurs (especially the unstacking of NEWed variables) and then the current value of $ETRAP is executed. Note that, if $ETRAP had been NEWed at the current or intervening level, the unstacked value of $ETRAP is executed.

SETting $ECODE to an invalid value is an error. SETting $ECODE to a valid error behaves like detection of error. SETting $ECODE="" does not cause a change in the flow, but effects $STACK(), subsequent $QUITs and errors.

> **Note**
>
> To force execution of an error trap or to flag a user-defined error ("U" errors), make the value of $ECODE non-empty:
>
> ```
> SET $ECODE=",U13-User defined error trap,"
> ```

> **Note**
>
> The value of $ECODE provides information about errors that have occurred since the last time it was reset to an empty string. In addition to the information in this variable, more detailed information can be obtained from the intrinsic function $STACK. For more information, see the section on "$STack()" (page 238).

$ZSTATUS Content

$ZSTATUS contains a string value specifying the error condition code and location of the last exception condition that occurred during routine execution.

> **Note**
>
> For further details, see "$ZStatus" (page 327).

$ZERROR and $ZYERROR

After an error occurs, if $ZYERROR is set to a valid entryref that exists in the current environment, GT.M invokes the routine at that entryref with an implicit DO before returning control to M code specified by a device EXCEPTION, $ETRAP or $ZTRAP. It is intended that the code invoked by $ZYERROR use the value of $ZSTATUS to select or construct a value to which it SETs $ZERROR.

If $ZYERROR is empty, $ZYERROR="unprocessed $ZERROR, see $ZSTATUS".

If there is a problem with the content of $ZYERROR or if the execution of the code it invokes, GT.M sets $ZERROR=$ZSTATUS for the secondary error and terminates the attempt to use $ZYERROR. During code evoked by $ZYERROR, the value of $ZERROR is the empty string.

$ETRAP Behavior

If, at the time of any error, the value of $ETRAP is non-empty, GT.M proceeds as if the next instruction to be excuted were the first one on "the next line" and the code on that next line would be the same as the text in the value of $ETRAP. Furthermore, GT.M behaves as if the line following "the next line" looks like:

```
QUIT:$QUIT "" QUIT
```

When SET assigns a value to $ETRAP, the new value replaces the previous value, and if $ZTRAP was not empty (in control), the value of $ZTRAP becomes equal to the empty string without being stacked.

Nesting $ETRAP and using $ESTACK

When you need to set up a stratified scheme where one level of subroutines use one error trap setting and another more nested subroutine uses a different one; the more nested subroutine must NEW $ETRAP. When $ETRAP is NEWed, its old value is saved and copied to the current value. A subsequent SET $ETRAP=<new-value> then establishes the error trapping code for the current execution level.

The QUIT command that reverts to the calling routine causes the NEWed values to be unstacked, including the one for $ETRAP.

If an error occurs while executing at the current execution level (or at an execution level farther from the initial base stack frame), GT.M executes the code from the current $ETRAP. Unless a GOTO or ZGOTO in $ETRAP or any code it invokes redirects the flow of execution, when the execution of the $ETRAP code completes, control reverts to the implicit QUIT command, which returns to the routine that invoked the code that encountered the error. At this time, the QUIT reinstates any prior value of $ETRAP.

While at the more nested execution level(s), if an error occurs, GT.M executes the code from the current $ETRAP. After the QUIT to a less nested level, GT.M invokes the code from the now current $ETRAP. The current $ETRAP may be different from the $ETRAP at the time of the error due to unstacking. This behavior continues until one of the following possible situations occur:

- $ECODE is empty. When the value of $ECODE is equal to the empty string, error processing is no longer active, and normal processing resumes.

- A QUIT reaches an execution level where the value of $ETRAP is empty ($ZTRAP might be non-empty at that level). When the values of both $ZTRAP and $ETRAP are equal to the empty string, no error trapping is active and the QUIT repeats until it unstacks a $ETRAP or $ZTRAP.

- The stack is reduced to an empty state. When there is no previous level left to QUIT into, GT.M returns to the operating system level shell. A frame that is in direct mode stops the process by putting the user back into the Direct Mode shell.

When dealing with stratified error trapping, it is important to be aware of two additional intrinsic variables: $STACK and $ESTACK. The values of both of these variables indicate the current execution level. The value of $STACK is an "absolute" value that counts from the start of the GT.M process, whereas the value of $ESTACK restarts at zero (0) each time $ESTACK is NEWed.

It is often beneficial to NEW both $ETRAP and $ESTACK a the same time.

$ZTRAPBehavior

If, at the time of any error, the value of $ZTRAP is non-empty, GT.M uses the $ZTRAP contents to direct execution of the next action.Refer to the $ZTRAP section in Chapter 8: "Intrinsic Special Variables" (page 291).

By default, execution proceeds as if the next instruction to be executed were the first one on "the next line", and the code on that next line would be the same as the text in the value of $ZTRAP. Unless $ZTRAP or any code it invokes issues a GOTO or ZGOTO, after GT.M has executed the code in $ZTRAP, GT.M attempts to execute the line with the error again. When a value is assigned to $ZTRAP, the new value replaces the previous value. If the value of $ETRAP is a non-empty one, $ETRAP is implicitly NEWed, and the value of $ETRAP becomes equal to the empty string; this ensures that at most one of $ETRAP and $ZTRAP is not the empty string. If the environment variable gtm_ztrap_new evaluates to Boolean TRUE (case insensitive string "TRUE", or case insensitive string "YES", or a non-zero number), $ZTRAP is NEWed when $ZTRAP is SET; otherwise $ZTRAP is not stacked when it is SET.

Other than the default behavior, $ZTRAP settings are controlled by the environment variable gtm_ztrap_form as described in the following table.

gtm_ztrap_form	$ZTRAP and EXCEPTION Behavior
code	Content is code executed after the error; in the absence of GOTO, ZGOTO, or QUIT, execution resumes at the beginning of the line containing the error - note that the default behavior tends to create an indefinite loop.
entryref	Content is an entryref to which control is transferred by an implicit GOTO
adaptive	If content is valid code treat it as described for "code", otherwise attempt to treat it as an entryref
popentryref	Content is entryref - remove M virtual stack levels until the level at which $ZTRAP was SET, then GOTO the entryref; the stack manipulation occurs only for $ZTRAP and not for EXCEPTION
popadaptive	If content is valid code treat it as described for code, otherwise attempt to treat it as an entryref used as described for popentryref

Although the "adaptive" and "popadaptive" behaviors permit mixing of two behaviors based on the current value of $ZTRAP, the $ZTRAP behavior type is selected at process startup from gtm_ztrap_form and cannot be modified during the life of the process.

> Like $ZTRAP values, invocation of device EXCEPTION values, with the exception noted, follow the pattern specified by the current gtm_ztrap_form setting.

Differences between$ETRAP and $ZTRAP

The activation of $ETRAP and $ZTRAP are the same, however there are a number of differences in their subsequent behavior.

For subsequent errors the then current $ZTRAP is invoked, while with $ETRAP, behavior is controlled by the state of $ECODE. This means that when using $ZTRAP, it is important to change $ZTRAP, possibly to the empty string, at the beginning of the action in order to protect against recursion caused by any errors in $ZTRAP itself or in the code it invokes.

If there is no explicit or implicit GOTO or ZGOTO in the action, once a $ZTRAP action completes, execution resumes at the beginning of the line where the error occurred, while once a $ETRAP action completes, there is an implicit QUIT. This means that $ZTRAP actions that are not intended to permit a successful retry of the failing code should contain a GOTO, or more typically a ZGOTO. In contrast, $ETRAP actions that are intended to cause a retry must explicitly reinvoke the code where the error occurred.

For QUITs from the level at which an error occurred, $ZTRAP has no effect, where $ETRAP behavior is controlled by the state of $ECODE. This means that to invoke an error handler nested at the lower level, $ZTRAP actions need to use an explicit ZMESSAGE command, while $ETRAP does such invocations implicitly unless $ECODE is SET to the empty string.

$ZTRAP InteractionWith $ETRAP

It is important to be aware of which of the trap mechanisms is in place to avoid unintended interactions, and aware of which conditions may cause a switch-over from one mode of error handling to the other.

When a SET command assigns a value to either $ZTRAP or $ETRAP, GT.M examines the value of the other error handling variable. If the other value is non-empty, GT.M executes an implicit NEW command that saves the current value of that variable, and then assigns that variable to the empty string, then makes the requested assignment effective.

For example, re-setting $ETRAP is internally processed as:

```
NEW:$LENGTH($ZTRAP) $ZTRAP $ETRAP SET $ETRAP=code
```

Whereas, SET $ZTRAP=value is internally processed as:

```
NEW:$LENGTH($ETRAP) $ETRAP SET:$LENGTH($ETRAP)="" SET $ZTRAP=value
```

Note that, after saving the prior value, GT.M ensures the superseded $ETRAP or $ZTRAP implicitly gets the value of the empty string. As a result, at most one of the two error handling mechanisms can be effective at any given point in time.

If an error handling procedure was invoked through the $ETRAP method, and the value of $ECODE is non-empty when QUITing from the level of which the error occurred, the behavior is to transfer control to the error handler associated with the newly unstacked level. However, if the QUIT command at the end of error level happens to unstack a saved value of $ZTRAP (and thus cause the value of $ETRAP to become empty), the error handling mechanism switches from $ETRAP-based to $ZTRAP-based.

> **Note**
>
> At the end of an error handling procedure invoked through $ZTRAP, the value of $ECODE is not examined, and this value (if any) does not cause any transfer to another error handling procedure. However, if not cleared it may later trigger a $ETRAP unstacked by a QUIT.

Choosing$ETRAP or $ZTRAP

Making a choice between the two mechanisms for error handling is mostly a matter of compatibility. If compatibility with existing GT.M code is important, and that code happens to use $ZTRAP, then $ZTRAP is the best effort choice. If compatibility with code written in MUMPS dialects from other vendors is important, then $ETRAP or a non-default form of $ZTRAP probably is the better choice.

When no pre-existing code exists that favors one mechanism, the features of the mechanisms themselves should be examined.

Almost any effect that can be achieved using one mechanism can also be achieved using the other. However, some effects are easier to achieve using one method, and some are easier using with the other.

If the mechanisms are mixed, or there is a desire to refer to $ECODE in an environment using $ZTRAP, it is recommended to have $ZTRAP error code SET $ECODE="" at some appropriate time, so that $ECODE does not become cluttered with errors that have been successfully handled.

> **Note**
>
> A device EXCEPTION gets control after a non-fatal device error and $ETRAP/$ZTRAP get control after other non-fatal errors.

Example 1: Returning control to a specific execution level

The following example returns control to the execution level "level" and then to an error processing routine "proc^prog".

With $ZTRAP: Set $ZTRAP="ZGOTO "_level_":proc^prog"

With $ETRAP: Set $ETRAP="Quit:$STACK>"_level_" Do proc^prog"

Note that, ZGOTO can be used with $ETRAP and $STACK with $ZTRAP. Alternatively if $ESTACK were NEWed at LEVEL:

```
Set $ETRAP="Quit:$ESTACK>0 Do proc^prog"
```

Example 2: Ignoring an Error

With $ZTRAP: Set $ZTRAP="Quit"

With $ETRAP: Set $ETRAP="Set $ECODE="""" Quit"

Note that, while it is not necessary to SET $ECODE="" when using $ZTRAP it is advisable to do it in order to permit mixing of the two mechanisms.

Example 3: Nested Error Handlers

With $ZTRAP: New $ZTRAP Set $ZTRAP=...

With $ETRAP: New $ETRAP Set $ETRAP=...

> **Note**
>
> In both cases, QUITting to a lower level may effectively make the other mechanism active.

Example 4: Access to "cause of error"

With $ZTRAP: If $ZSTATUS[...

With $ETRAP: If $ECODE[...

> **Note**
>
> The value of $ZSTATUS reflects only the most recent error, while the value of $ECODE is the cumulative list of all errors since its value was explicitly set to empty. Both values are always maintained and can be used with either mechanism.

Error Processing Cautions

$ETRAP and $ZTRAP offer many features for catching, recognizing, and recovering from errors. Code within an error processing subroutines may cause its own errors and these need to be processed without causing an infinite loop (where an error is caught, which, while being processed causes another error, which is caught, and so on).

During the debugging phase, such loops are typically the result of typographical errors in code. Once these typographical errors are corrected, the risk remains that an error trapping subroutine was designed specifically to deal with an expected condition; such as the loss of a network connection. This then creates an unexpected error of its own, such as:

- a device that had not yet been opened because the loss of network connectivity occured sooner than expected

- an unexpected data configuration caused by the fact that an earlier instance of the same program did not complete its task for the same reason

> **Note**
>
> It is important to remain aware of any issues that may arise within an error trapping procedure, and also of the conditions that might cause the code in question to be invoked.

$ETRAP is recursively invoked if it invokes a GOTO or a ZGOTO and the error condition persists in the code path and the code SETs $ECODE="". $ZTRAP is recursively invoked if the error condition persists in the code path.

Input/OutputErrors

When GT.M encounters an error in the operation of an I/O device, GT.M executes the EXCEPTION deviceparameter for the OPEN/USE/CLOSE commands. An EXCEPTION deviceparameter specifies an action to take when an error occurs in the operation of an I/O device. The form of the EXCEPTION action is subject to the gtm_ztrap_form setting described for $ZTRAP, except that there is never any implicit popping with EXCEPTION actions. If a device has no current EXCEPTION, GT.M uses $ETRAP or $ZTRAP to handle an error from that device.

GT.M provides the option to:

- Trap or process an exception based on device error.

- Trap or process an exception based on terminal input.

An EXCEPTION based on an error for the device applies only to that device, and provides a specific error handler for a specific I/O device.

The CTRAP deviceparameter for USE establishes <CTRL-n> where 0<=n<=31 as an interrupting signal. When GT.M encounters <CTRL-n> with CTRAP enabled, GT.M executes the EXCEPTION deviceparamenter, or, $ETRAP or $ZTRAP if the device has no current EXCEPTION. <CTRL-C> is unique among <CTRL> characters, in that the OS recognizes it as an out-of-band signal and delivers it immediately; GT.M recognizes other <CTRL> character when they appear duing a READ.

Example:

```
GTM>ZPRINT ^EP12
EP12    WRITE !,"THIS IS ",$TEXT(+0)
        SET $ECODE="";this only affects $ETRAP
        SET $ETRAP="GOTO ET"
        ;N $ZT S $ZT="W !,""CAN'T TAKE RECIPROCAL OF 0"",*7"
        USE $P:(EXCEPTION="D BYE":CTRAP=$C(3))
        WRITE !,"TYPE <CTRL-C> TO STOP"
LOOP    FOR DO
        . READ !,"TYPE A NUMBER: ",X
        . WRITE ?20,"HAS RECIPROCAL OF: ",1/X
        . QUIT
ET      . WRITE !,"CAN'T TAKE RECIRPOCAL OF 0",*7
        . SET $ECODE=""
        QUIT
BYE     WRITE !,"YOU TYPED <CTRL-C> YOU MUST BE DONE!"
        USE $P:(EXCEPTION="":CTRAP="")
        WRITE !,"$ZSTATUS=",$ZSTATUS
        ZGOTO 1
```

```
GTM>DO ^EP12
THIS IS EP12
TYPE <CTRL-C> TO STOP
TYPE A NUMBER: 1 HAS RECIPROCAL OF: 1
TYPE A NUMBER: 2 HAS RECIRPOCAL OF: .5
TYPE A NUMBER: 3 HAS RECIPROCAL OF: .33333333333333
TYPE A NUMBER: 4 HAS RECIPROCAL OF: .25
TYPE A NUMBER: HAS RECIPROCAL OF:
CAN'T TAKE RECIPROCAL OF 0
TYPE A NUMBER:
YOU TYPED <CTRL-C> YOU MUST BE DONE!
$ZSTATUS=150372498,LOOP+1^EP12,%GTM-E-CTRAP,Character trap $C(3) encountered
GTM>
```

This routine prompts the user to enter a number at the terminal. If the user enters a zero, GT.M encounters an error and executes $ETRAP (or $ZTRAP). The action specified reports the error and returns to prompt the user to enter a number. With $ZTRAP, this is very straightforward. With $ETRAP, some care is required to get the code to resume at the proper place. The CTRAP deviceparameter establishes <CTRL-C> as a trap character. When GT.M encounters a <CTRL-C>, GT.M executes the EXCEPTION string whcih transfers control to the label BYE. At the label BYE, the routine terminates execution with an error message. Using the EXCEPTION deviceparameter with CTRAP generally simplifies $ETRAP or $ZTRAP handling.

$ZSTATUS allows the routine to find out which trap character GT.M encountered. When a routine has several character traps set, $ZSTATUS provides useful information for identifying which character triggered the trap, and thereby allows a custom response to a specific input.

Error Actions

In the following examples (and the previous one as well), $ETRAP and $ZTRAP in most cases have similar behavior. The most prominent difference is that, when $ETRAP is active, $ECODE determines whether or not a second error in an M stack level triggers an immediate implicit QUIT from that level. For additional information, see the sections on $ECODE and $ETRAP in Chapter 8: "Intrinsic Special Variables" (page 291). Because of the effect of $ECODE on the processing flow when $ETRAP is active, there is a benefit to including appropriate $ECODE maintenance in $ZTRAP related code, so that things stay well behaved when the two mechanisms are intemixed. Other differences are discussed in some of the examples.

Break on an Error

When $ZTRAP is set to a BREAK command and an error occurs, GT.M puts the process into Direct Mode. The default for $ZTRAP is a BREAK command. When developing a program, $ZTRAP="BREAK" allows you to investigate the cause of the error from Direct Mode. For information on GT.M debugging tools, see Chapter 4: "Operating and Debugging in Direct Mode" (page 50).

Example:

```
GTM>zprint ^EP1
EP1     WRITE !,"THIS IS "_$TEXT(+0)
        KILL A
BAD     WRITE A
        WRITE !,"THIS IS NOT DISPLAYED"
        QUIT

GTM>do ^EP1
THIS IS EP1%GTM-E-UNDEF, Undefined local variable: A
```

```
              At M source location BAD^EP1
GTM>ZSHOW
BAD^EP1    ($ZTRAP)
    (Direct mode)
+1^GTM$DMOD    (Direct mode)
GTM>QUIT
GTM>ZSHOW
EP1+1^EP1    (Direct mode)
+1^GTM$DMOD    (Direct mode)
GTM>
```

Because by default $ETRAP="" and $ZTRAP="B", this example does not explicitly set either $ETRAP or $ZTRAP. When the routine encounters an error at BAD^EP1, GT.M initiates Direct Mode. The ZSHOW displays the M stack that has, at the bottom, the base Direct Mode frame and, at the top, EP1 with a notation that $ZTRAP has been invoked. The QUIT command at the prompt removes EP1 from the stack.

To prevent a program such as a production image from accessing Direct Mode, assign an action other than "BREAK" to $ETRAP or $ZTRAP. The following sections discuss various alternative values for $ETRAP or $ZTRAP.

In order to prevent inappropriate access to Direct Mode, eliminate all BREAKs from the production code. If the code contains BREAK commands, the commands should be subject to a postconditional flag that is only turned on for debugging. ZBREAK serves as an alternative debugging tool that effects only the current process and lasts only for the duration of an image activation.

Unconditional Transfer on an Error

The GOTO command instructs GT.M to transfer execution permanently to another line within the routine or to another routine. When stopping to investigate an error is undesirable, use the GOTO command in $ETRAP or $ZTRAP to continue execution at some other point.

Example:

```
GTM>ZPRINT ^EP2
EP2      WRITE !,"THIS IS "_$TEXT(+0)
         SET $ECODE=""         ;this affects only $ETRAP
         SET $ETRAP="GOTO ET"        ;this implicitly stacks $ZTRAP
         ;N $ZT S $ZT="GOTO ET"  ;would give a similar result
         DO SUB1
         WRITE !,"THIS IS THE END"
         QUIT
SUB1     WRITE !,"THIS IS SUB1"
         DO SUB2
         QUIT
SUB2     WRITE !,"THIS IS SUB2"
         KILL A
BAD      WRITE A
         WRITE !,"THIS IS NOT DISPLAYED"
         QUIT
ET       ;SET $ZTRAP=""          ;if using $ZTRAP to prevent recursion
         WRITE !,"CONTINUING WITH ERROR TRAP AFTER AN ERROR"
         WRITE !,"$STACK: ",$STACK
         WRITE !,"$STACK(-1): ",$STACK(-1)
         WRITE !,"$ZLEVEL: ",$ZLEVEL
         FOR I=$STACK(-1):-1:1 DO
```

```
        . WRITE !,"LEVEL: ",I
        . SET K=10
        . FOR J="PLACE","MCODE","ECODE" DO
        . . WRITE ?K," ",J,": ",$STACK(I,J)
        . . SET K=K+20
        WRITE !,$ZSTATUS,!
        ZSHOW "S"
        SET $ECODE=""        ;this affects only $ETRAP
        QUIT

GTM>do ^EP2
THIS IS EP2
THIS IS SUB1
THIS IS SUB2
CONTINUING WITH ERROR TRAP AFTER AN ERROR
$STACK: 3
$STACK(-1): 3
$ZLEVEL: 4
LEVEL: 3   PLACE: BAD^EP2      MCODE: BAD     WRITE A  ECODE: ,M6,Z150373850,
LEVEL: 2   PLACE: SUB1+1^EP2   MCODE:         DO SUB2  ECODE:
LEVEL: 1   PLACE: EP2+4^EP2    MCODE:         DO SUB1  ECODE:
150373850,BAD^EP2,%GTM-E-UNDEF, Undefined local variable: A
ET+12^EP2
SUB1+1^EP2
EP2+4^EP2
+1^GTM$DMOD     (Direct mode)
THIS IS THE END
GTM>
```

This routine specifies a GOTO command transferring execution to the ET label when an error occurs. The $ZLEVEL special variable contains an integer indicating the M stack level.

The ZGOTO command is similar to the GOTO command, however, the ZGOTO allows the removal of multiple levels from the program stack. ZGOTO can ensure that execution returns to a specific point, such as a menu.

Example:

```
GTM>ZPRINT ^EP3
EP3     ;
MENU    WRITE !,"THIS IS MENU IN ",$TEXT(0)
        SET $ECODE=""        ;this affects only $ETRAP
        SET $ETRAP="SET $ECODE="""" ZGOTO 2"
        ;N $ZT S $ZT="ZGOTO 2" ;would give a similar result
        DO SUB1
        WRITE !,"`MENU' AFTER $ETRAP"
        WRITE !,"$STACK: ",$STACK
        WRITE !,"$ZLEVEL: ",$ZLEVEL
        QUIT
SUB1    WRITE !,"THIS IS SUB1"
        DO SUB2
        WRITE !,"THIS IS SKIPPED BY ZGOTO"
        QUIT
SUB2    WRITE !,"THIS IS SUB2"
        KILL A
BAD     WRITE A
        WRITE !,"THIS IS NOT DISPLAYED"
```

```
        QUIT

GTM>do ^EP3
THIS IS MENU IN
THIS IS SUB1
THIS IS SUB2
`MENU' AFTER $ETRAP
$STACK: 1
$ZLEVEL: 2
```

This routine instructs GT.M to reset the execution to level 2 if it encounters an error. GT.M removes all intermediate levels.

In general, coding ZGOTO level information based on $ZLEVEL provides a more robust technique than the "hard-coding" shown in the previous example.

Example:

```
GTM>ZPRINT ^EP4
EP4     WRITE !,"THIS IS "_$TEXT(+0)
        SET $ECODE=""           ;this affects only $ETRAP
        DO MAIN
        WRITE !,"THIS IS ",$TEXT(+0)," AFTER THE ERROR"
        WRITE !,"$ZLEVEL: ",$ZLEVEL
        QUIT
MAIN    WRITE !,"THIS IS MAIN"
        WRITE !,"$ZLEVEL: ",$ZLEVEL
        SET $ETRAP="ZGOTO "_$ZLEVEL_":ET"
        ;N $ZT S $ZT="ZGOTO "_$ZLEVEL_":ET ;alternative
        DO SUB1
        QUIT
SUB1    WRITE !,"THIS IS SUB1"
        WRITE !,"$ZLEVEL: ",$ZLEVEL
        DO SUB2
        QUIT
SUB2    WRITE !,"THIS IS SUB2"
        WRITE !,"$ZLEVEL :",$ZLEVEL
        KILL A
BAD     WRITE A
        WRITE !,"THIS IS NOT DISPLAYED"
        QUIT
ET      ;SET $ZTRAP="" ;if using $ZTRAP to prevent recursion
        WRITE !,"CONTINUING WITH ERROR TRAP AFTER AN ERROR"
        WRITE !,"$STACK: ",$STACK
        WRITE !,"$STACK(-1): ",$STACK(-1)
        WRITE !,"$ZLEVEL: ",$ZLEVEL
        FOR I=$STACK(-1):-1:1 DO
        . WRITE !,"LEVEL: ",I
        . SET K=10
        . FOR J="PLACE","MCODE","ECODE" DO
        . . WRITE ?K," ",J,": ",$STACK(I,J)
        . . SET K=K+20
        WRITE !,$ZSTATUS,!
        ZSHOW "S"
        SET $ECODE=""           ;this affects only $ETRAP
        QUIT
```

```
GTM>do ^EP4
THIS IS EP4
THIS IS MAIN
$ZLEVEL: 3
THIS IS SUB1
$ZLEVEL: 4
THIS IS SUB2
$ZLEVEL :5
CONTINUING WITH ERROR TRAP AFTER AN ERROR
$STACK: 2
$STACK(-1): 4
$ZLEVEL: 3
LEVEL: 4   PLACE: BAD^EP4      MCODE: BAD    WRITE A  ECODE: ,M6,Z150373850,
LEVEL: 3   PLACE: SUB1+2^EP4   MCODE:        DO SUB2 ECODE:
LEVEL: 2   PLACE: MAIN+4^EP4   MCODE:        DO SUB1 ECODE:
LEVEL: 1   PLACE: EP4+2^EP4    MCODE:        DO MAIN ECODE:
150373850,BAD^EP4,%GTM-E-UNDEF, Undefined local variable: A
ET+12^EP4
EP4+2^EP4
+1^GTM$DMOD    (Direct mode)
THIS IS EP4 AFTER THE ERROR
$ZLEVEL: 2
GTM>
```

This routine sets $ETRAP or $ZTRAP to a ZGOTO specifying the current level. When the routine encounters an error at label BAD, GT.M switches control to label ET at the level where $ETRAP (or $ZTRAP) was established. At this point in the execution, ET replaces SUB1+2^EP4 as the program stack entry for the level specified, that is, $ZLEVEL=3. The QUIT command then returns control to the level where $ZLEVEL=2.

Setting $ZTRAP for Each Level

The command NEW $ETRAP or NEW $ZTRAP stacks the current value of $ETRAP or $ZTRAP respectively and, in the case of $ZTRAP, sets the value equal to the empty string. Normally, a SET $ETRAP or $ZTRAP immediately follows a NEW $ETRAP or $ZTRAP. When GT.M encounters a QUIT command that leaves a level where $ETRAP or $ZTRAP had been NEWed, GT.M deletes any value set to the ISV after the NEW command and restores the value that the ISV held previous to the NEW. NEW $ETRAP or $ZTRAP enables the construction of error handlers corresponding to the nesting of routines. A SET $ETRAP or $ZTRAP implicitly NEWs the other variable if it does not already have the value of the empty string. This enables the interleaving of $ETRAP and $ZTRAP at different levels, although (as mentioned above) such interleaving requires that $ZTRAP handlers deal appropriately with $ECODE.

Example:

```
GTM>ZPRINT ^EP5
EP5     WRITE !,"THIS IS "_$TEXT(+0)
        SET $ECODE="";this affects only $ETRAP
        WRITE !,"STARTING $ETRAP: ",$ETRAP
        WRITE !,"STARTING $ZTRAP: ",$ZTRAP
        DO SUB1
        WRITE !,"ENDING $ETRAP: ",$ETRAP
        WRITE !,"ENDING $ZTRAP: ",$ZTRAP
        QUIT
MAIN    WRITE !,"THIS IS MAIN"
        WRITE !,"$ZLEVEL: ",$ZLEVEL
        DO SUB1
```

```
          QUIT
SUB1      WRITE !,"THIS IS SUB1"
          NEW $ETRAP SET $ETRAP="GOTO ET1"
          ;NEW $ZTRAP SET $ZTRAP="GOTO ET1" ;alternative
          WRITE !,"$ETRAP FOR SUB1: ",$ETRAP
          KILL A
BAD       WRITE A
          WRITE !,"THIS IS NOT DISPLAYED"
          QUIT
ET1       WRITE !,"ERROR TRAP 1"
          WRITE !,"$ETRAP AFTER THE TRAP: ",$ETRAP
          WRITE !,"$ZTRAP AFTER THE TRAP: ",$ZTRAP
          SET $ECODE="";this affects only $ETRAP
          QUIT

GTM>do ^EP5
THIS IS EP5
STARTING $ETRAP:
STARTING $ZTRAP: B
THIS IS SUB1
$ETRAP FOR SUB1: GOTO ET1
ERROR TRAP 1
$ETRAP AFTER THE TRAP: GOTO ET1
$ZTRAP AFTER THE TRAP:
ENDING $ETRAP:
ENDING $ZTRAP: B
GTM>
```

At SUB1, this routine NEWs $ETRAP and assigns it a value, which implicitly NEWs $ZTRAP. When the routine encounters an error at the SUB1 level, GT.M transfers control to label ET1 without modifying the value of $ETRAP or $ZTRAP. When the routine encounters a QUIT command in routine ET1, GT.M transfers control to the command after the DO that invoked ET1 and restores $ETRAP or $ZTRAP to the values they held before the NEW and the SET.

> **Note**
>
> If the transfer to ET1 was accomplished with a ZGOTO that reduced the stack level, after the trap, $ETRAP would have the value of the empty string and $ZTRAP would be "B".

Nested Error Handling

$ETRAP or $ZTRAP set to a DO command instructs GT.M to transfer execution temporarily to another line within this or another routine when it encounters an error. A QUIT command within the scope of the DO transfers control back to the code specified by the $ETRAP or $ZTRAP. When the code in the ISV terminates due to an explicit or implicit QUIT, the behavior of $ETRAP and $ZTRAP is different. When $ETRAP is in control, the level at which the error occurred is removed, and control returns to the invoking level. When $ZTRAP contains code, execution picks up at the beginning of the line with the error. A DO command within $ZTRAP is normally used for I/O errors that an operator may resolve, because a DO command permits re-execution of the line containing the error.

Example:

```
GTM>ZPRINT ^EP6
EP6       WRITE !,"THIS IS "_$TEXT(+0)
          NEW
```

```
        NEW $ZTRAP SET $ZTRAP="DO ET"
        SET (CB,CE)=0
BAD     SET CB=CB+1 WRITE A SET CE=CE+1
        WRITE !,"AFTER SUCCESSFUL EXECUTION OF BAD:",!
        SET A="A IS NOT DEFINED"
        ZWRITE
        QUIT
ET      W !,"CONTINUING WITH ERROR TRAP AFTER AN ERROR",!
        ZWRITE
        SET A="A IS NOW DEFINED"
GTM>do ^EP6
THIS IS EP6
CONTINUING WITH ERROR TRAP AFTER AN ERROR
CB=1
CE=0
A IS NOW DEFINED
AFTER SUCCESSFUL EXECUTION OF BAD:
A="A IS NOT DEFINED"
CB=2
CE=1
GTM>
```

This example sets $ZTRAP to a DO command. When the routine encounters an error in the middle of the line at label BAD, GT.M transfers control to label ET. After QUITting from routine ET, GT.M returns control to the beginning of the line at label BAD.

Example:

```
GTM>ZPRINT ^EP6A
EP6A    WRITE !,"THIS IS "_$TEXT(+0)
        NEW
        NEW $ETRAP SET $ETRAP="GOTO ET"
        SET (CB,CE)=0
BAD     SET CB=CB+1 WRITE A SET CE=CE+1
        WRITE !,"AFTER SUCCESSFUL EXECUTION OF BAD:",!
        ZWRITE
        QUIT
ET      W !,"CONTINUING WITH ERROR TRAP AFTER AN ERROR",!
        ZWRITE
        SET A="A IS NOW DEFINED"
        SET RETRY=$STACK($STACK,"PLACE")
        SET $ECODE=""
        GOTO @RETRY

GTM>DO ^EP6A
THIS IS EP6A
CONTINUING WITH ERROR TRAP AFTER AN ERROR
CB=1
CE=0
A IS NOW DEFINED
AFTER SUCCESSFUL EXECUTION OF BAD:
A="A IS NOW DEFINED"
CB=2
CE=1
RETRY="BAD^EP6A"
GTM>
```

This routine is an example of how $ETRAP handling can be coded to perform the same kind of resumtion of the original execution stream that occurs by default with $ZTRAP when there is no unconditional transfer of control.

Terminating Execution on an Error

If both $ETRAP and $ZTRAP are set to the empty string upon encountering an error, the current level is discarded and the error is reissued at the invoking level. When already at the lowest M stack level, GT.M terminates routine execution and returns control to the shell level. If $ZTRAP is used exclusively, $ZTRAP="" suppresses the unstacking of NEWed values of $ZTRAP associated with lower levels. $ETRAP values are always unstacked, however if the lowest level $ETRAP is the empty string (which it is by default when GT.M starts), GT.M performs the same termination as it does with $ZTRAP. These terminations with both ISVs empty provides a mechanism for returning to the shell with a status message when GT.M encounters an error.

Example:

```
GTM>ZPRINT ^EP7
EP7     WRITE !,"THIS IS ",$TEXT(+0)
        SET $ECODE="";this only affects $ETRAP
        SET $ETRAP="",$ZTRAP=""
        KILL A
BAD     WRITE A
        WRITE !,"THIS IS NOT DISPLAYED"
        QUIT

GTM>do ^EP7
THIS IS EP7
%GTM-E-UNDEF, Undefined local variable: A
%GTM-I-RTSLOC, At M source location BAD^EP7
$
```

GT.M issues a message describing the M error and releases control to the shell.

When the action specified by $ZTRAP results in another run-time error before changing the value of $ZTRAP, the routine may iteratively invoke $ZTRAP until a stack overflow terminates the GT.M image. SETting $ZTRAP="" at the beginning of error processing ensures that this type of infinite loop does not occur. Because $ETRAP is implicitly followed by a QUIT it does not have the tendency to recurse. While $ETRAP is resistant to recursion, it is not completely immune, because a GOTO or a ZGOTO within the same level can evade the implicit QUIT. $ETRAP error handling involving errors on more than one stack level can also be induced to recurse if $ECODE is inappropriately cleared before the errors at all levels have been properly dealt with.

Example:

```
GTM>ZPRINT ^EP8
EP8     WRITE !,"THIS IS ",$TEXT(+0)
        NEW $ZTRAP SET $ZTRAP="DO ET"
        KILL A
BAD     WRITE A
        WRITE !,"THIS IS NOT DISPLAYED"
        QUIT
ET      WRITE 2/0
        QUIT

GTM>DO ^EP8
THIS IS EP8
%GTM-E-STACKCRIT, Stack space critical
%GTM-E-ERRWZTRAP, Error while processing $ZTRAP
```

```
GTM>
```

When the routine encounters an error at label BAD, GT.M transfers control to label ET. When the routine encounters an error at label ET, it recursively does ET until a stack overflow condition terminates the GT.M image.

A set $ZTRAP="" command as soon as the program enters an error-handling routine prevents this type of "infinite" recursion.

```
GTM>zprint ^EP8A
EP8A     WRITE !,"THIS IS ",$TEXT(+0)
         SET $ECODE=""
         SET $ZTRAP="",$ETRAP="DO ET"
         KILL A
BAD      WRITE A
         WRITE !,"THIS IS NOT DISPLAYED"
         QUIT
ET       WRITE !,"CONTINUING WITH ERROR TRAP AFTER AN ERROR"
         ZSHOW "S"
         WRITE !,"HERE COMES AN ERROR IN THE TRAP CODE"
         WRITE 2/0
         QUIT


GTM>DO ^EP8A
THIS IS EP8A
CONTINUING WITH ERROR TRAP AFTER AN ERRORET+1^EP8A
BAD^EP8A     ($ZTRAP)
+1^GTM$DMOD     (Direct mode)
HERE COMES AN ERROR IN THE TRAP CODE
%GTM-E-DIVZERO, Attempt to divide by zero
GTM>
```

This demonstrates how $ETRAP behavior in this circumstance is more appropriate. Note that the $ZTRAP="" at the lowest level, prevents exection from returning to Direct Mode when the initial value of $ZTRAP ("B") is unstacked; this step takes $ZTRAP out of the equation and should be part of initialization when the intention is to use $ETRAP exclusively.

Example:

```
GTM>ZPRINT ^EP9
EP9      WRITE !,"THIS IS ",$TEXT(+0)
         SET $ZTRAP="DO ET"
         KILL A
BAD      WRITE A
         WRITE !,"THIS IS NOT DISPLAYED"
         QUIT
ET       SET $ZT=""
         WRITE !,"THIS IS THE ERROR TRAP"
ERROR    WRITE !,"HERE COMES AN ERROR IN THE ERROR TRAP"
         WRITE 2/0
         QUIT


GTM>DO ^EP9
THIS IS EP9
THIS IS THE ERROR TRAP
HERE COMES AN ERROR IN THE ERROR TRAP
%GTM-E-DIVZERO, Attempt to divide by zero
%GTM-I-RTSLOC,                At M source location ERROR+1^EP9
$
```

This routine sets the value of $ZTRAP to null as soon as the program enters the error handler. This insures program termination when an error occurs in the error handler.

Setting $ZTRAP toOther Actions

The QUIT, HALT and ZHALT commands also serve as useful $ETRAP or $ZTRAP actions.

The QUIT command terminates execution at that invocation level.

Example:

```
GTM>zprint ^EP10
EP10    WRITE !,"THIS IS ",$TEXT(+0)
        SET $ECODE="";this affects only $ETRAP
        S $ET="S $EC="""" Q" ;this implicitly stacks $ZTRAP
        ;N $ZT S $ZT="QUIT" ;would give a similar result
        DO SUB1
        QUIT
SUB1    WRITE !,"THIS IS SUB1"
        DO SUB2
        WRITE !,"THIS IS SUB1 AFTER THE ERROR WAS 'IGNORED'"
        QUIT
SUB2    WRITE !,"THIS IS SUB2"
        KILL A
BAD     WRITE A
        WRITE !,"THIS IS NOT DISPLAYED"
        QUIT

GTM>do ^EP10
THIS IS EP10
THIS IS SUB1
THIS IS SUB2
THIS IS SUB1 AFTER THE ERROR WAS 'IGNORED'
GTM>
```

This routine sets $ETRAP or $ZTRAP to the QUIT command. When the routine encounters an error at label BAD, GT.M executes the active error handling ISV. The QUIT command terminates execution of SUB2 and transfers execution back to SUB1. The WRITE displays the error message using the $ZSTATUS special variable. Because the default behavior is to QUIT after $ETRAP code completes, this technique is mostly useful with $ETRAP as a place holder to avoid the $ETRAP="" semantics when there is no action to take at the current level. With $ZTRAP, where the default behavior is to resume execution at the beginning the line that triggered the error, the QUIT is more than a placeholder.

The HALT command terminates routine execution and returns control to the shell level. Setting $ETRAP="HALT" or $ZTRAP="HALT" is similar to setting the ISV to the empty string except that the "HALT" code does not pass the error condition code back to the shell. After a HALT, $? contains zero (0).

ZHALT acts like HALT but takes and argument, which GT.M passes back to the OS shell. Note that UNIX shells typically limit return codes to a byte, so they may truncate the value of the ZHALT argument.

Example:

```
GTM>ZPRINT ^EP11
EP11    WRITE !,"THIS IS ",$TEXT(+0)
        SET $ECODE="";this affects only $ETRAP
```

```
        SET $ETRAP="HALT";this implicitly stacks $ZTRAP
        ;SET $ZTRAP="HALT";would give a similar result
        KILL A
BAD     WRITE !,A
        WRITE !,"THIS IS NOT DISPLAYED"
        QUIT

GTM>DO ^EP11
THIS IS EP11
$
```

Summary of $ETRAP & $ZTRAP Error-Handling Options

Summary of Error-Handling Options	
ERROR-HANDLING FEATURE	**DESCRIPTION AND POSSIBLE USES**
$ETRAP="BREAK" $ZTRAP="BREAK"	Returns to Direct Mode upon encountering an error that enables interactive debugging to determine the nature of the error.
$ETRAP="GOTO.." $ZTRAP="GOTO.."	Transfers control upon encountering an error and allows for continuation of execution after the error. Use with an error handling routine that may record or report an error.
$ETRAP="ZGOTO.." $ZTRAP="ZGOTO.."	Similar to GOTO, but additionally allows for removal of levels from the stack. Use to allow recovery to specific point, such as a menu.
NEW $ETRAP NEW $ZTRAP	NEW $ETRAP stacks the old value but does not change the current value, while NEW $ZTRAP stacks the old value and sets the current value to the empty string. Usually followed by a SET $ETRAP or SET $ZTRAP. After a QUIT from a given level, GT.M restores the value held prior to the NEW. Use to enable different methods of error handling at different levels within an application.
$ETRAP="DO..."	Transfers execution temporarily to another label upon encountering an error. After return from a DO, GT.M QUITs from the stack level at which the error occured. Whether control returns to the invoking code or to the trap handler at the less nested level, depends on the value of $ECODE.
$ZTRAP="DO..."	Transfers execution temporarily to another label upon encountering an error. When GT.M returns from a DO and completes the $ZTRAP action, execution continues at the beginning of the line containing the error and re-executes the entire line containing the error. Use with I/O device errors where operator may intervene to correct the error condition.
$ZTRAP=""	Returns to shell with the Status Code and terminates execution. If SET in error handling routines, prevents infinite loops. Prevents access to Direct Mode. Use in production code when the invoking shell needs to test $?.
$ETRAP="SET $ECODE=""""" $ZTRAP="QUIT"	Terminates execution at that level upon encountering an error, and returns to the invocation level at the point immediately following the invocation. Use to ignore errors on a particular level and continue executing.
$ZTRAP="HALT"	Returns to the shell as if normal termination occurred. Avoids access to Direct Mode. Use in production code when the invoking shell does not need to examine the exit status of the GT.M process.

Errors in $ZTRAP

If $ZTRAP contains invalid source code, GT.M displays an error message and puts the process into Direct Mode.

If the action specified by $ZTRAP results in another run-time error before changing the value of $ZTRAP, it may result in a loop that iteratively invokes $ZTRAP until a stack overflow terminates the GT.M image. Keep $ZTRAP simple and take special care to debug exception handling.

> **Note**
>
> An error in $ETRAP code triggers an implicit TROLLBACK:$TLEVEL QUIT:$QUIT "" QUIT.

Recording Information about Errors

GT.M provides a number of standard features and extensions to examine and record information about an error condition.

The extensions are:

- ZSHOW

- ZWRITE

- $ECODE

- $STACK

- $STACK()

- $ZSTATUS

- $ZJOBEXAM()

- $ZLEVEL

The ZSHOW command displays information about the current M environment. A ZSHOW argument may contain an expression that contains codes selecting one or more types of information for output.

A: selects auto-relink information

B: selects ZBREAK information

C: provides the list of loaded external call packages and their routines. ZSHOW "C" does not report packages that are accessible but have not been accessed by the process.

D: selects open device information

G: selects global statistic information

I: selects intrinsic special variables

L: selects locks held by the process

R: selects the M stack but with routine hashes

S: selects the M stack

V: selects local variables

*: selects all possible ZSHOW information except A.

A ZSHOW with no argument displays the M stack on the current device. It lists the program stack from initiation to the current execution level.

The ZWRITE command prints the current value of defined variables. ZWRITE provides a tool for examining or saving variable context. ZWRITE and ZSHOW can only display the current local variables, not any local variable states that have been protected by NEW commands, or appearance in an invoked formallist. A WRITE may also display current global variables.

The $ECODE special variable contains a M standardized/user defined/GT.M specific error code.

The $STACK special variable contains the current level of M execution stack depth.

The $STACK() function returns strings describing aspects of the execution environment.

The $ZLEVEL special variable maintains an integer that indicates the level of nesting of DO and XECUTE commands. $ZLEVEL always contains an integer count of the number of levels displayed by issuing a ZSHOW "S" in that context.

The $ZJOBEXAM() function returns a string indicating the full path to the file where it stored a process context dump.

The $ZSTATUS special variable records the error condition code and location of the last error condition during execution.

For I/O operations, GT.M uses the $ZA, $ZB and $ZEOF special variables. $ZA contains a status determined by the last read on the current device.

To simplify record keeping, an application may set $ZTRAP to an error-handling routine that records information about an error. The next section provides an example of a routine ERR.m that does this.

Program to Record Information on an Error using $ZTRAP

```
GTM>ZPRINT ^ERR
ERR0;;RECORD CONTENT OF AN ERROR
;
RECORD  SET $ZTRAP="GOTO OPEN"
        ZSHOW "*":^ERR($J,$H)
        GOTO LOOPT;$H might change
LOOPV   ZSHOW "V":^ERR($J,$H,"VL",$ZLEVEL)
LOOPT   IF $ZLEVEL>1 ZGOTO $ZLEVEL-1:LOOPV
STACK   SET $ZTRAP="GOTO WARN"
        SET %ERRVH=$H;can cause error if memory low
        SET ^ERR($J,%ERRVH,"$STACK")=$STACK
        SET ^ERR($J,%ERRVH,"$STACK",-1)=$STACK(-1)
        FOR %ERRVI=$STACK(-1):-1:1 DO
        . SET %ERRVK=""
        . FOR %ERRVJ="PLACE","MCODE","ECODE" DO
        . . SET %ERRVK=%ERRVK_$STACK(%ERRVI,%ERRVJ)_"|~|"
        . SET ^ERR($J,%ERRVH,"$STACK",%ERRVI)=%ERRVK
        GOTO WARN
OPEN    SET $ZTRAP="GOTO OPEN1"
        SET %ERRIO=$IO,%ERRZA=$ZA,%ERRZB=$ZB,%ERRZE=$ZEOF
        SET %ERRVF="REC.ERR"
        SET %ERRVF=$ZDATE($H,"YEARMMDD2460SS")_"_"_$J_".ERR"
        OPEN %ERRVF:NEWVERSION
        USE %ERRVF
```

```
        S $ZT="S $ZT="" G WARN"" U $P:(NOCENA:CTRAP="""") G STAC"
        ZSHOW "*"
        KILL %ERRVF,%ERRIO,%ERRZA,%ERRZB,%ERRZE
        GOTO LOOPU
LOOPF   WRITE !,"LOCAL VARIABLES FOR ZLEVEL: ",$ZLEVEL,!
        ZWRITE
LOOPU   IF $ZLEVEL>1 ZGOTO $ZLEVEL-1:LOOPF
        WRITE !
STAC    SET $ZTRAP="GOTO WARN"
        WRITE !,"PROGRAM STACK: ",!
        WRITE !,"$STACK: ",$STACK,!
        WRITE !,"$STACK(-1): ",$STACK(-1),!
        FOR %ERRVI=$STACK(-1):-1:1 DO
        . WRITE !,"LEVEL: ",%ERRVI
        . SET %ERRVK=10
        . FOR %ERRVJ="PLACE","MCODE","ECODE" DO
        .. W ?%ERRVK,"",%ERRVJ,":",$STACK(%ERRVI,%ERRVJ)
        .. SET %ERRVK=%ERRVK+20
        CLOSE $IO
WARN    SET $ZTRAP="GOTO FATAL"
        IF $P=$I SET %ERRIO=$IO,%ERRZA=$ZA,%ERRZB=$ZB,%ERRZE=$ZEOF
        USE $P:(NOCENABLE:CTRAP="":EXCEPTION="")
        WRITE !,"YOU HAVE ENCOUNTERED AN ERROR"
        WRITE !,"PLEASE NOTIFY JOAN Q SUPPORT PERSON",!
FATAL   SET $ZTRAP=""
        ZM +$P($ST($ST(-1),"ECODE"),"Z",2)
```

The routine sets $ZTRAP to a sequence of values so that, in the event of an error various fallback actions are taken. If a STACKCRIT error occurs, GT.M makes a small amount of space for error handling. However, if the error handler uses up significant amounts of space by nesting routines or manupulating local variables, the error handler may cause another STACKCRIT error. In this case, it is possible for the error handling to loop endlessly, therefore this routine changes $ZTRAP so that each error moves the routine closer to completion.

First it attempts to store the context information in the global ^ERR. The LOOPV-LOOPT code records the invocation levels using the ZSHOW command. This technique addresses the situation where the application program defines or NEWs local variables for each level. The code executes a pass through the loop for each instance where the value of $ZLEVEL is greater than one (1). For each pass, ERR.M decrements the value of $ZLEVEL with the ZGOTO. When the value of $ZLEVEL reaches one (1), the code at, and following, the STACK label stores the error context available in the $STACK() function.

If there is a problem with storing any of this information, ^ERR attempts to store the context information in a file in the current default working directory. If it uses a file, in order to (at the label OPEN), record information about I/O operations, on the current device at the time of the error, the error handler SETs local variables to the values of the device specific I/O special variables $IO, $ZA, $ZB and $ZEOF before opening the log file.

The routine OPENs the log file with a name made up of the date and $JOB of the process. The NEWVERSION deviceparameter instructs GT.M to create a new version of the file. The LOOPF-LOOPU code records the invocation levels using the ZWRITE command in a manner analogous to that described above. If an error occurs trying to write to the file, $ZTRAP USEs the principal device and transfers control to the STAC label in an attempt to provide a minimal error context on the user terminal. The code at and following the STAC label records the error context available in the $STACK() function.

At the label WARN, the routine attempts to notify the user that an error has occurred and who to notify.

At the label FATAL, the ZMESSAGE command resignals the error. Because (with proper setup) $ETRAP and $ZTRAP are now null, GT.M releases control of the process to the host shell. In this example, the user never has access to Direct Mode.

Example:

```
GTM>zprint ^EP13
EP13    WRITE !,"THIS IS ",$TEXT(+0)
        SET $ZTRAP="GOTO NODB"
        KILL ^ERR
NODB    SET $ECODE="";this affects only $ETRAP
        ;S $ET="GOTO ^ERR";this implicitly stacks $ZTRAP
        N $ZT S $ZT="GOTO ^ERR" ;gives similar result
        DO SUB1
        WRITE !,"THIS IS THE END"
        QUIT
SUB1    WRITE !,"THIS IS SUB1"
        NEW
        SET (A,B,C)=$ZLEVEL
        DO SUB2
        QUIT
SUB2    WRITE !,"THIS IS SUB2"
        NEW
        SET (B,C,D)=$ZLEVEL
        DO SUB3
        QUIT
SUB3    WRITE !,"THIS IS SUB3"
        NEW
        SET (A,C,D)=$ZLEVEL
        DO BAD
BAD     NEW (A)
        SET B="BAD"
        WRITE 1/0
        WRITE !,"THIS IS NOT DISPLAYED"
        QUIT

GTM>do ^EP13
THIS IS EP13
THIS IS SUB1
THIS IS SUB2
THIS IS SUB3
YOU HAVE ENCOUNTERED AN ERROR
PLEASE NOTIFY JOAN Q SUPPORT PERSON
```

Example EP13 uses the error recording routine by setting $ZTRAP="GOTO ^ERR". When the routine encounters an error at label BAD, GT.M transfers control to routine ERR. Afterwards the ^ERR global would have contents like:

```
GTM>zwrite ^ERR
^ERR(13258,"64813,17382","$STACK")=0
^ERR(13258,"64813,17382","$STACK",-1)=5
^ERR(13258,"64813,17382","$STACK",1)="NODB+3^EP13|~||        DO SUB1|~||~|"
^ERR(13258,"64813,17382","$STACK",2)="SUB1+3^EP13|~||        DO SUB2|~||~|"
^ERR(13258,"64813,17382","$STACK",3)="SUB2+3^EP13|~||        DO SUB3|~||~|"
^ERR(13258,"64813,17382","$STACK",4)="SUB3+3^EP13|~||        DO BAD|~||~|"
^ERR(13258,"64813,17382","$STACK",5)="BAD+2^EP13|~||         WRITE 1/0|~|,M9,Z150373210,|~|"
^ERR(13258,"64813,17382","D",1)="/dev/pts/0 OPEN TERMINAL NOPAST NOESCA NOREADS TYPE WIDTH=165 LENG=48 "
^ERR(13258,"64813,17382","G",0)="GLD:*,REG:*,SET:77,KIL:3,GET:0,DTA:0,ORD:0,ZPR:0,QRY:0,LKS:0,LKF:0,CTN:0,DRD:3,DWT:0,NTW:77,NTR:5,N
R3:0,TTW:0,TTR:0,TRB:0,TBW:0,TBR:0,TR0:0,TR1:0,TR2:0,TR3:0,TR4:0,TC0:0,TC1:0,TC2:0,TC3:0,TC4:0,ZTR:0,DFL:0,DFS:0,JFL:0,JFS:0"
^ERR(13258,"64813,17382","G",0,1)=",JBB:0,JFB:0,JFW:0,JRL:0,JRP:0,JRE:0,JRI:0,JRO:0,JEX:0,DEX:0,CAT:80,CFE:0,CFS:0,CFT:0,CQS:0,CQT:0
^ERR(13258,"64813,17382","G",1)="GLD:/home/gtc_twinata/staff/nitin/
a.gld,REG:DEFAULT,SET:79,KIL:4,GET:0,DTA:0,ORD:0,ZPR:0,QRY:0,LKS:0,LKF:0,CTN:79,DRD:3,DWT:0,NTW:79
```

```
,NTR:6,NBW:87,NBR:178,NR0:0,NR1:0,NR2:0,NR3:0,TTW:0,TTR:0,TRB:0,TBW:0,TBR:0,TR0:0,TR1:0,TR2:0,TR3:0,TR4:0,TC0:0,TC1:0,TC2:0,"
^ERR(13258,"64813,17382","G",1,1)="TC3:0,TC4:0,ZTR:0,DFL:0,DFS:0,JFL:0,JFS:0,JBB:0,JFB:0,JFW:0,JRL:0,JRP:0,JRE:0,JRI:0,JRO:0,JEX:0,D
:0,CQT:0,CYS:0,CYT:0,BTD:6,WFR:0,BUS:0,BTS:0,STG:0,KTG:0,ZTG:0,DEXA:0,GLB:0,JNL:0,MLK:0,PRC:0,TRX:0,ZAD:0,JOPA:0,AFRA:0,BREA:0,MLBA:
^ERR(13258,"64813,17382","I",1)="$DEVICE="""""
^ERR(13258,"64813,17382","I",2)="$ECODE="",M9,Z150373210,"""
^ERR(13258,"64813,17382","I",3)="$ESTACK=5"
^ERR(13258,"64813,17382","I",4)="$ETRAP="""""
^ERR(13258,"64813,17382","I",5)="$HOROLOG=""64813,17382"""
^ERR(13258,"64813,17382","I",6)="$IO=""/dev/pts/0"""
^ERR(13258,"64813,17382","I",7)="$JOB=13258"
^ERR(13258,"64813,17382","I",8)="$KEY="""""
^ERR(13258,"64813,17382","I",9)="$PRINCIPAL=""/dev/pts/0"""
^ERR(13258,"64813,17382","I",10)="$QUIT=0"
^ERR(13258,"64813,17382","I",11)="$REFERENCE=""^ERR(13258,""""64813,17382"""",""""I"""",10)"""
^ERR(13258,"64813,17382","I",12)="$STACK=5"
^ERR(13258,"64813,17382","I",13)="$STORAGE=2147483647"
^ERR(13258,"64813,17382","I",14)="$SYSTEM=""47,gtm_sysid"""
^ERR(13258,"64813,17382","I",15)="$TEST=1"
^ERR(13258,"64813,17382","I",16)="$TLEVEL=0"
^ERR(13258,"64813,17382","I",17)="$TRESTART=0"
^ERR(13258,"64813,17382","I",18)="$X=12"
^ERR(13258,"64813,17382","I",19)="$Y=6"
^ERR(13258,"64813,17382","I",20)="$ZA=0"
^ERR(13258,"64813,17382","I",21)="$ZALLOCSTOR=780808"
^ERR(197306,"64341,39400","I",21)="$ZAUDIT=0"
^ERR(13258,"64813,17382","I",22)="$ZB="""""
^ERR(13258,"64813,17382","I",23)="$ZCHSET=""M"""
^ERR(13258,"64813,17382","I",24)="$ZCLOSE=0"
^ERR(13258,"64813,17382","I",25)="$ZCMDLINE="""""
^ERR(13258,"64813,17382","I",26)="$ZCOMPILE="""""
^ERR(13258,"64813,17382","I",27)="$ZCSTATUS=0"
^ERR(13258,"64813,17382","I",28)="$ZDATEFORM=0"
^ERR(13258,"64813,17382","I",29)="$ZDIRECTORY=""/path/to/the/current/directory"""
^ERR(13258,"64813,17382","I",30)="$ZEDITOR=0"
^ERR(13258,"64813,17382","I",31)="$ZEOF=0"
^ERR(13258,"64813,17382","I",32)="$ZERROR=""Unprocessed $ZERROR, see $ZSTATUS"""
^ERR(13258,"64813,17382","I",33)="$ZGBLDIR=""/path/to/the/global/directory"""
^ERR(13258,"64813,17382","I",34)="$ZHOROLOG=""64813,17382,175283,14400"""
^ERR(13258,"64813,17382","I",35)="$ZICUVER="""
^ERR(13258,"64813,17382","I",36)="$ZININTERRUPT=0"
^ERR(13258,"64813,17382","I",37)="$ZINTERRUPT=""IF $ZJOBEXAM()"""
^ERR(13258,"64813,17382","I",38)="$ZIO=""/dev/pts/0"""
^ERR(13258,"64813,17382","I",39)="$ZJOB=0"
^ERR(13258,"64813,17382","I",40)="$ZKEY="""""
^ERR(13258,"64813,17382","I",41)="$ZLEVEL=6"
^ERR(13258,"64813,17382","I",42)="$ZMAXTPTIME=0"
^ERR(13258,"64813,17382","I",43)="$ZMODE=""INTERACTIVE"""
^ERR(13258,"64813,17382","I",44)="$ZONLNRLBK=0"
^ERR(13258,"64813,17382","I",45)="$ZPATNUMERIC=""M"""
^ERR(13258,"64813,17382","I",46)="$ZPIN=""/dev/pts/0"""
^ERR(13258,"64813,17382","I",47)="$ZPOSITION=""RECORD+1^ERR"""
^ERR(13258,"64813,17382","I",48)="$ZPOUT=""/dev/pts/0"""
^ERR(13258,"64813,17382","I",49)="$ZPROMPT=""GTM>"""
^ERR(13258,"64813,17382","I",50)="$ZQUIT=0"
^ERR(13258,"64813,17382","I",51)="$ZREALSTOR=802648"
^ERR(13258,"64813,17382","I",52)="$ZRELDATE=""20180614 00:33"""
^ERR(13258,"64813,17382","I",53)="$ZROUTINES="". /usr/lib/fis-gtm/V6.3-007_x86_64 /usr/lib/fis-gtm/V6.3-007_x86_64/plugin/o(/
usr/lib/fis-gtm/V6.3-007_x86_64/plugin/r)"""
^ERR(13258,"64813,17382","I",54)="$ZSOURCE="""""
^ERR(13258,"64813,17382","I",55)="$ZSTATUS=""150373210,BAD+2^EP13,%GTM-E-DIVZERO, Attempt to divide by zero"""
^ERR(13258,"64813,17382","I",56)="$ZSTEP=""B"""
```

```
^ERR(13258,"64813,17382","I",57)="$ZSTRPLLIM=0"
^ERR(13258,"64813,17382","I",58)="$ZSYSTEM=0"
^ERR(13258,"64813,17382","I",59)="$ZTIMEOUT=-1"
^ERR(13258,"64813,17382","I",60)="$ZTDATA=0"
^ERR(13258,"64813,17382","I",61)="$ZTDELIM="""""
^ERR(13258,"64813,17382","I",62)="$ZTEXIT="""""
^ERR(13258,"64813,17382","I",63)="$ZTLEVEL=0"
^ERR(13258,"64813,17382","I",64)="$ZTNAME="""""
^ERR(13258,"64813,17382","I",65)="$ZTOLDVAL="""""
^ERR(13258,"64813,17382","I",66)="$ZTRAP=""GOTO OPEN"""
^ERR(13258,"64813,17382","I",67)="$ZTRIGGEROP="""""
^ERR(13258,"64813,17382","I",68)="$ZTSLATE="""""
^ERR(13258,"64813,17382","I",69)="$ZTUPDATE="""""
^ERR(13258,"64813,17382","I",70)="$ZTVALUE="""""
^ERR(13258,"64813,17382","I",71)="$ZTWORMHOLE="""""
^ERR(13258,"64813,17382","I",72)="$ZUSEDSTOR=759855"
^ERR(13258,"64813,17382","I",73)="$ZUT=1528966182176530"
^ERR(13258,"64813,17382","I",74)="$ZVERSION=""GT.M V6.3-007 Linux x86_64"""
^ERR(13258,"64813,17382","I",75)="$ZYERROR="""""
^ERR(13258,"64813,17382","L",0)="MLG:0,MLT:0"
^ERR(13258,"64813,17382","R",1)="RECORD+1^ERR:e99b16e4f7e1112d058dc22cb53491fd"
^ERR(13258,"64813,17382","R",2)="SUB3+3^EP13:d9e026c6d14e42567d3e64eecd049726"
^ERR(13258,"64813,17382","R",3)="SUB2+3^EP13:d9e026c6d14e42567d3e64eecd049726"
^ERR(13258,"64813,17382","R",4)="SUB1+3^EP13:d9e026c6d14e42567d3e64eecd049726"
^ERR(13258,"64813,17382","R",5)="NODB+3^EP13:d9e026c6d14e42567d3e64eecd049726"
^ERR(13258,"64813,17382","R",6)="+1^GTM$DMOD    (Direct mode) "
^ERR(13258,"64813,17382","V",1)="A=5 ;*"
^ERR(13258,"64813,17382","V",2)="B=""BAD"""
^ERR(13258,"64813,17382","VL",3,"V",1)="A=3"
^ERR(13258,"64813,17382","VL",3,"V",2)="B=3"
^ERR(13258,"64813,17382","VL",3,"V",3)="C=3"
^ERR(13258,"64813,17382","VL",4,"V",1)="B=4"
^ERR(13258,"64813,17382","VL",4,"V",2)="C=4"
^ERR(13258,"64813,17382","VL",4,"V",3)="D=4"
^ERR(13258,"64813,17382","VL",5,"V",1)="A=5"
^ERR(13258,"64813,17382","VL",5,"V",2)="C=5"
^ERR(13258,"64813,17382","VL",5,"V",3)="D=5"
```

Chapter 14. Triggers

Revision History		
Revision V7.1-004	27 June 2024	• In "Program to Record Information on an Error using $ZTRAP" (page 586), Added WFR,BUS,BTS,STG,KTG,ZTG,DEXA,GLB,JNL,MLK,PI to the ZWR output • In "Set" (page 601), Revert
Revision V7.1-002	19 September 2023	• In "Program to Record Information on an Error using $ZTRAP" (page 586), Add the new $ZICUVER ISV
Revision V6.3-007	04 February 2019	• In "Program to Record Information on an Error using $ZTRAP" (page 586), add $ZTIMEOUT and $ZAUDIT to $zjobexam() output
Revision V6.3-005	29 June 2018	• In "Program to Record Information on an Error using $ZTRAP" (page 586), update ^EP13 and ^ERR examples for V6.3-005.
Revision V6.3-004	23 March 2018	• In "Don'ts" (page 613), add a point recommending against injecting or manipulating $ZWRTAC lines while processing zwrite output. • In "Trigger Definition Storage" (page 601), correct the description of ^#t and specify that GT.M automatically sets the key size of a trigger to 1019 bytes.
Revision V6.3-003	12 December 2017	• In "ZGoto" (page 605), separate ZGOTO 1 and ZGOTO 0 and add more explanation to the latter
Revision V6.3-001	20 March 2017	• In "Chained and Nested Triggers" (page 598), corrected typo: ZTTRIGGEROP -> ZTRIGGEROP. • In "Program to Record Information on an Error using $ZTRAP" (page 586), Update output • In "Accessing Trigger Xecute Source Code" (page 605), specified that execution of trigger ensures that trigger code returned with $TEXT() or ZPRINT is current.
Revision V6.2-001	27 February 2015	Added a new section called "Accessing Trigger Xecute Source Code" (page 605).

Triggers

GT.M allows you to set up a trigger mechanism that automatically executes a defined action in response to a database update operation on a matching global node.The trigger mechanism executes a fragment of M code (trigger code) "before" or "as part

of" a database update. You can define the specifications of this mechanism in a Trigger Definition File. For a trigger on KILL (and ZKILL), GT.M executes trigger code "before" the KILL operation. For example, a trigger on KILL ^CIF(:,1) might clear old cross references. For a trigger on SET, GT.M executes trigger code "as part of" the SET operation. Within trigger logic, the ISV $ZTOLDVAL provides read access to the value of global node prior to the update and $ZTVALUE provides read/write access to the tentative SET value. This allows you to modify the tentative SET value before GT.M commits it to the database. The term "as part of" means that SET triggers execute intertwined with the SET operation. Although it is not yet committed the database, the tentative new value appears to the process as assigned but the process must SET $ZTVALUE to make any revision to the tentative value, because a SET of the global node would nest the trigger recursively - a pathological condition. GT.M executes SET triggers during a MERGE update where GT.M internally performs a series of SET operations and while performing a $INCREMENT() operation where GT.M internally performs a SET operation.For all triggers, GT.M handles the database update event and the triggered actions as an Atomic (all or nothing) transaction. Because triggers use application code and are always part of an implicit or explicit TP transaction, trigger code must conform to ACID conventions discussed in the TP documentation.

Triggers meet many application needs including (but not limited to) the following:

1. **Enforce schema-level consistency**: Since database schema created in a normal M application are implicit, M applications implement logic to maintain and enforce conformance with an application schema. Using triggers to enforce schema-level consistency ensures all processes invoke the code uniformly, and increases code modularity and maintainability.

2. **Allow an application to maintain one or more non-primary key indexes**. For example, a trigger on updates to global nodes containing a customer id can maintain an index on the last name.

3. **Implement business logic**: For example, an update to an account could automatically trigger updates to related accounts.

4. **Reducing replication traffic**: Since the GT.M replication stream carries only the triggering updates, not the triggered updates, triggers reduce network traffic.

5. **Automate application defined logging or journaling of updates or maintaining historical records**. Triggers can be used to control these.

6. **Implement referential integrity**: For example, a trigger can prevent the posting of a bank transaction for an inactive account and display a rule violation message.

7. **Debugging**: Debugging an application with multiple concurrent accesses is hard. You can use triggers to establish "watch points" on global variable updates to trap incorrect accesses. For example, if an application is failing because certain global variable nodes either have incorrect values or when previously set values disappear. A trigger can be used to trap all such accesses.

8. **Implement a dataflow based programming paradigm**. Although not a primary goal of the implementation of triggers, you can use them to implement applications that use a dataflow programming paradigm.

Trigger Definition File

A trigger definition file is a text file used for adding new triggers, modifying existing triggers, or removing obsolete triggers. A trigger definition file consists of one or more trigger definitions. A trigger definition includes the following information:

- **Trigger signature**: A trigger signature consists of global variable, subscripts, value, command, and trigger code. GT.M uses a combination of global variable, subscripts, value, and command to find the matching trigger to invoke for a database update.

  1. **Global Variable**: The name of a specific global to which this trigger applies.

2. **Subscripts**: Subscripts for global variable nodes of the named global, specified using the same patterns as the ZWRITE command.

3. **Value**: For commands that SET or update the value at a node, GT.M honors an optional pattern to screen for changes to delimited parts of the value. A value pattern includes a piece separator and a list of pieces of interest.

4. **Command**: There are four commands: SET, KILL, ZTRIGGER, and ZKILL (ZWITHDRAW is identical to ZKILL) the shorter name for the command is used when specifying triggers. MERGE is logically treated as equivalent to a series of SET operations performed in a loop. GT.M handles $INCREMENT() of a global matching a SET trigger definition as a triggering update.

5. **Trigger code**: A string containing M code that GT.M executes when application code updates, including deletions by KILL and like commands, a global node with a matching trigger. The specified code can invoke additional routines and subroutines.

> ## Note
>
> While GT.M does not restrict trigger code from performing I/O operations, FIS recommends against using OPEN, USE, READ, WRITE and CLOSE within trigger application code. Such operations may be useful for development and diagnostic purposes. However, triggers implicitly run as TP transactions and I/O violates the ACID property of Isolation. In addition, MUPIP has somewhat different I/O handling characteristics than the main GT.M run-time, so I/O within triggers run by MUPIP may behave differently than within the originating application environment.

- **ACID property modifiers for triggered database updates**: Currently, GT.M merely performs a syntax check on this part of a trigger definition. GT.M ensures the triggering database update, and any updates generated by trigger logic executed with transaction semantics. With the VIEW "NOISOLATION" command, GT.M transaction processing has long provided a mechanism for an application to inform the GT.M runtime system that it need not enforce Isolation. In such a case, the application and schema design provides Isolation by ensuring only one process ever updates nodes in a particular global at any given time, say by using $JOB as a subscript. This property anticipates a time when a trigger specification can provide NOISOLATION for particular nodes, in contrast to entire globals, and for every update to that node, in contrast to by process use of a VIEW command. Currently, the GT.M runtime system enforces Consistency for application logic inside a transaction and for triggered updates. This property anticipates a time when a trigger specification permits an application to inform the runtime system the application and schema design ensures appropriate Consistency for a trigger and its logic, thus relieving the GT.M runtime system from that task.

- **Trigger Name**: You can optionally specify a trigger name that uniquely identifies each trigger. GT.M uses a trigger name for error reporting and configuration management of triggers - for example, a ZSHOW "S" reports the name of each trigger on the stack. If you do not specify a trigger name, GT.M automatically generates one using the global name as a base. User-specified trigger names and automatically generated trigger names occupy different name space; both last for the life of the definition. A user-specified trigger name is an alphanumeric string of up to 28 characters. It must start with an alphabetic character or a percent sign (%). For a trigger name, GT.M uses the same naming convention as an M name. In other contexts, GT.M truncates M names at 31 characters. However, GT.M treats a trigger name of over 28 characters as an error. This is because a trigger name uniquely identifies a trigger and truncation may cause duplication.

An automatically generated trigger name is a string comprised of two parts. Using the global name as a base, GT.M takes the first part as an alphanumeric string of up to 21 characters starting with an alphabetic character or a percent sign (%). The trailing part consists of an automatically incremented number in the form of #n# where n is a whole number that monotonically increases from 1 to 999999 that uniquely identifies a trigger for the same update. For example, if no trigger names are specified in the trigger definition file, GT.M automatically generates trigger names Account#1#, Account#2#, and Account#3# for the

first three triggers defined for global variable ^Account. An attempt to use automatic assignment for more than a million triggers produces an error. Once the numeric portion of the auto generated names reaches 999999, you must reload all triggers associated with the global variables that use the auto generated name space. At run-time GT.M generates a trailing suffix of a hash-sign (#) followed by up to two characters to ensure that every trigger has a unique designation, even when the environment is complex. The run-time suffix applies to both user-specified and automatically generated trigger names. It helps in differentiating triggers with the same name in different database files.

Suppose you want to set up a trigger called **TrigAcct** on every **s ^Acct("ID")** to invoke the routine **^OpenAccount**. Your trigger definition file may have an entry like **+^Acct("ID") -command=S -xecute="do ^OpenAccount" -name=TrigAcct**. The following diagram identifies the different parts of this trigger definition:
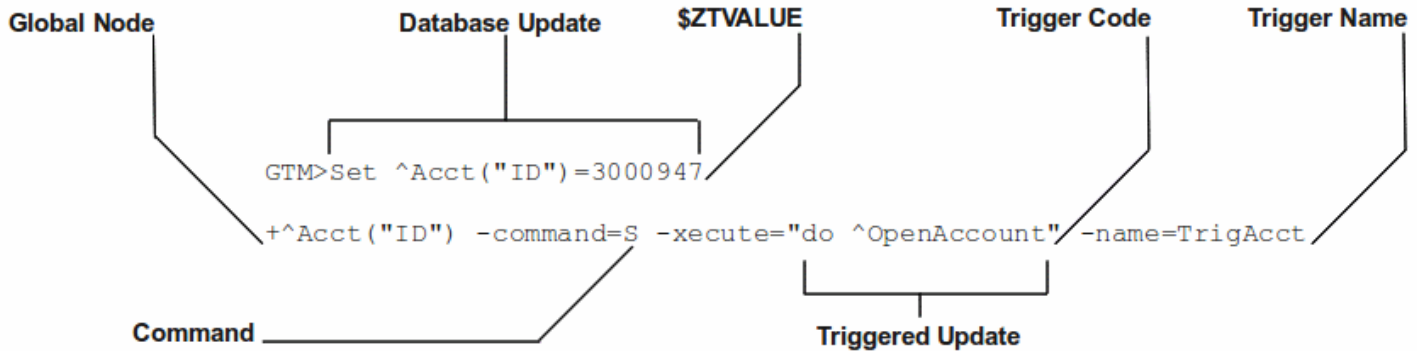
To apply this trigger definition file to GT.M, all you do is to load it using MUPIP TRIGGER -TRIGGERFILE or $ZTRIGGER(). GT.M would invoke trigger name **TrigAcct** on every SET operation on **^Acct("ID")**. Internally, GT.M stores trigger **TrigAcct** in the same database file where **^Acct** is stored. The syntax of an entry in a trigger definition file is:

{-triggername|-triggername-prefix*|-*|{+|-}trigvn -commands=*cmd[,...]* -xecute=*strlit1* [-[z]delim=*expr*][-pieces=[*lvn=]int1[:int2][;...]]* [-options={[no]i[solation]|[no]c[onsistencycheck]}...] [-name=*strlit2*]}

-triggername|-trigger-name-prefix*|-*	-triggername deletes a user-specified trigger name called **triggername** from the database.-**triggername*** deletes all those user-defined triggers whose starting name match **triggername**.-* deletes all triggers; if the MUPIP TRIGGER command does not specify -NOPROMPT , GT.M displays a warning and asks for user confirmation before deleting all triggers. If MUPIP TRIGGER command specifies -NOPROMPT and the definition file includes a -* line, GT.M deletes all the triggers without user confirmation. $ZTRIGGER() performs deletions -NOPROMPT.+**triggername** issues an error; to add a new user-specified trigger name, use -**name=strlit2**.
{+|-}tri gvn	**trigvn** is a global node on which you set up a trigger.-**trigvn** deletes any triggers in the database that match the specified trigger. +**trigvn** adds or replaces the specified trigger. If the specified trigger exists (with a matching specification), MUPIP TRIGGER or $ZTRIGGER() treats the matching definition as a no-op, resulting in no database update. If you want to specify more than one global node for the same trigger code, the following rules apply: 1. You can use patterns and ranges for subscripts. 2. You can specify a semicolon (;) separated list for subscripts. 3. You can specify a selection list that includes a mix of points, ranges and patterns, but a pattern cannot serve as either end of a range. For example, :,"a":"d";?1U is a valid specification but :,"a":?1A is not.

	4. You can specify a local variable name for each subscript. For example instead of **^X(1,:,:)**, you can specify **^X(1,lastname=:,firstname=:)**. This causes GT.M to define local variables lastname and firstname to the actual second and third level subscripts respectively from the global node invoking this trigger. The trigger code can then use these variables just like any other M local variable. As described in the Trigger Execution Environment section, trigger code executes in a clean environment - as if all code is preceded by an implicit NEW - the implicit assignments apply only within the scope of the trigger code and don't conflict or affect any run-time code or other triggers.
	5. You cannot use the **@** operator, unspecified subscripts (for example, **^A()** or **^A(:,))** or local or global variable names as subscripts.
	6. You cannot use patterns and ranges for the global variable name. Therefore, you cannot set a single trigger for **^Acct***.
	In order to account for any non-standard collation, GT.M evaluates string subscript ranges using the global specific collation when an application update first invokes a trigger - as a consequence, it detects and reports range issues at run-time rather than from MUPIP TRIGGER or $ZTRIGGER(), so test appropriately. For example, GT.M reports a run-time error for an inverted subscript range such as (ASCII) C:A.
-command=*cmd*	*cmd* is the trigger invocation command. Currently, you can specify one or more of S[ET], K[ILL], ZTR[IGGER], or ZK[ILL]. A subsequent GT.M release may support ZTK[ILL] for triggering on descendent nodes of a KILLed ancestor, but, while current versions accept ZTK, they convert it into K. If cmd specifies multiple command values, GT.M treats each M command as a separate trigger. Note that even if you specify both SET and KILL, only one M command matches at any given time. Trigger code is not executed in the following conditions: • A KILL of a node that does not exist. • A KILL of a node that has a cmd=ZK trigger, but no cmd=K trigger. • A ZKILL or ZWITHDRAW of a node that has descendents but no data and a trigger with cmd=**ZK**. • The trigger uses the "piece" syntax (described below) and no triggering piece changes in the update.
-xecute="|<<*strlit1*"|>>	strlit1 specifies the trigger code that is executed when an update matches trigvn. If strlit1 is a single line, enclose it with quotes (") and make sure that the quotes inside strlit1 are doubled as in normal M syntax. If strlit1 is in multiple lines, mark the beginning with << which must immediately follow the = after the -xecute. A newline must immediately follow the <<. >> should mark the end of multiple-line strlit1 and must be at the beginning of a line. The lines in strlit1 follow the standard conventions of a GT.M program, that is, optional label, line start, and M code. The maximum length of strlit1 (even if multi-line) is 1048576 (ASCII) characters or 4096 DB records, whichever is smaller. To validate strlit1, MUPIP TRIGGER or $ZTRIGGER() compiles it before applying the trigger definition to the database and issues a TRGCOMPFAIL error if it contains any invalid code.

> **Note**
>
> Trigger compilation detects compilation errors, but not run-time errors. Therefore, you should always test your trigger code before applying trigger definitions to the database.

> **Warning**
>
> As stated in the Trigger Definition File section, the text of trigger code is a part of the trigger signature. If you use two trigger signatures that have the same semantics (global variable, subscript, value, and command) but different text (for example: **set foo=$ztoldval**, **s foo= $ztoldval**, and **set foo=$ztol**), their signatures become different and GT.M treats them as different triggers. FIS recommends you to use comprehensive and strong coding conventions for trigger code or rely on user-specified names in managing the deletion and replacement of triggers.

Example:

```
+^multi –commands=set –name=example –xecute=<<
 do ^test1
 do stop^test2
>>
```

[-pieces=*int1[:int2][;...]*]	If **cmd** is **S[et]**, you can specify an optional piece list sequence where **int2**>**int1** and **int1:int2** denotes a integer range from **int1** to **int2**. The trigger gets executed only when any piece from the specified piece list changes. Suppose your trigvn has a list "**Window|Chair|Table|Door**" and you want to execute the trigger only when the value of the 3rd or 4th piece changes so you might specify the following trigger definition: ```+^trigvn –commands=S –pieces=3;4 –delim="|" –options=NOI,NOC –xecute="W ""3rd or` ` 4th element updated."""` `GTM>W ^trigvnWindow|Chair|Table|Door|` `GTM>s $Piece(^trigvn,"|",3)="Dining Table"` `3rd or 4th element updated.``` This trigger is not executed if you change the first element. For example: **S $Piece(^trigvn,"|",1)="Chandelier"** does not invoke the trigger. You can also specify a range for your piece sequence. For example, 3:5;7;9:11 specifies a trigger on pieces 3 through 5,7 and 9 through 11. GT.M merges any overlapping values or ranges - for example, 3:6;7 is the same as 3:7.
[-[z]delim=expr]	If **cmd** is **S[ET]**, you can specify an optional piece delimiter using **-[z]delim=expr** where expr is a string literal or an expression (with very limited syntax) evaluating to a string separating the pieces (e.g., "|") in the values of nodes, and is interpreted as an ASCII or UTF-8 string based on the environment variable gtm_chset. To allow for unprintable delimiters in the delimiter expression, MUPIP TRIGGER only accepts $CHAR() and $ZCHAR() and string concatenation (_) as embellishments to the string literals. If zdelim specifies a delimiter, GT.M uses the equivalent of $ZPIECE() to match pieces and to identify changes in $ZTUPDATE() (refer to the ISV description for additional information); otherwise, if delim specifies a delimiter, GT.M uses the equivalent of $PIECE() for the

	current mode (M or UTF-8). Specifying a delimiter for **cmd** other than S[ET] or specifying both **delim** and **zdelim** for the same trigger each produce an error.
[-options= *{no]i[solation]| [[no]c[onsistencycheck]}...*	You can specify **[NO] ISOLATION** or **[NO]CONSISTENCYCHECK** as a property of the triggered database updates. **NOISOLATION** is a facility for your application to instruct GT.M where the application logic and database schema take responsibility for ensuring the ACID property of **ISOLATION**, and that any apparent collisions are purely coincidental from multiple global nodes resident in the same physical block which serves as the GT.M level of granularity in conflict checking. In the current release this trigger designation is notational only - you must still implement **NOISOLATION** at the process level with the VIEW command, but you can use the trigger designation in planning to move to schema-based control of this facility. **NOCONSISTENCYCHECK** is a facility for your application to instruct GT.M that application logic and schema take responsibility for ensuring the ACID property of **CONSISTENCY**. The **[NO]CONSISTENCYCHECK** feature is not yet implemented and will be made available in a future GT.M release. For now, you can plan to move **CONSISTENCY** responsibility from your application to a trigger and implement it later when this feature becomes available.Note: **-options** are not part of the trigger signature and so can be modified without deleting an existing trigger.
[-name =*strlit2*]	*strlit2* is a user-specified trigger name. It is an alphanumeric string of up to 28 characters. It must start with an alphabetic character or a percent sign (%). Note: **-name** is not part of the trigger signature and so can be modified without deleting an existing trigger. Note also that the name can be used to delete a trigger - this alternative avoids potential issues with text variations in the code associated with the **-xecute** qualifier which is part of the trigger signature when the variations do not have semantic significance.

Trigger ISVs Summary

The following table briefly describes all ISVs (Intrinsic Special Variables) available for use by application logic using triggers. With the exception of $ZTWORMHOLE they return zero (0) if they have numeric values or an empty string when referenced by code outside of a trigger context. For more comprehensive description and usage examples of these ISVs, refer to "Triggers ISVs" (page 333).

$ZTNAME	Within a trigger context, $ZTNAME returns the trigger name. Outside a trigger context, $ZTNAME returns an empty string.
$ZTDATA	A fast path alternative to $DATA(@$REFERENCE)#2 for a SET or $DATA(@$REFERENCE) of the node for a KILL update.
$ZTDELIM	Within a SET trigger context, $ZTDE[LIM] returns the piece separator, as specified by -delim in the trigger definition. This allows triggers to extract updated pieces defined in $ZTUPDATE without having the piece separator hard coded into the routine. Outside of a SET trigger context, $ZTDELIM is null.
$ZTLEVEL	Returns the current level of trigger nesting (invocation by an update in trigger code of an additional trigger).
$ZTOLDVAL	Returns the prior (old) value of the node whose update caused the trigger invocation or an empty string if node had no value; refer to $ZTDATA to determine if the node had a data value.
$ZTRIGGEROP	For SET (including MERGE and $INCREMENT() operations), $ZTRIGGEROP returns the value "S". For KILL, $ZTRIGGEROP returns the value "K". For ZKILL or ZWITHDRAW, $ZTRIGGEROP returns the value "ZK". For ZTR, $ZTRIGGEROP returns the value "ZTR"

$ZTSLATE	$ZTSLATE allows you to specify a string that you want to make available in chained or nested triggers invoked for an outermost transaction (when a TSTART takes $TLEVEL from 0 to 1).
$ZTVALUE	For SET, $ZTVALUE has the value assigned to the node which triggered the update. Initially this is the value specified by the explicit (triggering) SET operation. Modifying $ZTVALUE within a trigger modifies the value GT.M eventually assigns to the node.
$ZTUPDATE	For SET commands where the GT.M trigger specifies a piece separator, $ZTUPDATE provides a comma separated list of ordinal piece numbers of pieces that differ between the current values of $ZTOLDVAL and $ZTVALUE.
$ZTWORMHOLE	$ZTWORMHOLE allows you to specify a string up to 128KB that you want to make available during trigger execution. You can use $ZTWORMHOLE to supply application context or process context to your trigger logic. Because $ZTWORMHOLE is retained throughout the duration of the process, you can read/write $ZTWORMHOLE both from inside and outside a trigger. Note that if trigger code does not reference $ZTWORMHOLE, GT.M does not make it available to MUPIP (via the journal files or replication stream). Therefore, if a replicating secondary has different trigger code than the initiating primary (an unusual configuration) and the triggers on the replicating node require information from $ZTWORMHOLE, the triggers on the initiating node must reference $ZTWORMHOLE to ensure GT.M maintains the data it contains for use by the update process on the replicating node. GT.M allows you to change $ZTWORMHOLE within trigger code so that a triggered update can trigger other updates but because of the arbitrary ordering of triggers matching the same node (refer to the discussion on trigger chaining below), such an approach requires careful design and implementation.

The Trigger Execution Environment section describes the interactions of the following ISVs with triggers: $ETRAP, $REFERENCE, $TEST, $TLEVEL, and $ZTRAP.

Chained and Nested Triggers

Triggers are chained or nested when a database update sets off more than one trigger. A nested trigger is a ▶ trigger set off by another trigger. GT.M assigns a nesting level to each nested trigger to up to 127 levels. While nested triggers are always Atomic with their triggering update GT.M gives each nested trigger a new trigger context rather than a part of the triggering update.A chained trigger is an arbitrary sequence of matching triggers for the same database update. Consider the following trigger definition entries:

```
+^Acct("ID") –commands=Set –xecute="Set ^Acct(1)=$ZTVALUE+1"
+^Acct(sub=:) –command=Set –xecute="Set ^X($ZTVALUE)=sub"
```

▼

This example sets off a chained sequence of two triggers and one nested trigger. On **Set ^Acct("ID")=10**, GT.M chains together an arbitrary sequence of triggers for ^**Acct("ID")** and ^**Acct(sub:)**. It is possible for either the ^**Acct(sub=:)** trigger or the ^**Acct("ID")** trigger to execute first and the other to follow because the trigger execution sequence is arbitrary. Whenever GT.M invokes the trigger for ^**Acct("ID")**, the **Set ^Acct(1)=$ZTVALUE+1** code sets off the trigger for ^**Acct(sub=:)** as a nested trigger.

Caution

FIS recommends against using chained and nested triggers that potentially update the same piece of a global variable. You should always assess the significance of having chained triggers for a database update especially because of the arbitrary trigger execution order.The following table shows the stacking behavior of some Intrinsic Special Variables in chained and nested triggers.

ISV	Chained Triggers	Nested Triggers
$REFERENCE	Shared	Stacked
$TEST	Stacked	Stacked
$ZTVALUE	Shared (updatable)	Stacked
$ZTOLDVAL	Shared	Stacked
$ZTDATA	Shared	Stacked
$ZTSLATE	Not Stacked	Not Stacked
$ZTRIGGEROP	Shared	Stacked
$ZTWORMHOLE	Not Stacked	Not Stacked
$ZTLEVEL	Shared	Stacked
$ZTUPDATE	depends on $ZTVALUE when trigger starts	Stacked

Stacked denotes an ISV whose value is restored at the completion of the trigger.

Not Stacked denotes an ISV whole value is retained after the completion of the trigger.

Shared denotes an ISV whose value is the same, possibly subject to updates, across chained updates

Note that a trigger that is both nested and chained has the characteristics from both columns - the "Chained" column is really about the relationship between triggers invoked by the same update and the "Nested" is really about the isolation of a trigger from the context that invoked it, whether or not that context is inside the context of another trigger.

A Simple Example

This section contains a simple example showing how a GT.M trigger can automatically maintain cross references in response to a SET or KILL operation on ^**CIF**(ACN,**1**). It also reinforces the basic trigger concepts explained above. Global nodes in ^**CIF**(ACN,**1**) have a structure ^**CIF**(ACN,**1**)=NAM|XNAME| where the vertical-bars are delimiters and XNAME is a customer's canonical name (e.g., **"Doe, Johnny"**). The application schema has one cross reference index, ^**XALPHA("A"**,XNAME,ACN)=**""**. A GT.M trigger specified for ^**CIF(:,1)** nodes can automatically maintain the cross references.

1. Using your editor, create a trigger definition file called **triggers.trg** with the following entry:

    +^**CIF(acn=:,1) -delim="|" -pieces=2 -commands=SET,KILL -xecute="Do ^XNAMEinCIF"**

    In this definition:

    - ^**CIF** - specifies the global variable to which the trigger applies.

    - **acn=:** - in ZWRITE syntax, ":" specifies any value for the first subscript.

    - **acn= prefix** requests GT.M assign the value of the first subscript (ACN) to the local variable **acn** before invoking the trigger logic.

    - **1** - specifies that the trigger matches only if the second subscript is 1 (one).

- **-delim**="|" - specifies that GT.M use "|" as the piece separator when checking the value of the node to see whether to invoke the trigger. The use of the keyword delim tells GT.M to use $PIECE() semantics for the value at the node; zdelim, instead, would instruct GT.M to use $ZPIECE() semantics.

- **-pieces**=2 - specifies that GT.M should only invoke the trigger when the update changes the second piece (XNAME) not for a change to the first piece (NAM), or any other piece without a change to XNAME.

- **-commands**=SET,KILL - specifies that GT.M invoke the trigger for SET and KILL updates (but not a ZKILL/ ZWITHDRAW command).

- **-xecute="Do ^XNAMEinCIF"** - provides code for GT.M to invoke to perform the trigger logic.

2. Execute a command like the following:

```
$ mupip trigger -triggerfile=triggers.trg
```

This command adds a trigger for ^CIF(:,1). On successful trigger load, this command displays an output like the following:

```
File triggers.trg, Line 1: ^CIF trigger added with index 1
=========================================
1 triggers added
0 triggers deleted
0 trigger file entries not changed
0 triggers modified
=========================================
```

3. Now, every SET and KILL operation on the global node ^CIF(:,1) executes the routine **XNAMEinCIF**.

4. Using your editor, create an M routine called XNAMEinCIF.m with the following code:

```
XNAMEinCIF ; Triggered Update for XNAME change in ^CIF(:,1)
    Set oldxname=$Piece($ZTOLDval,"|",2) Set:'$Length(oldxname) oldxname=$zchar(254); old XNAME
    Kill ^XALPHA("A",oldxname,acn); remove any old xref
                        ; Create a new cross reference if the command is a Set
    Do:$ZTRIggerop="S"
    . Set xname=$Piece($ZTVALue,"|",2) Set:'$Length(xname) xname=$zchar(254)            ; new XNAME
    . Set^XALPHA("A",xname,acn)=""                                                        ;
 create new xref
     ;
```

When the XNAME piece of a ^CIF(:,1) node is SET to a new value or KILLed, after obtaining the values, an unconditional KILL command deletes the previous cross reference index, if it exists. The deletion can be unconditional, because if the node did not previously exist, then the KILL is a no-op. Then, only if a SET invoked the trigger (determined from the ISV $ZTRIGGEROP), the trigger invoked routine creates a new cross reference index node. Note that because GT.M implicitly creates a new context for the trigger logic we do not have to worry about out choice of names or explicitly NEW any variables.

After obtaining the values, an unconditional KILL command deletes the previous cross reference index, if it exists. Then, only if a SET invoked the trigger (determined from the ISV $ZTRIGGEROP), the trigger invoked routine creates a new cross reference index node. Note that because GT.M implicitly creates a new context for the trigger logic we do not have to worry about out choice of names or explicitly NEW any variables.

Trigger Definition Storage

GT.M stores trigger definitions and trigger lookup information as nodes of a global-like structure (**^#t**). GT.M stores these structures in each region where triggers are mapped. You can manage the trigger definitions with MUPIP TRIGGER and $ZTRIGGER() but you cannot directly access **^#t** (except with DSE, which FIS recommends against under normal circumstances).

Database key and record size do not constrain the global like **^#t** structure. GT.M automatically sets the key size of **^#t** nodes to 1019 bytes which allows GT.M to store triggers more compactly in cases where regions have a small record size limit. GT.M can automatically span **^#t** nodes as needed to accommodate records that exceed block size.

Trigger Invocation and Execution Semantics

GT.M stores Triggers for each global variable in the database file for that global variable. When a global directory maps a global variable to its database file, it also maps triggers for that global variable to the same database file. When an extended reference uses a different global directory to map a global variable to a database file, that global directory also maps triggers for that global variable to that same database file.

Although triggers for SET and KILL / ZKILL commands can be specified together, the command invoking a trigger is always unique. The ISV $ZTRIGGEROP provides the trigger code which matched the triggering command.

Whenever a command updates a global variable, the GT.M runtime system first determines whether there are any triggers for that global variable. If there are any triggers, it scans the signatures for subscripts and node values to identify matching triggers. If multiple triggers match, GT.M invokes them in an arbitrary order. Since a future version of GT.M, potentially multi-threaded, may well choose to execute multiple triggers in parallel, you should ensure that when a node has multiple triggers, they are coded so that correct application behavior does not rely on the order in which they execute.

When a process executes a KILL, ZKILL or SET command, the target is the global variable node specified by the command argument for modification. With SET and ZKILL, the target is a single node. In the case of KILL, the target may represent an entire sub-tree of nodes. GT.M only matches the trigger against the target node, and only invokes the trigger once for each KILL command. GT.M does not check nodes in sub-trees to see whether they have matching triggers.

Kill / ZKill

If KILL or ZKILL updates a global node matching a trigger definition, GT.M executes the trigger code when a database state change has been computed but *before* it has been applied in the process space or the database. This means that the node to be KILLed and descendants (if any) remain visible to the trigger code. Note that a KILL trigger ignores $ZTVALUE.

Set

If a SET updates a global node matching a trigger definition, GT.M executes the trigger code *after* the node has been updated in the process address space, but before it is applied to the database. When the trigger execution completes, the trigger logic commits the value of a node from the process address space only if $ZTVALUE is not set. if $ZTVALUE is set during trigger execution, the trigger logic commits the value of a node from the value of $ZTVALUE.

Consider the following example:

```
GTM>set c=$ztrigger("S")
;trigger name: A#1#  cycle: 1
+^A -commands=S -xecute="set ^B=200"
```

```
;trigger name: B#1#  cycle: 1
+^B -commands=S -xecute="set $ztval=$ztval+1 "
GTM>set ^A=100,^B=100
GTM>write ^A
100
GTM>write ^B
201
```

SET ^A=100 invokes trigger A#1. When the trigger execution begins, GT.M sets ^A to 100 in the process address space, but does not apply it to the database. Therefore, the trigger logic sees ^A as set to 100. Other process accessing the database, however, see the prior value of ^A. When the trigger execution completes, the trigger logic commits the value of a node from the process address space only if $ZTVALUE is not set. The trigger logic commits the value of a node from the $ZTVALUE only if $ZTVALUE is set during trigger execution. Because $ZTVALUE is not set in A#1, GT.M commits the value of ^A from the process address space to the database. Therefore, GT.M commits ^A=100 to the database.SET ^B=200 invokes trigger B#2. $ZTVALUE is set during trigger execution, therefore GT.M commits the value of $ZTVALUE to ^B at the end of trigger execution.

> **Note**
>
> Within trigger code, any SET operation on ^B recursively invokes trigger B#1. Therefore, always set $ZTVALUE to change the value node during trigger execution.GT.M executes the triggering update and all associated triggers within the same transaction, whether or not the original command is inside a transaction. This means that although the trigger logic sees the updated value of the node, it is not visible to other processes until the outermost transaction commits to the database. If there is a conflicting update by another process, GT.M RESTARTs the explicit or implicit transaction to resolve the conflict.

A trigger may need to update the node whose SET initiated the trigger. Situations where this may occur include:

- a log or journal entry may need to be stored in a different piece of the same node as the update, or

- the node being updated may need its data to be stored in a canonical form (such as all-caps, or with standardized punctuation, regardless of how it was actually entered), or have its value limited to a range.

In such cases, the trigger logic should make the changes to the ISV $ZTVALUE instead of the global node. At the end of the trigger invocation, GT.M applies the value in $ZTVALUE to the node. Before the first matching trigger executes, GT.M sets $ZTVALUE. Since a command inside one trigger's logic can invoke another nested trigger, if already in a trigger, GT.M stacks the value of $ZTVALUE for the prior update before modifying it for the nested trigger initiation.

GT.M treats a MERGE command as a series of SET commands performed in collation order of the data source. GT.M checks each global node updated by the MERGE for matching triggers. If GT.M finds one or more matches, it invokes all the matching trigger(s) before the next command or the next set argument to the same SET command.

GT.M treats the $INCREMENT() function as a SET command. Since the result of a $INCREMENT() operation must be numeric, if the trigger code modifies $ZTVALUE, at the end of the trigger, GT.M applies the value of +$ZTVALUE (that is, $ZTVALUE coerced to a number) to the target node.

Trigger Execution Environment

As noted above, if there are multiple matching triggers, the GT.M process makes a list of matching triggers and executes them in an arbitrary order with no guarantee of predictability.

For each matching trigger:

1. The GT.M process implicitly stacks the naked reference, $REFERENCE, $TEST, $ZTOLDVAL, $ZTDATA, $ZTRIGGEROP, $ZTUPDATE and NEWs all local variables. At the beginning of trigger code execution, $REFERENCE, $TEST and the naked indicator initially retain the values they had just prior to being stacked (in the case of KILL/ZKILL, to the reference of the KILL/ZKILL command, even though the trigger executes prior to the removal of any nodes). If an update directly initiates multiple (chained) triggers, all start with identical values of the naked reference, $REFERENCE, $TEST, $ZTDATA, $ZTLEVEL, $ZTOLDVAL, and $ZTRIGGEROP. This facilitates triggers that are independent of the order in which they run. Application logic inside triggers can use $REFERENCE, the read-only intrinsic special variables $ZTDATA, $ZTLEVEL, $ZTOLDVAL, $ZTRIGGEROP & $ZTUPDATE, and the read-write intrinsic special variables $ZTVALUE, and $ZTWORMHOLE.

2. GT.M executes the trigger code. Note that in the course of executing this GT.M trigger, if the same trigger matches again for the same or a different target, GT.M *reinvokes the trigger recursively*. In other words, the same trigger can be invoked more than once for the same command. Note that such a recursive invocation is probably a pathological condition that will eventually cause a STACKCRIT error. Triggers may nest up to 127 levels, after which an additional attempt to nest produces a MAXTRGRNEST error.

3. When the code completes, GT.M clears local variables, restores what was stacked, except $ZTVALUE (refer to the ISV definitions for comments on modifying $ZTVALUE) to the values they had at the start of the trigger, and if there is any remaining trigger matching the original update, adjusts $ZTUPDATE and executes that next action. $ZTVALUE always holds the current target value for the node for which the application update initially invoked the trigger(s). Note that because multiple triggers for the same node execute in an arbitrary order, having more than one trigger change $ZTVALUE requires careful design and implementation.

After executing all triggers, GT.M commits the operation initiating the trigger as well as the trigger updates and continues execution with the next command (or, in the case of multiple nodes being updated by the same command, with the next node). Note that if the operation initiating the trigger is itself within a transaction, other processes will not see the database state changes till the TCOMMIT of the outermost transaction.

To ensure trigger actions are Atomic with respect to the update that invokes them, GT.M always executes trigger logic and the triggering update within a transaction. If the triggering update is not within an application transaction, GT.M implicitly starts a restartable "Batch" transaction to wrap the original update and any triggers generated by the update. In other words, when 0=$TLEVEL GT.M behaves as if implicit TStart *:Transactionid="BATCH" and TCommit commands bracket the upddate and its triggers. Therefore, the trigger code and/or its error trap always operate inside a Transaction and can use the TRESTART command even if the main application code never uses TSTART. $ETRAP code for use in triggers may include TROLLBACK logic.

The deprecated ZTSTART/ZTCOMMIT transactions are not compatible with triggers. If a ZTSTART transaction is already active when an update to a global that has any trigger defined occurs, GT.M issues a runtime error. Likewise GT.M treats any attempt to issue a ZTSTART within a trigger context as an error.

Error Handling during Trigger Execution

GT.M uses the $ETRAP mechanism to handle errors during trigger execution. If an error occurs during a trigger, GT.M executes the M code in $ETRAP. If $ETRAP does not clear $ECODE, GT.M does not commit the database updates within the trigger and passes control to the environment of the trigger update. If the $ETRAP action or the logic it invokes clears $ECODE, GT.M can continue processing the trigger logic.

Consider the following trivial example:

```
^Acct(id=:,disc=:) -commands=Set -xecute="Set msg=""Trigger Failed"",$ETrap=""If $Increment(^count) Write msg,!"" Set
 $ZTVAlue=x/disc"
```

During trigger execution if disc (the second subscript of the triggering update) evaluates to zero, resulting in a DIVZERO (Attempt to divide by zero) error, GT.M displays the message "Trigger Failed". Since the $ETRAP does not clear $ECODE, after printing the message, GT.M leaves the trigger context and invokes the error handler outside the trigger, if any. In a DIVZERO case, the process neither assigns a new value to ^Acct(id,disc) nor commits the incremented value of ^count to the database.

An application process can use a broad range of corrective actions to handle run-time errors within triggers. However, these corrective actions may not be available during MUPIP replication. As described in the Trigger Environment section, GT.M replicates only the trigger definitions, but not the triggered updates, which are executed by triggers when a replicating instance replays them. If a trigger is invoked in a replicating instance, it means that trigger was successfully invoked on the originating instance. For normal application requirements, you should ensure that the trigger produces the same results on a correctly configured replicating instance. Therefore your $ETRAP code on MUPIP should deal with the following cases where:

1. The run-time $ETRAP code modified the trigger logic to achieve the desired result

2. The replicating configuration is different from the initiating configuration

3. The filters between the initiating and replicating instance introduce an error

In the later two cases there are probably basically two possibilities for the mismatch environments - they are:

1. Intended and the $ETRAP mechanism is an integral part of managing the difference

2. Unintended and the $ETRAP mechanism should help notify the operational team to correct the difference and restart replication

The trigger facility includes an environment variable called gtm_trigger_etrap. It provides the initial value for $ETRAP in trigger context and can be used to set error traps for trigger operations in both mumps and MUPIP processes. The code can, of course, also SET $ETRAP within the trigger context. During a run-time trigger operation if you do not specify the value of gtm_trigger_etrap and a trigger fails, GT.M uses the current trap handler. In a mumps process, if the trap handler was $ZTRAP at the time of the triggering update and gtm_trigger_etrap isn't defined, the error trap is implicitly replaced by $ETRAP="" which exits out of both the trigger logic and the triggering action before the $ZTRAP unstacks and takes effect. In a MUPIP process, if you do not specify the value of gtm_trigger_etrap and a trigger fails, GT.M implicitly performs a SET $ETRAP="If $ZJOBEXAM()" and terminates the MUPIP process. $ZOBEXAM() records diagnostic information (equivalent to ZSHOW "*") to a file that provides a basis for analysis of the failure.

> **⚠ Important**
>
> $ZJOBEXAM() dumps the context of a process at the time the function executes and the output may well contain sensitive information such as identification numbers, credit card numbers, and so on. You should secure the location of files produced by the MUPIP error handler or set up appropriate security characteristics for operating MUPIP. Alternatively, if you do not want MUPIP to create a $ZJOBEXAM() file, explicitly set the gtm_trigger_etrap environment variable to a handler such as "Write !,$ZSTATUS,!, $ZPOSITION,! Halt".
>
> Other key aspects of error handling during trigger execution are as follows:

1. Any attempt to use the $ZTRAP error handling mechanism for triggers results in a NOZTRAPINTRIGR error.

2. If the trigger initiating update occurs outside any transaction ($TLEVEL=0), GT.M implicitly starts a transaction to wrap the initiating update and the triggered updates. Consequently if a TROLLBACK or TCOMMIT within the trigger context causes the code to come back to complete the initiating update with a different $TLEVEL than when the trigger started (including any implicit TSTART), GT.M issues a TRIGTCOMMIT error and does not commit the original update.

3. Any TCOMMIT that takes $TLEVEL below what it was when at trigger initiation, causes a TRIGTLVLCHNG error. This behavior applies to any trigger, whether chained, nested or singular.

4. It may appear that GT.M executes trigger code as an argument for an XECUTE. However, for performance reasons, GT.M internally converts trigger code into a pseudo routine and executes it as if it is a routine. Although this invisible for the most part, the trigger name can appear in places like error messages and $STACK() return values.

5. Triggers are associated with a region and a process can use one or more global directories to access multiple regions, therefore, there is a possibility for triggers to have name conflicts. To avoid a potential name conflict with other resources, GT.M attempts to add a two character suffix, delimited by a "#" character to the user-supplied or automatically generated trigger name. If this attempt to make the name unique fails, GT.M issues a TRIGNAMEUNIQ error.

6. Defining gtm_trigger_etrap to hold M code of any complexity exposes mismatches between the quoting conventions for M code and shell scripts. FIS suggests an approach of enclosing the entire value in single-quotes and only escaping the single-quote ('), exclamation-point (!) and back-slash (\) characters. For a comprehensive (but hopefully not very realistic) example:

```
$ export gtm_trigger_etrap='write:1\'=2 $zstatus,\!,"5\\2=",5\\2,\! halt'
$ echo $gtm_trigger_etrap
write:1'=2 $zstatus,!,"5\2=",5\2,! halt
GTM>set $etrap=$ztrnlnm("gtm_trigger_etrap")
GTM>xecute "write 1/0"
150373210,+1^GTM$DMOD,%GTM-E-DIVZERO, Attempt to divide by zero
5\2=2
$
```

ZGoto

To maintain the transactional integrity of triggers and to avoid branching control to an inappropriate destination, ZGOTO behaves as follows:

1. GT.M does not support ZGOTO 1:<entryref> arguments in MUPIP because they form an attempt to replace the MUPIP context.

2. When a ZGOTO argument specifies an entryref at or below the level of the update that initiated the trigger, GT.M redirects the flow of control to the entryref without performing the triggering update. Alternatively if GT.M finds a non-null $ECODE, indicating an unhandled error when it goes to complete the trigger, it throws control to the current error handler rather than committing the original triggering update.

3. ZGOTO 1 returns to the base stack frame, which has to be outside any trigger invocation.

4. ZGOTO 0 terminates the process; when ""=$ZTRAP and ""!=$ECODE, ZGOTO 0 returns a non-zero status, derived from the error code in $ZSTATUS, to the shell.

5. ZGOTO from within a run-time trigger context cannot directly reach a subsequent M command on the line containing the command that invoked the trigger, because a ZGOTO with an argument specifying the level where the update originated but no entryref returns to the update itself (as would a QUIT) and, if $ECODE is null, GT.M continues processing with any additional triggers and the triggering update before resuming the line.

Accessing Trigger Xecute Source Code

ZPRINT/$TEXT()/ZBREAK recognize both a runtime-disambiguator, delimited with a hash-sign (#), and a region-disambiguator, delimited by a slash(/). ZPRINT and ZBREAK treat a trigger-not-found case as a TRIGNAMENF error, while $TEXT() returns the empty string. When their argument contains a region-disambiguator, these features ignore a null runtime-

disambiguator. When their argument does not contain a region-disambiguator, these features act as if runtime-disambiguator is specified, even if it has an empty value. When an argument specifies both runtime-disambiguator and region-disambiguator and the runtime-disambiguator identifies a trigger loaded from a region different from the specified region, or the region-disambiguator identifies a region which holds a trigger that is not mapped by $ZGBLDIR, these features treat the trigger as not found.

ZPRINT or $TEXT() of trigger code may be out-of-date if the process previously loaded the code, but a $ZTRIGGER() or MUPIP TRIGGER has since changed the code. In other words, execution of a trigger (not $TEXT()) ensures that trigger code returned with $TEXT() is current.

GT.CM

GT.CM servers do not invoke triggers. This means that the client processes must restrict themselves to updates which don't require triggers, or explicitly call for the actions that triggers would otherwise perform. Because GT.CM bypasses triggers, it may provide a mechanism to bypass triggers for debugging or complex corrections to repair data placed in an inconsistent state by a bug in trigger logic.

Other Utilities

During MUPIP INTEG, REORG and BACKUP (including -BYTESTREAM), GT.M treats trigger definitions just as it treats any normal global node.

Because they are designed as state capture and [re]establishment facilities, MUPIP EXTRACT does **not** extract trigger definitions and MUPIP LOAD doesn't restore trigger definitions or invoke any triggers. While you can construct input for MUPIP LOAD which bypasses triggers, there is no way for M code itself to bypass an existing trigger, except by using a GT.CM configuration. The $ZTRIGGER() function permits M code to modify the triggers, add/delete/change, across all regions, excluding those served by GT.CM. However, those actions affect all processes updating the node associated with any trigger. Like MUPIP EXTRACT and LOAD, the ^%GI and ^%GO M utility programs do not extract and load GT.M trigger definitions. Unlike MUPIP LOAD, ^%GI invokes triggers just like any other M code, which may yield results other than those expected or intended.

Triggers in Journaling and Database Replication

GT.M handles "trigger definitions" and "triggered updates" differently.

1. Trigger definition changes appear in both journal files and replication streams so the definitions propagate to recovered and replicated databases.

2. Triggered updates appear in the journal file, since MUPIP JOURNAL RECOVER/ROLLBACK to not invoke triggers. However, they do not appear in the replication stream since the Update Process on a replicating instance apply triggers and process their logic.

Journaling

When journaling is ON, GT.M generates journal records for database updates performed by trigger logic. For an explicit database update, a journal record specifies whether any triggers were invoked as part of that update. GT.M triggers have no effect on the generation and use of before image journal records, and the backward phase of rollback / recovery.

A trigger associated with a global in a region that is journaled can perform updates in a region that is not journaled. However, if triggers in multiple regions update the same node in an unjournaled region concurrently, the replay order for recovery or

rollback might differ from that of the original update and therefore produce a different result; therefore this practice requires careful analysis and implementation. Except when using triggers for debugging, FIS recommends journaling any region that uses triggers.

The following sample journal extract shows how GT.M journals records updates to trigger definitions and information on $ZTWORMHOLE:

```
GDSJEX04
01\61731,15123\1\16422\gtm.node1\gtmuser1\21\0\\\
02\61731,15123\1\16422\0
01\61731,15126\1\16423\gtm.node1\gtmuser1\21\0\\\
08\61731,15126\1\16423\0\4294967297
05\61731,15126\1\16423\0\4294967297\1\4\^#t("trigvn","#LABEL")="1"
05\61731,15126\1\16423\0\4294967297\2\4\^#t("trigvn","#CYCLE")="1"
05\61731,15126\1\16423\0\4294967297\3\4\^#t("trigvn","#COUNT")="1"
05\61731,15126\1\16423\0\4294967297\4\4\^#t("trigvn",1,"TRIGNAME")="trigvn#1#
"05\61731,15126\1\16423\0\4294967297\5\4\^#t("trigvn",1,"CMD")="S"
05\61731,15126\1\16423\0\4294967297\6\4\^#t("trigvn",1,"XECUTE")="W $ZTWORMHOLE
s ^trigvn(1)=""Triggered Update"" if $ZTVALUE=1 s $ZTWORMHOLE=$ZTWORMHOLE_""
Code:CR"""
05\61731,15126\1\16423\0\4294967297\7\4\^#t("trigvn",1,"CHSET")="M"
05\61731,15126\1\16423\0\4294967297\8\4\^#t("#TRHASH",175233586,1)="trigvn"_$C(0,0,0,0,0)_
"W $ZTWORMHOLE s ^trigvn(1)=""Triggered Update"" if $ZTVALUE=1 s $ZTWORMHOLE=$ZTWORMHOLE
_"" Code:CR""1"
05\61731,15126\1\16423\0\4294967297\9\4\^#t("#TRHASH",107385314,1)="trigvn"_$C(0,0)_"
W $ZTWORMHOLE s ^trigvn(1)=""Triggered Update"" if $ZTVALUE=1 s $ZTWORMHOLE=$ZTWORMHOLE_""
Code:CR""1"
09\61731,15126\1\16423\0\4294967297\1\1\
02\61731,15127\2\16423\0
01\61731,15224\2\16429\gtm.node1\gtmuser1\21\0\\\
08\61731,15224\2\16429\0\8589934593
11\61731,15224\2\16429\0\8589934593\1\"A process context like--> Discount:10%;Country:IN"
05\61731,15224\2\16429\0\8589934593\1\1\^trigvn="Initial Update"
09\61731,15224\2\16429\0\8589934593\1\1\BA
08\61731,15232\3\16429\0\12884901889
11\61731,15232\3\16429\0\12884901889\1\"A process context like--> Discount:10%;Country:IN Code:CR"
05\61731,15232\3\16429\0\12884901889\1\1\^trigvn="1"
09\61731,15232\3\16429\0\12884901889\1\1\BA
08\61731,15260\4\16429\0\17179869185
11\61731,15260\4\16429\0\17179869185\1\"A process context like--> Discount:10%;Country:IN Code:CR"
05\61731,15260\4\16429\0\17179869185\1\1\^trigvn="Another Update"
09\61731,15260\4\16429\0\17179869185\1\1\BA
02\61731,15263\5\16429\0
01\61731,15865\5\26697\gtm.node1\gtmuser1\21\0\\\
08\61731,15865\5\26697\0\21474836481
05\61731,15865\5\26697\0\21474836481\1\2\^trigvn(1)="Updated outside the trigger."
09\61731,15865\5\26697\0\21474836481\1\1\BA
02\61731,15870\6\26697\0
01\61731,15886\6\26769\gtm.node1\gtmuser1\21\0\\\
08\61731,15886\6\26769\0\25769803777
11\61731,15886\6\26769\0\25769803777\1\" Code:CR"
05\61731,15886\6\26769\0\25769803777\1\1\^trigvn="1"
09\61731,15886\6\26769\0\25769803777\1\1\BA
02\61731,15895\7\26769\0
01\61731,15944\7\26940\gtm.node1\gtmuser1\21\0\\\
08\61731,15944\7\26940\0\30064771073
05\61731,15944\7\26940\0\30064771073\1\3\^trigvn="Another Update"
09\61731,15944\7\26940\0\30064771073\1\1\BA
08\61731,16141\8\26940\0\34359738369
11\61731,16141\8\26940\0\34359738369\1\"A process context like--> Discount:10%;Country:IN  Code:CR"
05\61731,16141\8\26940\0\34359738369\1\1\^trigvn="1"
```

```
09\61731,16141\8\26940\0\34359738369\1\1\BA
08\61731,16178\9\26940\0\38654705665
11\61731,16178\9\26940\0\38654705665\1\"A process context like--> Discount:10%;Country:IN  Code:CR"
05\61731,16178\9\26940\0\38654705665\1\1\^trigvn="Another update"
09\61731,16178\9\26940\0\38654705665\1\1\BA
02\61731,16210\10\26940\0
01\61731,16517\10\5337\gtm.node1\gtmuser1\21\0\\\
08\61731,16517\10\5337\0\42949672961
05\61731,16517\10\5337\0\42949672961\1\2\^trigvn(1)="4567"
09\61731,16517\10\5337\0\42949672961\1\1\BA
08\61731,16522\11\5337\0\47244640257
11\61731,16522\11\5337\0\47244640257\1\" Code:CR"
05\61731,16522\11\5337\0\47244640257\1\1\^trigvn="1"
09\61731,16522\11\5337\0\47244640257\1\1\BA
08\61731,16544\12\5337\0\51539607553
11\61731,16544\12\5337\0\51539607553\1\"No context Code:CR"
05\61731,16544\12\5337\0\51539607553\1\1\^trigvn="1"
09\61731,16544\12\5337\0\51539607553\1\1\BA
02\61731,16555\13\5337\0
03\61731,16555\13\5337\0\0
```

This journal extract output shows $ZTWORMHOLE information for each triggered update to ^**trigvn**. Notice how GT.M stored trigger definitions as a node of a global-like structure ^**#t** and how GT.M journals the trigger definition for ^**trigvn** and the triggered update for ^**trgvn.**

Note: GT.M implicitly wraps a trigger as an M transaction. Therefore, a journal extract file for a database that uses triggers has Type 8 and 9 (TSTART/TCOMMIT) records even if the triggers perform no updates (that is, are effectively No-ops).

MUPIP JOURNAL -RECOVER / -ROLLBACK

The lost and broken transaction files generated by MUPIP JOURNAL -RECOVER / -ROLLBACK contain trigger definition information. You can identify these entries + or - and appropriately deal with them using MUPIP TRIGGER and $ZTRIGGER().

Multisite Database Replication

During replication, GT.M replicates trigger definitions to ensure that when MUPIP TRIGGER updates triggers on an initiating instance, all replicating instances remain logically identical.

The replication stream has no records for updates generated by implicit GT.M trigger logic. If your trigger action invokes a routine, specify the value of the environment variable gtmroutines before invoking replication with MUPIP so the update process can locate any routines invoked as part of trigger actions.

To support upward compatibility, V5.4-000 allows your originating primary to replicate to:

1. An instance with a different a trigger configuration.

2. An instance running a prior GT.M version (having no trigger capability), in which case it replicates any triggered updates.

When a replicating instance needs to serve as a possible future originating instance, you must carefully design your replication filters to handle missing triggers or trigger mismatch situations to maintain logical consistency with the originating primary.

Replicating to an instance with a different trigger configuration

During an event such as rolling upgrade, the replicating instance may have a new database schema (due to application upgrades) and in turn a new set of triggers. Therefore, GT.M replication allows you to have different-trigger configuration for

originating (primary) and replicating (secondary) instances. When replication starts between the two instances, any update to triggers on the originating instance automatically flow (through the filters) to the replicating instance. For the duration of the rolling upgrade, your application must use replication filters to ensure trigger updates on the originating instance produce an appropriate action on the replicating instance. However, whenever you follow the practice of creating replicating instances from backups of other appropriate originating instances, you do not have to use additional replication filters, because the backups include GT.M trigger definitions, under normal conditions instances automatically have the same triggers.

Because the replication stream carries the native key format, having different collation for a replicated global on the replicating node from that on the initiating node is effectively a schema change and requires an appropriate filter to appropriately transform the subscripts from initiating form to replicating form. This is true even without triggers. However, with triggers a mismatch also potentially impacts appropriate trigger invocation.

Because GT.M stores triggers in the database files as pseudo global variables, an application upgrade requiring a change to triggers is, in the worst case, no different than an application upgrade that changes the database schema, and can be handled under current rolling upgrade methods. Some changes to GT.M triggers may well be much simpler than a database schema change, and may not need a rolling upgrade.

Replicating to an instance that does not support triggers

At replication connection, if an originating primary detects a replicating instance that does not support triggers, the Source Server issues a warning to the operator log and the Source Server log. The Source Server also sends a warning message to the operator log and the Source Server log the first time it has to replicate an update associated with a trigger. In this configuration, internal filters in GT.M strip the replication stream of trigger-related information such as $ZTWORMHOLE data and trigger definition updates from MUPIP TRIGGER or $ZTRIGGER(). The Source Server does send updates done within trigger logic. Unless the application has replication filters that appropriately compensate for the trigger mismatch, this is a situation for concern, as the replicating instance may not maintain logical consistency with the originating primary. Note that filters that deal with $ZTWORMHOLE issues must reside on the originating instance.

Update & Helper Processes

For any replication stream record indicating triggers were invoked, the Update Process scans for matching GT.M triggers and unconditionally executes the implicit GT.M trigger logic.

MUPIP Trigger and $ZTRIgger()

MUPIP TRIGGER provides a facility to examine and update triggers. The $ZTRIGGER() function performs trigger maintenance actions analogous to those performed by MUPIP TRIGGER. $ZTRIGGER() returns the truth value expression depending on the success of the specified action. You choice of MUPIP TRIGGER or $ZTRIGGER() for trigger maintenance should depend on your current application development model and configuration management practices. Both MUPIP TRIGGER and $ZTRIGGER() use the same trigger definition syntax. You should familiarize yourself with the syntax of an entry in a trigger definition file before exploring MUPIP TRIGGER and $ZTRIGGER(). For more information and usage examples of MUPIP TRIGGER, refer to **GT.M Administration and Operations Guide**. For more information and usage examples of $ZTRIGGER(), refer to "$ZTRIgger()" (page 286).

Appendix A. M Coding Standards - Do's and Don'ts

M Coding Standards - Do's and Don'ts

M coding standards help:

- Improve quality and maintainability by producing uniform standardized code.

- Aid in readability and comprehension of developed source code by providing an unambiguous, easy to read standard to follow.

- Avoid common errors that would not be picked up by a compiler by using these as a framework for good coding practices.

- Provide an objective reference point for the code authors, maintainers, and reviewers.

Do's

Character Set	Construct literal strings using only graphic ASCII characters . For non-graphic characters, use the $CHAR() or $ZCHAR() functions to build strings, use a resource file, or build them from comments in the source code using $TEXT().
	Use VIEW "BADCHAR" in production environments.
Internationalization	Use standard M functions - $ASCII(), $CHAR(), $EXTRACT(), $FIND(), $JUSTIFY(), $LENGTH(), $PIECE(), and $TRANSLATE() - for character-oriented operations. Use analogous byte-oriented functions - $ZASCII(), $ZCHAR(), $ZEXTRACT() / $ZSUBSTR(), $ZFIND(), $ZJUSTIFY(), $ZLENGTH(), $ZPIECE(), and $ZTRANSLATE() when the logic calls for byte, rather than character operations. Use $ZSUBSTR() to ensure the byte-oriented operation produces a valid character result and $ZEXTRACT() when the operation is strictly byte-oriented, with no character implications.
	Use $ZCONVERT() rather than $TRANSLATE() / $ZTRANSLATE() for case conversion.
Alias Variables and Containers	Use Alias Variables and Containers where they promote modularity and an appropriate object-like approach.
Parameter Passing	Use parameter passing to minimize scoping risk and to implicitly document [sub]routine interfaces. Place variables that are optionally passed at the end of the parameter list. Choose pass-by-reference variable names for clarity and to avoid side effects. Note that using the same name in the actuallist and the formallist may have implications.
Naked References	Avoid naked references except when they reduce the width of the line and materially improve readability. In such cases, ensure any naked references follows the full reference, on the same line with no intervening invocation of other code.
Entryrefs	Use the top entryref for invocation from the shell rather than from another routine. Where there is a risk of inadvertent execution from the shell of a routine intended only to be called from M code, protect the top entryref with code that uses $STACK to validate whether or not it is invoked from the shell. Existing utility routines with interfaces documented as not requiring a label are exempt from this requirement.

Lines with multiple commands	Use lines with multiple, related commands to improve readability as long as they are not too long. Avoid lines with multiple, unrelated commands.
Lines Width	Limit lines to no wider than 132 columns.

> **Note**
>
> The line oriented nature of M means that sometimes a single long line can be more readable than a block of code. Such occasional long lines are permissible where so justified.

Error Handling	All product code must have an error trap. Unless the intent of the code is to invoke Direct Mode, the error handler must never use a BREAK explicitly or implicitly to do so. The base error handler in a program suite, and possibly some other error handlers, must provide a way to appropriately preserve the context of unpredictable errors.
	New code should generally use $ETRAP error handling rather than $ZTRAP error handling. As changing error handling can be risky (for example, indirect references in databases), retain error handling in existing code unless you have tested and verified the safety of the change.
Transaction Processing	Except for tests, code transactions as restartable, avoid non-Isolated actions (BREAK, JOB, LOCK ZSYSTEM or I/O) within transactions; minimize transaction size, use TRANSACTION="BATCH" for better performance where the solution doesn't require strong Durability or provides Durability with application logic. When LOCKs are appropriate, place them outside the transaction.
BREAK	To prevent applications from inadvertently falling into direct mode, only use the BREAK command when there is a specific requirement for its use. When circumstances require BREAK commands more persistent than those placed with ZBREAK, or in places within lines, conditionalize them on a debug setting, for example BREAK: ($get(debug)&(<condition>)).
Argumentless Do	This language construct provides a way to code an embedded subroutine, which stacks $TEST - something extrinsic functions ($$) also do, but DO with an entryref argument does not. It also provides some relief from the line-oriented structure of the language, albeit at some cost.
	When the logic calls for multiple invocations of a subroutine avoid using multiple copies of the same argumentless DO body.
	Leave a space between the last level indicator (li) and the first command or a comment delimiter.
	Be careful with level indicators, as any reduction in number, even for a comment, terminates one or more levels.
GOTO and ZGOTO	Except for handling logic to effect an exit from nested logic and nested function calls or where appropriate in test code, avoid the use of GOTO and ZGOTO commands. When using these commands outside of test code, you must include a comment explaining why such use works better than any refactoring that would eliminate the [Z]GOTO.
HALT, QUIT, and HALT	Choose QUIT, rather than HALT, to terminate a routine, unless there is a clear requirement to the contrary.

Account for the difference in QUIT from a FOR, which does not change the stack level of the M virtual machine and QUIT from a [sub]routine which does change the stack level. This may require the use of a state flag to terminate a FOR that invokes a subroutine.

Choose the argumentless (which requires a following double space) or value form of QUIT as appropriate. When a subroutine can be invoked with either a DO or extrinsic ($$), explicitly code the alternative exits, rather than relying on a setting of $ZQUIT_ANYWAY.

QUIT at the end of a [sub]routine is required unless the [sub]routine ends in a HALT. QUIT at the end of an argumentless DO level is optional.

Use ZHALT to return an invocation to the shell that the termination is abnormal.

LOCK, ZALLOCATE, and ZDEALLOCATE	Always specify a timeout on LOCK or LOCK+ commands.
	Use a protocol for resource name order to minimize deadlocks. Use the standard incremental LOCK (+/-), rather than ZALLOCATE and ZDEALLOCATE.
NEW	Minimize use of argumentless and exclusive NEW except to satisfy requirements. These variants may be appropriate at the beginning of the base routine of an application.
READ	Always use a timeout, except when READing from a file in NOFOLLOW mode. Except when collecting raw or externally validated data, READ into a local variable and validate that the value is appropriate - check for length, range, delimiters and any value restrictions. Always validate input before using it in ways that assume it meets expectations; this is critical when using it in indirection or XECUTE, or storing it durably in a global variable.
SET	When setting several nodes to the same value, specify a list of names within parentheses, rather than separately, e.g., SET (A,B,C)=0 vs. SET A=0,B=0,C=0.
XECUTE, Indirection, $ZSYSTEM and PIPE device commands	To minimize run-time errors and prevent out-of-design user induced outcomes, ensure strings for use by the indirection operator, the XECUTE command, the ZSYSTEM command and PIPE device commands are valid, either by program design and implementation, or by validating the value prior to use.
Else	Use ELSE with care. Because GT.M stacks $TEST only at the execution of an extrinsic or an argumentless DO command, any XECUTE or DO with an argument has the potential side effect of altering $TEST.
Post conditionals	When conditionalizing a single command or transfer of control argument, and there is no need to set $TEST, use postconditionals, as they provide a slight performance advantage and tend to improve readability by tying the condition closely to the action. For example, choose SET:<condition> over IF <condition> SET.
$ZDATA() and $DATA()	Use $ZDATA() rather than $DATA() unless the logic needs to ignore alias implications.
$Increment	You may use an IF to discard the result of INCREMENT() in order to take advantage of the INCREMENT() side-effect.
$Next	Use $ORDER() rather than $NEXT(), which is deprecated.

$Piece()	If using a piece of data more than once, extract the data to a local variable for reuse, rather than using repeated invocations of $PIECE() to extract the same piece of data.
$Random()	$RANDOM(1) always returns 0 and so is never appropriate. Adjust $RANDOM() results with appropriate arithmetic to achieve the desired range.
$ZPREVIOUS()	Use the standard $ORDER(x,-1), rather than $ZPREVIOUS(x).
$ETrap vs. $ZTRAP	Use $ETRAP rather than $ZTRAP unless there are good reasons not to.
$KEY vs. $ZB	Use $KEY rather than $ZB, unless code must run on old versions of GT.M which do not maintain $KEY for the target device.
$STack vs. $ZLEVEL	Use $STACK rather than $ZLEVEL, unless it eliminated arithmetic or existing uses of $ZLEVEL are so numerous as to make a change high risk.
$ZCMDLINE	Validate all input from $ZCMDLINE as if it were from a READ for user input.
Extrinsic Functions and Special Variables ($$)	When a subroutine needs to return a single value or an an array of values, choose an Extrinsic invocation over a DO in order to minimize scoping risks.
Triggers	While GT.M does not restrict trigger code from performing I/O operations, avoid using OPEN, USE, READ, WRITE and CLOSE within trigger application code. Such operations may be useful for development and diagnostic purposes. Triggers implicitly run as TP transactions and I/O violates the ACID property of Isolation , as do JOB, LOCK, ZSYSTEM and external calls. ($&) .
	Use comprehensive and strong coding conventions for trigger code or rely on user-specified names in managing the deletion and replacement of triggers.
	Except when using triggers for debugging, use journaling any region that uses triggers.
Call-in/Call-outs	Use gtm_malloc/gtm_free in the external functions for enhanced performance and better debugging capability in case memory management problems occur with external calls.
	Use gtm *t types defined in gtmxc_types.h be used instead of the native types (int, float, char, etc) to avoid potential size mismatches with the parameter types.
Autorelink	Either auto-relink-enable or auto-relink-disable the directory in the $zroutines for the life of the process.
	Use the same value of $gtm_linktmpdir for all processes. All processes that share a directory whose contents are subject to ZRUPDATE use the same value for $gtm_linktmpdir so that all processes see update notifications - with different values of $gtm_linktmpdir, a ZRUPDATE by a process with one value of $gtm_linktmpdir would not be observed by a process with a different value of that environment variable.

Don'ts

Source and Object Files	Never change the name of an object file.
	When forming routine names, the compiler truncates object filenames to a maximum length of 31 characters. For example, for a source file called

Adatabaseenginewithscalabilityproven.m the compiler generates an object file called Adatabaseenginewithscalabilityp.o. Never let GT.M routines file names exceed 31 characters.

kill -9

Don't killing a process with **kill -9** and it may cause database damage. Use MUPIP STOP or MUPIP INTRPT instead. Use kill -9 as the last resort if the process does not respond to MUPIP STOP. **kill -9** terminates the process abruptly and may leave database files improperly closed and require a MUPIP RUNDOWN. Because kill -9 may cause database damage, perform a MUPIP INTEG immediately after a kill -9.

Operate as root

Never run a routine as root.

Other than GT.M installation, never perform any GT.M operation as root.

Triggers

Never use chained and nested triggers that potentially update the same piece of a global variable. You should always assess the significance of having chained triggers for a database update especially because of the arbitrary trigger execution order.

Never access ^#t with DSE, except with guidance from your GT.M support channel. Manage trigger definitions with MUPIP TRIGGER and $ZTRIGGER().

Local variables

Never use exponential numeric form in the subscripts. It may lead to ambiguities. Because numeric subscripts collate ahead of string subscripts, the string subscript "01E5" is not the same as the numeric subscript 01E5.

Never SET $ZWRTACn "variables". They are used by GT.M to make ZWRITE output more useful but are not supported for any other purpose. They are only mentioned here because you may see them in the output of ZWRITE and ZSHOW "V".

You can use SET @ to process ZWRITE or ZSHOW "V" output containing $ZWRTACn variables for restoring an alias container variable to a prior state. While processing the output, never attempt to inject or manipulate $ZWRTACn lines as it may lead to unintended consequences or undermine the benefit you might achieve from using alias containers. Lines containing SET $ZWRTACn=<value> are no-ops unless they have a preceding SET $ZWRTAC="" and an alias container variable association. In the ZWRITE or ZSHOW "V" output of an alias container, SET $ZWRTAC lines appear in the order that GT.M expects for restoration. FIS can change the use of $ZWRTAC in GT.M at any time.