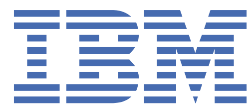
z/VM
7.3

CMS Application Multitasking



SC24-6258-73

Note:

Before you use this information and the product it supports, read the information in "Notices" on page 333.

This edition applies to version 7, release 3 of IBM® z/VM® (product number 5741-A09) and to all subsequent releases and modifications until otherwise indicated in new editions.

Last updated: 2023-05-15

© Copyright International Business Machines Corporation 1992, 2023.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures ... ix

Tables ... xi

About This Document ... xiii
- Intended Audience ... xiii
- Where to Find More Information ... xiii
  - Links to Other Documents and Websites ... xiii

How to provide feedback to IBM ... xv

Summary of Changes for z/VM: CMS Application Multitasking ... xvii
- SC24-6258-73, z/VM 7.3 (May 2023) ... xvii
- SC24-6258-73, z/VM 7.3 (September 2022) ... xvii
- SC24-6258-02, z/VM 7.2 (March 2021) ... xvii
- SC24-6258-01, z/VM 7.2 (September 2020) ... xvii

Chapter 1. Basic Multitasking Concepts ... 1
- Multitasking Process Model ... 1
  - Basic Constructs ... 2
  - Interactions of Threads, Processes, and Sessions ... 2
- Multitasking Functions ... 4
  - CMS Process Model Compared with OS/2 ... 8

Chapter 2. Process Management ... 11
- Creating Threads ... 11
- Thread Priority ... 11
- Dispatching Classes ... 11
- Implicit Process Creation ... 12
- Process Termination ... 12
- Process Management Examples ... 13

Chapter 3. Event Management ... 17
- Event Definition ... 17
- Event Signaling ... 18
- Event Monitors ... 19
- Event Monitor Processing ... 19
- Event Signal Processing ... 20
- Overview of Event Management Functions ... 20
- Event Management Examples ... 22

Chapter 4. Interprocess Communication ... 27
- Queue Definition ... 27
  - Operation ... 27
  - Properties ... 28
  - Queue Names ... 28
  - Queue Name Scopes ... 28
  - Export Level Search Order ... 29

Primary Queue....29
Keys....29
Network-Level Queues....31
Authorization....35
Replies....36
Service Queues....37
Timeouts....37
Interactions with Process Management....38
Interactions with Event Services....38
General Queue API Notes....38
Interprocess Communication Examples....38
Setting Up Network-Level Queues....44

Chapter 5. Synchronization....47
Synchronization Examples....48
Accessing a Critical Section Protected by a Mutex....49

Chapter 6. Multiprocessor Configuration Control....57
Guidelines for Defining Virtual CPUs....57

Chapter 7. Timer Services....59
Timer Services Examples....59
DateTimeSubtract Examples....60

Chapter 8. Accounting Services....65
Accounting Services Examples....66

Chapter 9. Abend Services....67
Monitoring Error Events....68
Error Recovery....68
Retry Routines....69
Recovery in the High-Level Language Environment....69
Advanced Error Recovery....70
Interactions with ABNEXIT and Simulated MVS Recovery....70
Abend Services Examples....72

Chapter 10. Trace Services....75
The CMS Trace Table....75
User Application Information....76

Chapter 11. CMS Monitor Data....79

Chapter 12. Writing Multitasking Applications....81
VMMTLIB Callable Services Library....81
Programming Language Binding Files....81
Writing Multitasking Applications in C....83
Using the C POSIX Entry Linkage....83
Using the CMS Multitasking applmain() Linkage....83
Calling Multitasking Functions from C....85
Writing Multitasking Applications in Assembler....85
Calling Multitasking Functions from Assembler....86
Outline of an Assembler Application....87
Building an Assembler Multitasking Program....88
Writing Multitasking Applications in REXX/VM....89
Calling Multitasking Functions from REXX....89
General CMS API Considerations....92

Chapter 13. CMS Multitasking Function Descriptions.. 95
Notation Used in Parameter Descriptions..95
Using the Online HELP Facility..95
AbnormalEnd - Terminate a Process Abnormally...97
AccountControl — Define and Query Accounting Attributes.. 99
AccountIdentify — Identify an Accounting Entity..102
CondVarCreate — Create a Condition Variable... 104
CondVarDelete — Delete a Condition Variable...106
CondVarGetHandle — Get the Handle of a Condition Variable.......................................108
CondVarSignal — Signal a Condition Variable... 110
CondVarWait — Wait on a Condition Variable.. 111
DateTimeGet — Query Time and Date..113
DateTimeSubtract -- Compute Time Differences... 115
EventCreate — Create an Event Definition.. 128
EventDelete — Delete an Event Definition... 131
EventDiscard — Inhibit Further Propagation of Signals..133
EventEnable — Enable or Disable for Specific Events.. 135
EventModify — Modify an Event Definition.. 137
EventMonitorCreate — Define an Event Handling Environment.....................................139
EventMonitorDelete — Delete an Event Handling Environment.....................................142
EventMonitorEnable — Enable or Disable Specific Monitors...144
EventMonitorQuery — Obtain Information About an Event Monitor.............................. 146
EventMonitorReset — Reset the State of an Event Monitor..150
EventMonitorSelect — Start or Stop Monitoring by Specific Monitors............................ 152
EventQuery — Obtain Information about an Event Definition.......................................154
EventQueryAll — Obtain All Event Names and Monitor Tokens..................................... 157
EventRetrieve — Retrieve Data From an Event.. 159
EventSelect — Start or Stop Monitoring for Specific Events..161
EventSignal — Signal the Occurrence of an Event.. 163
EventTest — Test for the Occurrence of Events... 165
EventTrap — Define an Asynchronous Event Handler.. 167
EventWait — Wait for the Occurrence of Events.. 169
MonitorBufferGet — Obtain the Address of the CMS Monitor Data Area........................ 171
MutexAcquire — Acquire a Mutex...173
MutexCreate — Create a Mutex... 175
MutexDelete — Delete a Mutex.. 177
MutexGetHandle — Get the Handle of a Mutex.. 179
MutexRelease — Release a Mutex..181
ProcessCheckPoint — Take a Snapshot of the Process State...182
ProcessGetID — Obtain the ID of a Process... 184
ProcessQueryBlocked — Find Blocked Threads... 185
ProcessQuerySuspended — Find Suspended Threads...188
QueueClose — Close a Queue.. 190
QueueCreate — Create a Queue.. 191
QueueDelete — Delete a Queue... 193
QueueIdentifyCarrier — Identify a Communication Carrier.. 195
QueueIdentifyService — Identify a Service Queue...197
QueueOpen — Open a Queue...199
QueueQuery — Query Waiting Message Count... 201
QueueReceiveBlock — Receive a Message (Blocking).. 203
QueueReceiveImmed — Receive a Message (Nonblocking)... 206
QueueReply — Reply to a Message.. 208
QueueSend — Send a Message.. 210
QueueSendBlock — Send a Message and Block..212
QueueSendReply — Send a Message and Request Reply... 214
QueueSignalEvents — Signal Queue Events.. 216

SemCreate — Create a Semaphore... 218
SemDelete — Delete a Semaphore... 220
SemGetHandle — Get the Handle of a Semaphore.. 221
SemQueryValue — Query the Value of a Semaphore...223
SemReInit — Reinitialize a Semaphore's Value.. 224
SemSignal — Signal a Semaphore..225
SemWait — Wait on a Semaphore.. 226
ThreadCreate — Create a Thread... 227
ThreadDelay — Delay This thread.. 231
ThreadDelete — Delete Threads...232
ThreadGetID — Obtain the ID of the Calling Thread...234
ThreadQueryDispatchClass — Query a Thread's Dispatch Class.. 235
ThreadQueryEntryPoint — Query a Thread's Entry Point... 237
ThreadQueryParameterList — Query a Thread's Parameter List...238
ThreadQueryPriority — Query a Thread's Priority...240
ThreadQuerySuspendCount — Query a Thread's Suspend Count.. 241
ThreadQueryUserData — Query User Data Word...242
ThreadResume — Decrement a Thread's Suspend Count...243
ThreadSetDispatchClass — Set the Dispatching Class of Threads..244
ThreadSetPriority — Set the Dispatching Priority of Threads..246
ThreadSetUserData — Set User Data Word... 248
ThreadSuspend — Increment a Thread's Suspend Count...249
ThreadYield — Yield Control to Another Thread.. 251
TimerStartInt — Start an Interval Timer.. 253
TimerStartMicros — Start an Interval Timer.. 256
TimerStartTOD — Start a TOD Timer... 258
TimerStop — Cancel a Timer... 260
TimerStopAll — Cancel All Timers..262
TimerStopMicros — Cancel a Timer... 263
TimerTest — Query a Timer... 265
TimerTestMicros — Query a Timer... 267
TraceControl — Define and Queries Trace Attributes.. 269
TraceSignal — Signal a Trace Event...272
VCPUCreate — Create a Virtual Processor (Virtual CPU)... 274
VCPUSelect — Request Special Virtual CPU Dispatching.. 276

Chapter 14. System Exits.. 277
System Exit Linkage Conventions.. 277
General-Purpose Exits..277
Session Initialization Exit...277
Thread Initialization Exit..278
Thread Termination Exit...278
Root Process Exit... 278
Building a System Exits Module...279
Programming Language Environment Exits.. 279
Process Creation.. 280
Process Deletion.. 281
Thread Creation...281
Thread Deletion...282
Run a Routine in Context...282
Context Switching.. 283
Building a Language Environment Manager... 284

Chapter 15. Suggestions for Server Writers.. 287
Interrupt Handling... 287
Communication..287
Data Management..288

General Guidelines........ 288

Chapter 16. Using CMS Multitasking with OpenExtensions Services........ 291
CMS Events For OpenExtensions Signals........ 291

Appendix A. Return and Reason Code Values........ 295
For Process Management........ 295
For Synchronization........ 295
For Event Services........ 296
For Trace Services........ 298
For Accounting Services........ 298
For Interprocess Communication........ 298
For Timer Services........ 299
For VCPU Services........ 300
For CMS Monitor Data........ 301

Appendix B. CMS Trace Record Formats........ 303
Communication Trace Record Formats (Type 1)........ 303
Dispatch Trace Record Formats (Type 2)........ 304
Process Management Trace Record Formats (Type 3)........ 305
Language Adapter Trace Record Formats (Type 4)........ 305
Synchronization Trace Record Formats (Type 5)........ 305
Miscellaneous Trace Record Formats (Type 6)........ 306

Appendix C. Remote IPC Support........ 307
Functional Overview........ 307
Interface Definition........ 309
IPC0 QCRBs (Kernel Request to Carrier)........ 313
IPC1 QCRBs (Carrier Response to Kernel)........ 313
IPC2 QCRBs (Carrier Request to Kernel)........ 314
IPC3 QCRBs (Kernel Response to Carrier)........ 314
Usage Notes........ 314
APPC/VM Carrier Line Flows........ 315
Structure........ 316
Request Flows........ 316
Response Flows........ 317

Appendix D. Example of a C Multitasking Program........ 319

Appendix E. Supplementary Information on System Defined Events........ 331
System Event Characteristics........ 331
VMCONINPUT and VMCON1ECB........ 332
VMSOCKET Signal Data........ 332

Notices........ 333
Programming Interface Information........ 334
Trademarks........ 334
Terms and Conditions for Product Documentation........ 334
IBM Online Privacy Statement........ 335

Bibliography........ 337
Where to Get z/VM Information........ 337
z/VM Base Library........ 337
z/VM Facilities and Features........ 338
Prerequisite Products........ 340
Related Products........ 340

Additional Publications..340

Index.. 341

Figures

1. The CMS Multitasking, Multiprocessor Environment....1
2. Queue-Based Structure.... 14
3. Event-Based Structure....15
4. EventWait Example.... 22
5. EventTrap Example.... 23
6. EventTest Example....23
7. Broadcast Signals Example....24
8. Sequentially Propagated Signals Example.... 24
9. Loose and Bound Signal Limits Example....25
10. Process Level Events Example.... 26
11. $SERVER$ NAMES Entries for Network Queues....35
12. Creating and Opening Queues....39
13. Simple Message Transmission....40
14. A Rendezvous....40
15. Using QueueReply....41
16. Two Threads Sharing a Queue.... 42
17. Using Service Queues.... 43
18. Closing and Deleting Queues....44
19. Accessing a Critical Section Protected by a Mutex.... 49
20. Wait/Post Processing Using a Semaphore.... 50
21. Multiple Waiters Using a Semaphore.... 52
22. Producer Threads Supply Messages to Consumer Threads.... 53
23. Monitor Initialization....53

24. Monitor Write Procedure....54

25. Monitor Read Procedure.... 55

26. EventWait and Timer Services.... 60

27. Requesting and Collecting Accounting Data.... 66

28. AbnormalEnd with Error Recovery Specifying Retry Routine.... 72

29. Event Trap Deleting Failing Thread....73

30. Skeleton of an IPC Carrier (Kernel-Initiated Requests)....308

31. Skeleton of an IPC Carrier (Carrier-Initiated Requests).... 309

32. Flow of Requests and Responses for Distributed IPC.... 310

Tables

1. Multitasking Functions 5
2. CMS and OS/2 Process Management and Related APIs 8
3. CMS System Events 17
4. Queue Authorization Rules 35
5. DateTimeSubtract Example 1 62
6. DateTimeSubtract Example 2 62
7. DateTimeSubtract Example 3 63
8. DateTimeSubtract Example 4 63
9. DateTimeSubtract Example 5 64
10. Accounting Record 65
11. VMERROR Data 67
12. Modifiable Data Area 67
13. Trace Entry 76
14. Header Files for C Programs 81
15. Macros for Assembler Programs 82
16. COPY Files for REXX Programs 82
17. Special case functions when called from REXX/VM 90
18. Formats and Format Identifiers 117
19. Format Limits 121
20. Supported Combinations of Formats 122
21. DateTimeSubtract's Reckoning Dates 125
22. Run-Time Support Entry Points and Functions 279
23. Event Data 292

24. IPC0 QCRB Parameter Usage (Kernel Request to Carrier)....... 313

25. IPC1 QCRB Parameter Usage (Carrier Response to Kernel)....... 313

26. IPC2 QCRB Parameter Usage (Carrier Request to Kernel)....... 314

27. IPC3 QCRB Parameter Usage (Kernel Response to Carrier)....... 314

28. Request Header Record Formats....... 317

29. Response Header Record Formats....... 318

30. System Event Characteristics....... 331

About This Document

The purpose of this document is to describe how you can use the multitasking function of the IBM® z/VM® Conversational Monitor System (CMS) to develop and execute multitasking application programs using C/C++ for z/VM, C for VM/ESA®, Assembler H, High Level Assembler, or REXX/VM.

This document provides both introductory and tutorial information as well as detailed reference material. Read the introductory chapters and the chapter on writing multitasking programs before delving into the function descriptions chapter. Because many functions are provided for debugging and tailoring, as well as for addressing advanced programming situations, you will find it easier to start using CMS multitasking quickly if you begin with the overview material, proceed to the example programs, and consult the detailed reference material for those functions referred to by the overview sections. You need only a few functions to get started writing a multitasking application.

Intended Audience

This document is intended for C, REXX, and assembler language programmers who want to develop multitasking programs for the CMS environment.

Readers should know the C language, REXX, or Basic Assembler Language and have experience with z/VM programming concepts and techniques.

Where to Find More Information

For information about related publications, see the "Bibliography" on page 337.

Links to Other Documents and Websites

The PDF version of this document contains links to other documents and websites. A link from this document to another document works only when both documents are in the same directory or database, and a link to a website works only if you have access to the Internet. A document link is to a specific edition. If a new edition of a linked document has been published since the publication of this document, the linked document might not be the latest edition.

© Copyright IBM Corp. 1992, 2023

How to provide feedback to IBM

We welcome any feedback that you have, including comments on the clarity, accuracy, or completeness of the information. See How to send feedback to IBM for additional information.

© Copyright IBM Corp. 1992, 2023

Summary of Changes for z/VM: CMS Application Multitasking

This information includes terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations for the current edition are indicated by a vertical line (|) to the left of the change.

SC24-6258-73, z/VM 7.3 (May 2023)

This information includes editorial changes.

SC24-6258-73, z/VM 7.3 (September 2022)

This edition supports the general availability of z/VM 7.3. Note that the publication number suffix (-73) indicates the z/VM release to which this edition applies.

SC24-6258-02, z/VM 7.2 (March 2021)

[VM66201, VM66425] z/Architecture Extended Configuration (z/XC) support

With the PTFs for APARs VM66201 (CP) and VM66425 (CMS), z/Architecture® Extended Configuration (z/XC) support is added. CMS applications that run in z/Architecture can use multiple address spaces. A z/XC guest can use VM data spaces with z/Architecture in the same way that an ESA/XC guest can use VM data spaces with Enterprise Systems Architecture. z/Architecture CMS (z/CMS) can use VM data spaces to access Shared File System (SFS) Directory Control (DIRCONTROL) directories. Programs can use z/Architecture instructions and registers (within the limits of z/CMS support) and can use VM data spaces in the same CMS session. For more information, see *z/VM: z/Architecture Extended Configuration (z/XC) Principles of Operation*.

Information in the following topic is updated:

- "General CMS API Considerations" on page 92

SC24-6258-01, z/VM 7.2 (September 2020)

This edition supports the general availability of z/VM 7.2.

© Copyright IBM Corp. 1992, 2023

Chapter 1. Basic Multitasking Concepts

CMS provides a multitasking, multiprocessor environment for applications and servers. The multitasking services allow a program to sub-divide itself into multiple independently executable parts, and coordinate these execution streams so they together accomplish the objective of the program. These services also allow multitasking programs running in different virtual machines to communicate with each other and to coordinate their processing.

A multitasking application can also take advantage of multiple processors in the computer complex. Multiprocessor exploitation is supported in XA- or XC-mode virtual machines only. The other aspects of multitasking are equally supported in all virtual machine architectures.

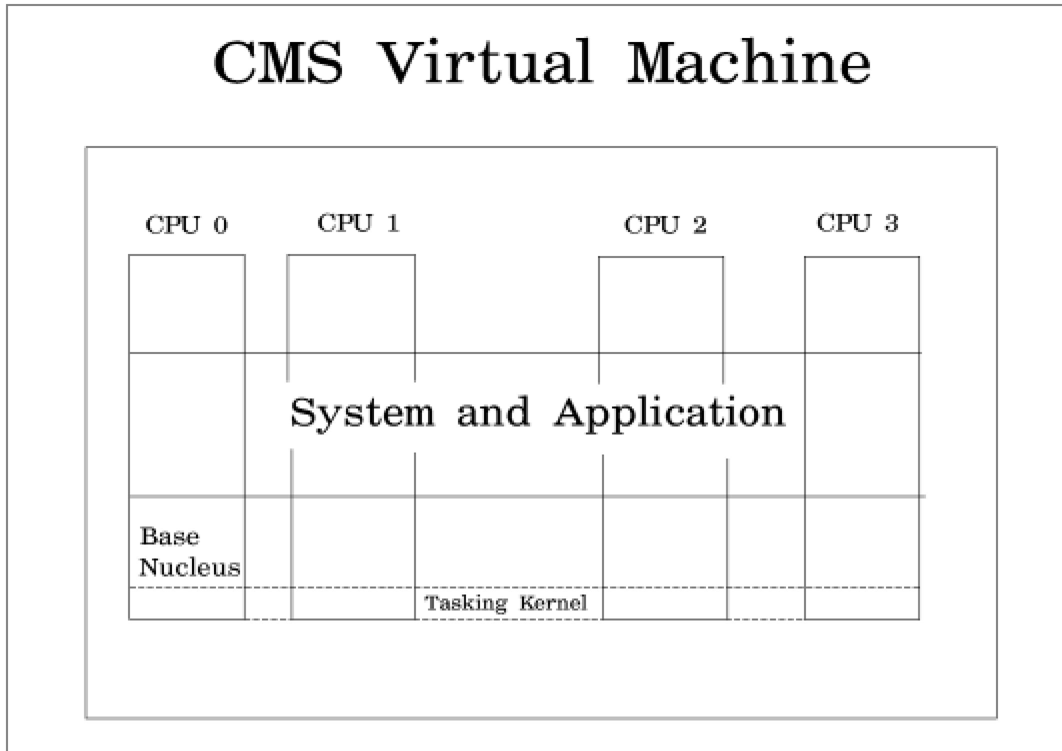


Figure 1. The CMS Multitasking, Multiprocessor Environment

The primary goals of the multitasking facilities are to provide support for high performance CMS-based servers and provide a means for CMS applications to harness the power of the multiprocessor capability of the underlying computer complex. They support an environment in which one multitasking application is in execution in a particular virtual machine at any one time. This allows a programmer to write a multitasking application, but it is not intended to provide a means for a CMS end-user to run multiple separate applications at the same time.

Multitasking Process Model

A process model defines the rules and relationships involved in writing multitasking applications and performing concurrent activities. Primarily it defines the units of execution, how these execution units interact, and how resources are managed relative to them.

The CMS process model defines concurrent programming concepts that promote distributed and parallel computation. CMS provides for multiple concurrent processes within a single application scope, and within each process multiple concurrent threads of execution may be established. In particular, its structure supports programmable workstation affinity and exploits the multiprocessing capability of the underlying VM system. In its fundamental concepts, the process model of CMS multitasking closely

© Copyright IBM Corp. 1992, 2023

resembles that of OS/2, and, where possible, the process management interfaces presented by CMS are a superset of those of OS/2. The CMS process model also provides the foundation for z/VM support of the IEEE POSIX standards 1003.1 (system services), 1003.1a (1003.1 extensions), and 1003.1c (threads).

Basic Constructs

The basic dispatchable entity in the system is the **thread**. The thread represents an instance of execution of a unit of program code. Its environment is characterized by a PSW, a set of register values, and a save area stack. At creation time, the system assigns an identifier to each thread. This identifier is unique within the process containing it and is not reused during the life of that process.

The locus of ownership of concurrent programming resources in the system is the **process**. A process consists of a collection of threads (at least, but not necessarily more than, one) performing related work. The resources which may be created or allocated, and thus owned, by a process include queues, synchronization objects, events, event monitors, and timers. The resources available to a process include those which it owns and those which are created with session scope by other processes in the same session. All threads in a process have equal access to the resources available to the process. When a process terminates, the resources which it owns are automatically deleted or terminated, as appropriate.

Each process has a name of arbitrary length and character composition supplied by its creator; the name must be unique at least within the session which contains the process but may be reused freely after the process terminates. The system also assigns an identifier to each process at its creation; this identifier is unique within the session containing it and is not reused during the life of the session.

A collection of processes sharing a common environment constitutes a **session**. All traditional CMS assets and facilities, such as storage subpools, the file system search order, the program name space, exit routines, library lists, set options, and global variables, belong to the session and are equally accessible to all its processes. Additionally, the session environment includes the virtual configuration of processors and I/O devices provided by CP and all resources owned by processes in the session which are created with session scope. Each session has a primary address space that is shared by all its processes. The session may also have access to one or more data spaces that may be shared with other sessions occupying the same real memory.

Interactions of Threads, Processes, and Sessions

Explicit thread interactions are confined to the bounds of an individual process because the model intentionally provides no means for a thread in one process to identify a particular thread in a different process; implicit interactions of threads in different processes may, however, occur as a result of dispatching decisions conditioned by dispatching class membership or the use of event management, interprocess communication, or synchronization services. Interprocess communication is provided between processes in either the same or different virtual machines, possibly even at different nodes of a communications network; other interactions between processes are confined to a single session.

The initial session (only one session is currently supported) is created implicitly during the IPL of CMS in the virtual machine. Creation of a session causes creation of an initial process known as the *root process* with the architected name **Root** which is constant and well-known in all CMS sessions. The root process performs session initialization and creates a number of threads dedicated to performing standard system functions or managing shared facilities (for example, the virtual machine timers).

Processes are created in a hierarchical structure whose depth is limited only by the virtual memory available to the session. A process may control the execution of any of its descendants by suspending or resuming it, altering its relative priority, or even terminating it. Termination of a process by default causes termination of all its descendants; termination of the root process causes termination of the session.

The creator of a process specifies the program which the process is to execute. One such program is the CMS command interpreter, which reads and executes commands from the terminal; a process running the command interpreter is referred to as a **commands process**. During initialization the root process creates a commands process with the architected name **Commands**. Conventional CMS commands and non-multitasking CMS applications are executed directly in the commands process on the command interpreter thread. Starting a multitasking application in the commands process causes the implicit creation of a child process in which the multitasking program is actually executed while the command

interpreter thread waits for its completion. The name of such an implicitly created process is the name of the program whose invocation caused it to be created, suffixed with a timestamp if necessary to produce a unique name within the session. If a thread of a multitasking program subsequently invokes another multitasking program through the CMSCALL interface, another child process is implicitly created for it.

Note: The native CMS Multitasking application programming interface does not permit the explicit creation of additional processes.

Interprocess communication in CMS is based on queues. Creating a process establishes a **primary queue** on which messages may be sent to the process. The primary queue bears the name of the process and has session-level scope by default. The primary queue may not be deleted except during process termination. Additional queues may be created and deleted as required. The threads in a single process can communicate and coordinate work using queues. Additionally, queues can be used to communicate with other processes in the same session or even in different virtual machines. See Chapter 4, “Interprocess Communication,” on page 27 for more information on queue management.

Creating a process also creates an initial thread on which the specified program starts executing. Any thread may create additional threads in its process and may freely suspend, resume, terminate, or alter the priority of any other thread in the same process. All threads within a process are peers; normal termination of one thread has no effect on other threads in the process. When the last thread in a process terminates, the process itself terminates.

At creation each process is assigned a relative priority that defaults to the priority of its creator. The priority of the process determines how its threads are treated by the dispatcher relative to all other processes in the session. In addition, at creation each thread is assigned a priority relative to other threads in the same process. The priority of a thread may be altered by any thread, including itself, in the same process. The priority of a process may be altered by any ancestor of the process or by the process itself. In particular, the root process may alter the priority of any process in the session. The actual dispatching priority of a thread is determined by combining its own priority with the priority of its process in such a way that if two processes have different process priorities, all threads in the higher priority process will have higher dispatching priority than all threads in the lower priority process.

Note: An implicitly created process has the same priority as the process in which it was created, and its initial thread has the same relative priority, and hence the same dispatching priority, as the thread on which it was created.

CMS uses a few basic principles to regulate the passing of control of a processor from one thread to another. First, a thread can lose control of a processor *only* when it calls one of a subset of the CSL routines described in this book. Such calls are easily identified: they all share the trait of having the potential to adjust the dispatchability of one or more threads in the system. QueueSend, EventSignal, and SemSignal are examples of such routines, while QueueQuery and EventMonitorCreate are not. Second, loss of control of a processor is classified as either involuntary or voluntary. Involuntary loss of control, also known as **preemption**, occurs even though the thread itself does nothing that would cause it to become unable to continue executing. Voluntary loss of control, which includes **blocking** and **yielding**, occurs when the thread performs some function that makes it unable or unwilling to continue to execute. An example of this is calling EventWait: the thread cannot continue executing until the wait is satisfied. A list of the functions that can cause a thread to lose control voluntarily can be found in Chapter 2, “Process Management,” on page 11.

CMS groups threads into **dispatch classes** to manage the thread switches that can occur at involuntary and voluntary losses of control. A thread can be preempted by (that is, involuntarily lose control to) only those threads residing in classes different from its own. A thread can never be preempted by a thread in its own dispatch class. When a thread blocks or yields, though, it can lose control to a thread from *any* dispatch class, including its own. Dispatch classes also govern parallelism. Threads in different classes can execute in parallel in a multiprocessor virtual machine, but only one thread from a given class can execute at a given time.

Note: The initial thread of an implicitly created process is assigned to the dispatching class of the thread on which it was created.

CMS provides facilities that let processes recognize and respond to the occurrence of named events either internal to the process or elsewhere in the session. It defines a set of system events, such as

exception occurred, *trace data became available*, and *accounting data became available*, that it signals at the appropriate times. Applications may also define their own events and may signal the occurrence of any defined event. A process may establish an event handler to be driven asynchronously when the event occurs, or may create a thread to wait explicitly for the occurrence. See Chapter 3, "Event Management," on page 17 for a complete description of event management services.

Normal termination of a thread occurs implicitly when it returns control through the last save area on its stack; other threads in the process continue unaffected. A thread may also explicitly request termination of itself alone or of the entire process containing it; either one of these methods also constitutes a normal termination. Normal termination of a process, whether by explicit request or as a result of normal termination of the last existing thread, causes *end of process* event handlers to be signaled with a normal termination code; when end of process event handling is complete, any remaining active threads in the process are terminated, all resources belonging to the process are deleted, and the process name becomes undefined and available for reuse.

Note: By the time control returns from a multitasking application to the command interpreter, any implicitly created processes will have been terminated and their resources deleted as described above. End of command cleanup in the commands process also deletes any resources which may have been created by nonmultitasking programs running directly on the command interpreter thread. Should the commands process itself terminate, the root process would automatically create a new one.

Abnormal termination of a thread may be requested either by the system, if a program check or other error has been detected, or by the thread itself, by explicitly signaling an error event. The rules for error handling depend upon the kinds of error handling environments which are extant at the time the exception occurs. Consider first the relatively simple case in which no CMS ABNEXITs or simulated MVS ESPIE or ESTAE exits exist in the session; in that case, error handling is localized to the process that caused the exception since the error event has process scope. If the process has previously established an error event handler, it is driven with all other threads in the process suspended; the event handler may elect either to let termination continue or to recover. If the process has established multiple error event handlers, they are driven in LIFO (last in, first out) order until one has elected to recover or all have chosen to continue termination. A parent process is also given the opportunity to perform error recovery for its child processes. If no applicable error event handler has been established in the process, or if all applicable error event handlers choose not to recover, abnormal termination of a thread induces abnormal termination of the entire process. The steps are the same as in the normal termination case except that the end of process event handlers are signaled with the abnormal termination code.

The presence of ABNEXITs, ESTAEs, or ESPIEs anywhere in the session complicates matters considerably. Because such exits have traditionally enjoyed session-wide scope, for compatibility they must be eligible to be driven on an appropriate exception occurring anywhere in the session. Programs using these facilities, however, are likely to be old nonmultitasking programs ill-equipped to recover from errors in new multithreaded programs. In a multitasking environment, therefore, these exit facilities are supported in such a manner as to maximize compatibility with the behavior of previous levels of CMS while minimizing the interference with the error event handling protocol. Multitasking programs must use error event handling for robust and predictable error recovery. See Chapter 9, "Abend Services," on page 67 for a complete description of abend services.

Multitasking Functions

An application program using CMS has at its command facilities to:

- Manage multiple threads of execution
- Handle asynchronous conditions
- Coordinate the execution of concurrent computation.

These facilities can be grouped into several broad areas of services:

- Process Management

  These services provide for the creation, deletion and control of threads. They are the fundamental services for creating a multitasking application.

- Event Management

  The complex tasks of handling asynchronous events and signaling conditions are handled by these services. These functions provide a means for defining events, signaling their occurrence, and taking action in response to the events.
- Interprocess Communication

  This set of functions provides for communication between the threads of a process and between different processes by means of message queues. A powerful feature of these communication functions is that the same functions are used for communication within one virtual machine and between virtual machines anywhere in a network.
- Synchronization

  Threads can serialize access to shared data and coordinate activity through these functions. They provide support for structured synchronization techniques and for basic locking.
- Multiprocessor Configuration Control

  CMS allows an application to harness the power of a multiprocessor complex by means of the CP virtual multiprocessor capability. While the application need not be sensitive to virtual multiprocessing, it can request from CMS that additional virtual CPU's be added to its virtual machine.
- Timer Services

  These services provide threads the capability to set and manipulate timers. Threads can wait for a time interval to expire, test to see if it has expired, or be asynchronously signaled on timer expiration.
- Accounting Services

  These services allow the application to account for resource use. They include the capability to group accounting information to correlate resource use with some other user on whose behalf service is being performed.
- Abend Services

  These services allow the application to request its own abnormal termination and provide for general abend recovery.
- Trace Services

  The tracing services provide for the collection of diagnostic trace information. Tracing for CMS and for the application is provided, as well as the capability to capture trace events as they happen.

The following table lists all the multitasking functions and their use. For more information on specific functions see Chapter 13, “CMS Multitasking Function Descriptions,” on page 95.

Table 1. Multitasking Functions

Function	Use
AbnormalEnd	Abnormally terminates a process
AccountControl	Defines and queries accounting attributes
AccountIdentify	Defines an accounting entity
CondVarCreate	Creates a condition variable
CondVarDelete	Deletes a condition variable
CondVarGetHandle	Gets the handle of a condition variable
CondVarSignal	Signals a condition variable
CondVarWait	Waits on a condition variable
DateTimeGet	Queries the time and date

Table 1. Multitasking Functions (continued)

Function	Use
DateTimeSubtract	Computes differences of times and performs time format and zone conversions
EventCreate	Creates an event definition
EventDelete	Deletes an event definition
EventDiscard	Inhibits further propagation of signals
EventEnable	Enables or disables for specific events
EventModify	Modifies an event definition
EventMonitorCreate	Defines an event handling environment
EventMonitorDelete	Deletes an event handling environment
EventMonitorEnable	Enables or disables specific monitors
EventMonitorQuery	Gets information about an event monitor
EventMonitorReset	Resets the state of an event monitor
EventMonitorSelect	Starts or stops monitoring by specific monitors
EventQuery	Gets information about an event definition
EventQueryAll	Gets all event names and monitor tokens
EventRetrieve	Gets data accompanying the occurrence of an event
EventSelect	Enables or disables for specific events
EventSignal	Signals the occurrence of an event
EventTest	Tests for the occurrence of events
EventTrap	Defines or deletes an asynchronous handler for events
EventWait	Waits for the occurrence of events
MonitorBufferCreate	Gets the address of the CMS monitor data area
MutexAcquire	Gets a mutex
MutexCreate	Creates a mutex
MutexDelete	Deletes a mutex
MutexGetHandle	Gets the handle of a mutex
MutexRelease	Releases a mutex
ProcessCheckPoint	Takes a snapshot of the process state
ProcessGetID	Gets the ID of a process
ProcessQueryBlocked	Finds blocked threads
ProcessQuerySuspended	Finds suspended threads
QueueClose	Closes a queue
QueueCreate	Creates a queue
QueueDelete	Deletes a queue
QueueIdentifyCarrier	Identifies an interprocess communication carrier

Table 1. Multitasking Functions (continued)

Function	Use
QueueIdentifyService	Identifies a service queue
QueueOpen	Opens a queue
QueueQuery	Queries the count of messages waiting
QueueReceiveBlock	Receives a message from a queue (blocking)
QueueReceiveImmed	Receives a message from a queue (nonblocking)
QueueReply	Replies to a message
QueueSend	Sends a message
QueueSendBlock	Sends a message and block
QueueSendReply	Sends a message and request reply
QueueSignalEvents	Signals queue events
SemCreate	Creates a semaphore
SemDelete	Deletes a semaphore
SemGetHandle	Gets the handle of a semaphore
SemQueryValue	Gets the value of a semaphore
SemReInit	Reinitializes a semaphore's value
SemSignal	Signals a semaphore
SemWait	Waits on a semaphore
ThreadCreate	Creates a thread
ThreadDelay	Delays the current thread
ThreadDelete	Deletes a thread
ThreadGetID	Gets the ID of the current thread
ThreadQueryDispatchClass	Queries a thread's dispatch class
ThreadQueryEntryPoint	Queries a thread's entry point
ThreadQueryParameterList	Queries a thread's parameter list
ThreadQueryPriority	Queries a thread's priority
ThreadQuerySuspendCount	Queries a thread's suspend count
ThreadQueryUserData	Queries a thread's user data word
ThreadResume	Resumes a thread
ThreadSetDispatchClass	Sets the dispatching class affiliation of threads
ThreadSetPriority	Sets the dispatching priority of threads
ThreadSetUserData	Sets a thread's user data word
ThreadSuspend	Suspends a thread
ThreadYield	Yields control to another thread
TimerStartInt	Starts an interval timer

Table 1. Multitasking Functions (continued)	
Function	**Use**
TimerStartMicros	Starts an interval timer
TimerStartTOD	Starts a TOD timer
TimerStop	Cancels a timer
TimerStopAll	Cancels all timers
TimerStopMicros	Cancels a timer
TimerTest	Queries a timer
TimerTestMicros	Queries a timer
TraceControl	Defines and queries trace attributes
TraceSignal	Signals a trace event with header information
VCPUCreate	Creates a virtual processor
VCPUSelect	Requests special virtual CPU dispatching

CMS Process Model Compared with OS/2

The following list shows the differences between the CMS and OS/2 process models:

- CMS provides only implicit creation of new processes, whereas OS/2 supports multiple explicitly created processes. The ability to explicitly create additional processes is not provided in CMS, but this is a restriction of the implementation only, not of the process model.
- CMS process and thread priorities are on a simple scale; OS/2 priorities are arranged in various classes. The OS/2 dispatcher time-slices; the CMS dispatcher may preempt but does not time-slice (but, of course, CP does).
- In OS/2, the initial thread in a process has special attributes and characteristics; in CMS, all threads in a single process are equivalent.
- In OS/2, synchronization is done through system and RAM semaphores; in CMS, all semaphores are essentially system semaphores, and other more structured synchronization mechanisms are provided.

The following table compares the CMS and OS/2 process management and related application program interfaces (APIs):

Table 2. CMS and OS/2 Process Management and Related APIs	
OS/2 Function Call	**Corresponding CMS Services**
DosCloseQueue	QueueClose
DosCreateQueue	QueueCreate
DosCreateSem	SemCreate
DosCreateThread	ThreadCreate
DosCWait	EventWait
DosEnterCritSec	ThreadSetDispatchClass
DosError	EventTrap
DosExecPgm	N/A (one process only)
DosExit	ThreadDelete
DosExitCritSec	ThreadSetDispatchClass

Table 2. CMS and OS/2 Process Management and Related APIs (continued)

OS/2 Function Call	Corresponding CMS Services
DosExitList	EventMonitorCreate, EventTrap
DosFlagProcess	EventSignal
DosGetEnv	ProcessQuery functions, ThreadQuery functions
DosGetInfoSeg	ProcessQuery functions, ThreadQuery functions
DosGetPid	ProcessGetID
DosGetPrty	ThreadQueryPriority
DosHoldSignal	EventTrap
DosKillProcess	N/A (only implicitly created processes)
DosMuxSetWait	CondVarWait, EventWait
DosOpenQueue	QueueOpen
DosPeekQueue	N/A
DosPurgeQueue	N/A
DosQueryQueue	QueueQuery
DosReadQueue	QueueReceiveBlock or QueueReceiveImmed
DosResumeThread	ThreadResume
DosSelectSession	N/A
DosSemClear	MutexRelease
DosSemClose	N/A
DosSemCreate	MutexCreate
DosSemOpen	N/A
DosSemRequest	MutexAcquire
DosSemSet	SemSignal
DosSemSetWait	SemSignal, SemWait
DosSemWait	SemWait
DosSendSignal	EventSignal
DosSetPrty	ThreadSetPriority
DosSetSession	N/A
DosSetSigHandler	EventTrap
DosSetVec	EventTrap
DosSuspendThread	ThreadSuspend
DosWriteQueue	QueueSend, QueueSendBlock, QueueSendReply, or QueueReply

Chapter 2. Process Management

Process management is the set of services that lets an application create, interrogate, and manipulate the threads within a process. The threads of an application share the resources of the process while the processor time allocated to the virtual machine in which the application is running is distributed among them.

The threads created through the process management services are termed *lightweight* because they own only their execution state, or context. This implies that threads require minimal overhead for the system to create, maintain and delete. The lightweight thread approach to multitasking lets the programmer create threads dynamically to handle new work and use threads to wait for asynchronous conditions or events. In general, the simplest programming approach is to allow the concurrency between threads to handle asynchrony while the program code running under the thread proceeds in a purely synchronous fashion. Later examples return to this point.

Creating Threads

A concurrent (or multitasking) application is a program with multiple threads of execution proceeding through the program's code over the life of the program. When an application is started, a process is created by the CMS services with an initial thread that begins execution at the main program entry point. This thread can then use the ThreadCreate function to create additional threads. This function takes as input the address (function name) where the thread should begin execution and returns a unique identifier of the thread, called the thread ID. All later references to the thread are made through its thread ID. The thread ID is an integer greater than zero.

When a thread is created, it is also assigned a priority and a thread class. These determine how the thread acts relative to other threads in the process.

For more information on the ThreadCreate function, see "ThreadCreate — Create a Thread" on page 227.

Thread Priority

Threads are dispatched based on priority. The highest priority runable thread is always the next to be dispatched. Priority values fall in a range of 0 through 32767 with higher values denoting higher priority. If all threads are assigned equal priority, a round-robin dispatching effect is achieved.

Dispatching Classes

In addition to assigning priority, creating a thread also assigns the thread to a dispatching class. A dispatching class is a set of threads with two properties:

- No thread in the class is ever given control in place of another in the same class unless the executing thread voluntarily gives up control.
- No two threads in a class are ever dispatched in parallel. That is to say, they are never in execution at the same time on different processors (CPUs). However, any thread can be preempted by or dispatched in parallel with any thread in a different class.

A parameter on the ThreadCreate call controls the assignment of the new thread to a class. The new thread is assigned either to the class of the creating thread or to a new class, putting the thread in *a class by itself*. Other threads can join this new class through subsequent ThreadCreate calls or through the class reassignment function, ThreadSetDispatchClass.

These dispatching classes provide a means to control parallelism. For example, if the application is a server that needs to use all the power of a large multiprocessor complex, each thread could be assigned to a different class, thereby allowing each thread to be dispatched in parallel. If, however, some part of the server could not execute in multiprocessor mode, the threads that execute in that particular code

© Copyright IBM Corp. 1992, 2023

could be assigned to the same class. This would let those particular server functions execute without explicit thread synchronization.

When using multiple threads in a class, the programmer should note that the following functions can result in another thread in the same class being dispatched. Each represents a voluntary yielding of control.

- ThreadYield
- ThreadSuspend
- ThreadDelete
- QueueSendBlock
- QueueReceiveBlock
- EventWait
- EventSignal
- EventMonitorReset
- MutexAcquire
- CondVarWait
- SemWait

These functions are documented in detail in Chapter 13, “CMS Multitasking Function Descriptions,” on page 95.

Implicit Process Creation

The process management services do not provide a way to explicitly create a process, but CMS will create a new process whenever a CMS SVC invocation is made to a program which was built with the multitasking initialization routine *VMSTART* or to an OpenExtensions™ application. If a thread issues a CMSCALL to invoke another module and that module is a multitasking program, CMS will create a new process and start its initial thread at the module entry point. The thread that issued the CMSCALL waits until the new process has completed.

A CMSCALL to a program that was not built with the VMSTART entry point or that is not an OpenExtensions application does not result in a process creation, but is simply another CMS SVC-level on the calling thread. This new program can also create other threads. Care must be taken however, because a process has only one language environment. So, if a module is built without the VMSTART entry point but does create threads, it implicitly depends on the fact that it will be invoked by a program of the same language that was built with the VMSTART multitasking initialization entry point.

Process Termination

The point at which the last thread in the application ends its execution is called *process termination*. At this point, CMS cleans up storage and other resources. In addition, it signals an event, known as the process end event, that can cause application post-processing to occur. This is similar in concept to the service call for nucleus extensions done by CMS.

The event handler for this event can do any clean-up needed. The following is a formal definition of this event. See Chapter 3, “Event Management,” on page 17 for an explanation of the concepts mentioned in the definition. The process end event has an event name of VMPROCESSEND. It is a session level, broadcast event that synchronizes the signalling thread. The data associated with the VMPROCESSEND event is as follows:

4-byte integer return code
4-byte integer abend code
16-byte character process name.

The key is the name of the process ending. If the process completes normally, the abend code is zero. Otherwise, it is the terminating abend code.

An additional consideration for process termination is the use of the CMSRET service from threads. Normally a thread should terminate by simply exiting (returning). If a thread issues the CMSRET service from its initial SVC-level, the same starting level it shares with the other threads in the process, CMS deletes the process. If a thread invokes another application through CMSCALL, a CMSRET on that thread causes control to return to the point of the CMSCALL as usual. It is only in the base SVC-level, the one that represents this multitasking application, that CMSRET corresponds to a process deletion.

Process Management Examples

To show meaningful examples, some of the other multitasking services must be introduced. However, the facilities introduced in this process management overview, together with the concepts described in the process model description in “Multitasking Process Model” on page 1, are enough to describe several concurrent program structures.

The simplest concurrent program structure is one having some number of threads, each having a unique function to perform and none interacting with each other until the end of the computation. Some numeric mathematics problems and some classes of transactions may be approached in this fashion. For example, a transaction composed of updating a file, a database, and a tape log could have three threads, each assigned to one of the three output objects, and each performing its update in parallel with the others.

For many servers, a more sophisticated structure may be more appropriate. In these servers, threads can be dedicated to individual services, individual clients (users of the server), or individual managed resources. Widely differing levels of coordination and synchronization may be required depending on the choice of structure.

A basic, but widely applicable, server structure is the queue-based model. In such a server, requests arrive from the client on a queue. The queue, being owned by the process, is equally accessible to all the threads in the process. Each thread processes a request based on the contents of a message received on the queue and then sends a reply back to a queue owned by the client. Each thread can handle a specific request (thus resulting in a server with one thread for each type of request supported by the server) or each thread can handle any of the possible requests.

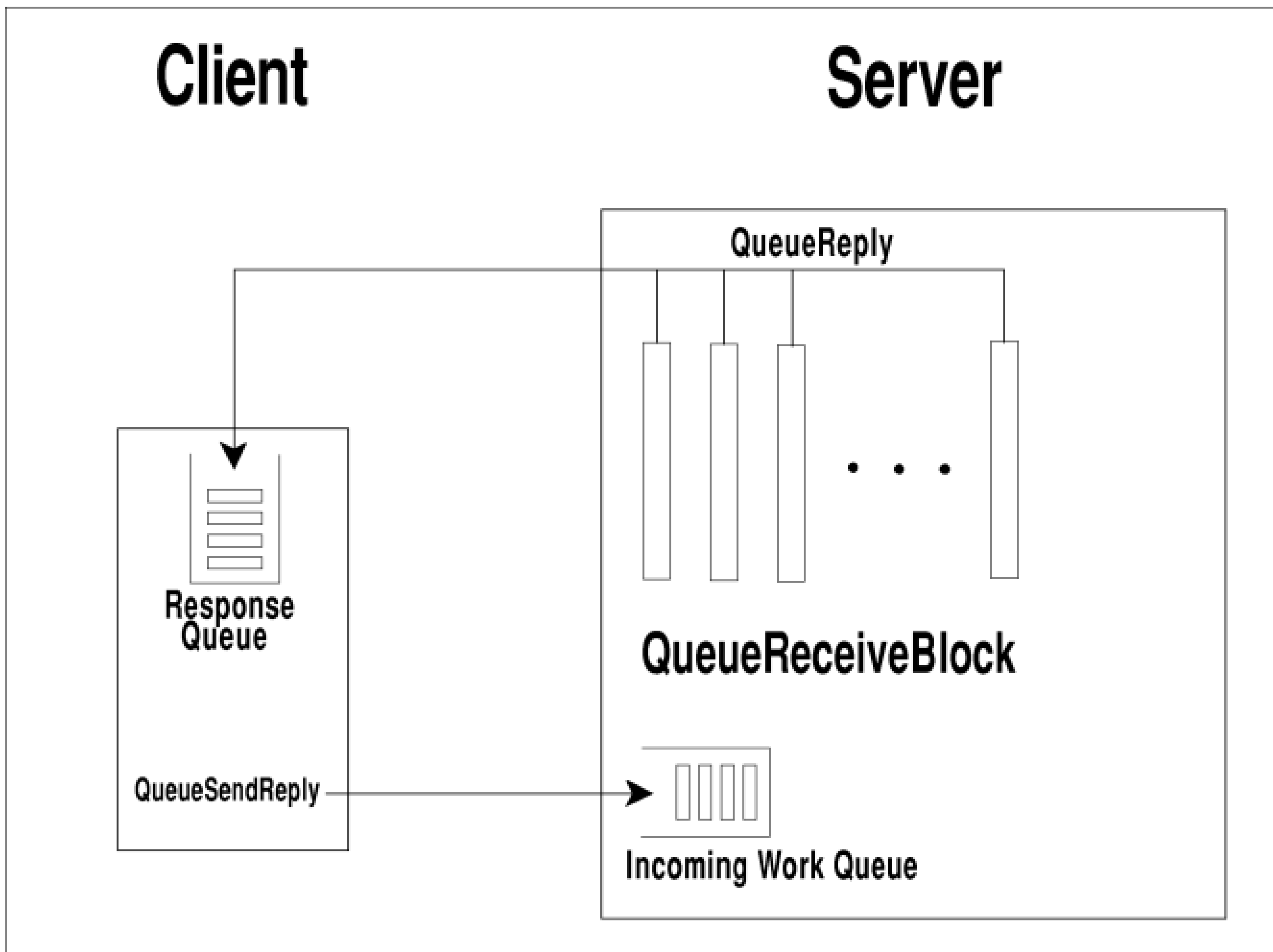


Figure 2. Queue-Based Structure

In either case, the basic structure of a thread in this model is:

```
Do While (server running)
  QueueReceiveBlock the request
          .
          .
          .
          Process the request
          .
          .
          .
  QueueReply the result back to the client
End
```

When the thread issues the QueueReceiveBlock function, it is left waiting until a message arrives on the queue. When processing is complete, the thread sends a response back to the client. It then loops back to receive the next request.

Another applicable structure for servers that use APPC/VM or IUCV to communicate with their clients is an event-based model. The approach translates interrupts to signals that can be handled using the event services. Instead of reading messages from a queue, the server threads wait on an event monitor. An interrupt exit, such as one established by CMSIUCV, handles interrupts by collecting the interrupt information and signaling an event. This signal then satisfies an event wait, letting a thread process the request. The result is sent back by the thread using the appropriate communication mechanism, such as APPC/VM.

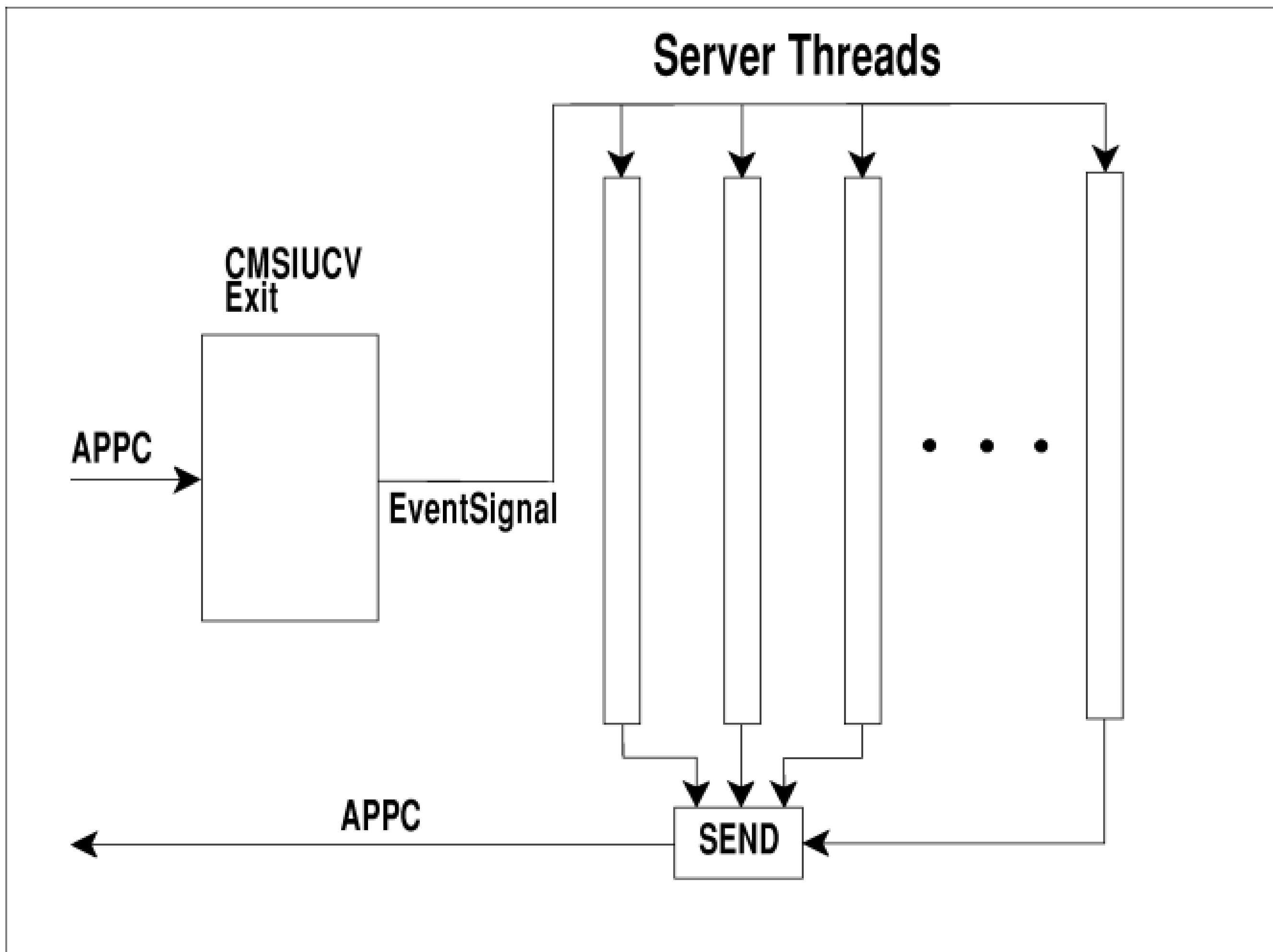


Figure 3. Event-Based Structure

A thread in this model closely resembles its counterpart in the queue model:

```
Do While (server running)
  EventWait for the request
        .
        .
        .
        Process the request
        .
        .
        .
  APPC/VM Send the result back to the client
End
```

As in the queue model, the thread waits at the top of the loop for the next request. In both models, the threads process their work sequentially, with the concurrency and asynchrony provided by using multiple threads.

Chapter 3. Event Management

Events define or represent activities that occur during program execution that may be of interest to application programs or to CMS itself. These activities may include both *hardware* events, such as interrupts, and *software* events, such as abnormal termination. The functions provided by CMS for managing events fall into the following categories:

- Event definition
- Event signaling, or indicating the occurrence of an event
- Event monitoring, or defining what signals a process is interested in observing
- Event monitor processing
- Event signal processing

Event Definition

An event definition is provided to CMS by the issuer of the EventCreate function. The event definition includes an event name, as well as the definition of the characteristics of this event. An **event name** is a character string of arbitrary composition that provides the primary identification of an event. It identifies the event for signaling and monitoring purposes. The event name may have either process or session scope. In either case, the name must be unique among all event names known within that scope.

For a description of the characteristics available to be included in the event definition, see the description of the EventCreate function in "EventCreate — Create an Event Definition" on page 128.

System events are defined by CMS either during session initialization or at first invocation of the service that employs the event. Their occurrence is signaled by CMS as appropriate. Table Table 3 on page 17 lists the CMS system events.

Table 3. CMS System Events

Name	Event	For more information, see...
VMACCOUNT	Production of an accounting record	Chapter 8, "Accounting Services," on page 65 of this book
VMCONINPUT	Unsolicited attention received at the console	"VMCONINPUT and VMCON1ECB" on page 332 of this book
VMCON1ECB	Input available at the console	"VMCONINPUT and VMCON1ECB" on page 332 of this book
VMCPIC	CPI Communications event	*Common Programming Interface Communications Reference* and *z/VM: CPI Communications User's Guide*
VMERROR	Beginning of abnormal termination of a process	Chapter 9, "Abend Services," on page 67 of this book
VMERRORCHILD	Notification of a child process abend	Chapter 9, "Abend Services," on page 67 of this book
VMIPC	Arrival of a message on a queue	"QueueSignalEvents — Signal Queue Events" on page 216 of this book
VMPOSGNL	Generation or delivery of a POSIX signal	Chapter 16, "Using CMS Multitasking with OpenExtensions Services," on page 291 of this book

© Copyright IBM Corp. 1992, 2023

Table 3. CMS System Events (continued)

Name	Event	For more information, see...
VMPROCESSEND	End-of-process cleanup	"Process Termination" on page 12 of this book
VMSFSASYNC	Completion of an SFS asynchronous request	*z/VM: CMS Application Development Guide*
VMSOCKET	Completion of the REXX Sockets Select function	"VMSOCKET Signal Data" on page 332 of this book; also see *z/VM: REXX/VM Reference*
VMTIMECHANGE	Dynamic time zone change	Chapter 7, "Timer Services," on page 59 of this book
VMTIMER	Expiration or cancellation of a timer	Chapter 7, "Timer Services," on page 59 of this book
VMTRACE	Production of a trace record	Chapter 10, "Trace Services," on page 75 of this book

For information about the characteristics of each system defined event, see "System Event Characteristics" on page 331.

Application programs may define other events for their own use; CMS provides identical event management services for system and user events. By convention, system event names have a maximum length of 24 bytes, and consist of uppercase and lowercase alphabetic, numeric, and break characters.

Event Signaling

The EventSignal function defines and communicates the occurrence of events. The signal is always associated with the process under which EventSignal is executed. An event may be defined so that its signal is recognizable only within the process in which it occurred or throughout the session containing that process. Any program may signal any event defined to the process in which it is executing. In particular, application programs are free to signal events defined by other applications or by CMS itself. The signaling program must determine when it is appropriate to signal any particular event.

The EventSignal function lets data be associated with a signal. The signal may be further characterized by specifying any portion of the signal data as a *key*. For example, a timer event carries a key that is the unique identifier of the particular timer that has resulted in this signal. The issuer of the EventSignal function provides the key. A process monitoring for such signals may use the key to specify the particular occurrences of an event that are of interest.

Interpreting the event signal data rests on a private protocol between the event signaler and the event signal processors and must be established separately for each named event; event management simply provides a channel for delivering this data.

Each process in the range of the signal is examined to determine if this process is monitoring for this signal. If a process is performing such monitoring, the signal is delivered to it. If multiple processes are monitoring for this signal, the signal is delivered to all of them concurrently. If no process in the range of the signal is monitoring for this signal, the signal is said to be *loose*. The number of loose signals of an event that are retained is established when the event definition is created. When that limit is exceeded, the oldest signals are discarded as new ones occur. Loose signals can be processed later, as described below.

For details of the EventSignal function, see "EventSignal — Signal the Occurrence of an Event" on page 163.

Event Monitors

The signals of an event or set of events may be monitored by establishing an *event monitor* with the EventMonitorCreate function. A process may monitor any signals that occur within itself and those signals whose occurrence has session-wide visibility that occur in other processes in the session.

If all signals of an event or set of events are not of interest, the issuer of the EventMonitorCreate function may choose to monitor particular signals of an event by specifying a key, possibly including special characters, which is matched against the key provided by the signaler. The special characters supported, and the matching rules applied when they are used, are the same as those used for match keys by interprocess communication. When a signal is observed that matches one of the event list entries specified by the EventMonitorCreate function, that signal is said to be *bound to the monitor*.

The event monitor defines the condition under which bound signals may be processed. When the condition specified by the monitor is satisfied, the monitor is said to be *eligible for activation*. A monitor is actually activated only when one of the monitor processing functions (EventTrap, EventWait, and EventTest) is issued against a monitor that is eligible for activation.

Each event list entry specified by the EventMonitorCreate function establishes the beginning of a list of zero or more matching signals that are bound to the monitor in order of delivery. When the monitor is activated, the first signal on each such nonempty list is collected to form the current signal set of the monitor. Each time the monitor is activated, only the signals that make up the current signal set may be processed. Signals that arrive while the monitor is active are not added to the current signal set and are not visible at least until the current activation is complete. However, a signal that is delivered while the monitor is inactive, even after sufficient signals have been bound to satisfy the monitored condition, becomes part of the current signal set at activation if no equivalent signal is already bound.

The number of bound signals that are retained is specified when the monitor is created. When the limit is exceeded, the oldest bound signal of a particular type (excluding a member of the current signal set if the monitor is active at the time) is automatically removed to make room for a new arrival.

For details of the EventMonitorCreate function, see "EventMonitorCreate — Define an Event Handling Environment" on page 139.

Event Monitor Processing

The monitor processing functions are EventTest, EventWait, and EventTrap. The processing performed by each of these functions depends on the condition of the monitor against which the function is issued.

If the monitor is not yet eligible for activation:

- EventTest reports on the state of each event list entry.
- EventWait suspends the invoking thread until the monitor is eligible for activation. At this time, the monitor is activated, the thread is resumed, and the state of each of the event list entries is reported.
- EventTrap informs CMS which routine, referred to as a *trap routine*, is to be run when the monitor is eligible for activation. When this time comes, the monitor is activated and the trap routine is run on a thread that CMS creates to run the trap routine. Because CMS does not know what environment the trap routine will run in, no information is initially passed to the trap routine. The trap routine must, therefore, issue EventTest to get a report on the state of each of the event list entries. If, at the time the monitor is eligible for activation, both an EventTrap and an EventWait have been issued, the EventWait takes precedence and the trap routine is *not* run.

If the monitor is eligible for activation, but is not yet activated:

- EventTest activates the monitor and reports on the state of each of the event list entries.
- EventWait activates the monitor and reports on the state of each of the event list entries.
- EventTrap activates the monitor and runs the specified trap routine on a thread created by CMS to run the trap routine.

If the monitor is currently active:

- EventTest reports on the state of each of the event list entries.
- EventWait inactivates the monitor and suspends the invoking thread until the monitor is again eligible for activation.
- EventTrap informs CMS what routine is to be run at the time the monitor is again eligible for activation.

Only one activation of an event monitor at a time is permitted; however, multiple event monitors may be active simultaneously. While an event monitor is active, signals for which it is monitoring are queued to it (subject to the bound signal limits established at monitor creation) for consideration when the current activation is complete.

A monitor is *reset* when it changes from being active to inactive. A monitor is reset implicitly in one of two ways:

- When a thread currently associated with an active monitor returns control
- An EventWait is issued against an active monitor.

A monitor is reset explicitly when an EventMonitorReset is issued against an active monitor.

When a monitor is reset, the current signal set is released automatically and any bound signals are examined to determine whether the monitored condition is again satisfied. If so, the monitor is eligible for immediate reactivation.

Because an active event monitor is associated with the thread executing the signal processing program, terminating this thread automatically resets all of its active event monitors.

If multiple monitors existing in a single process are monitoring for the same signal, the way the signal is delivered depends on how the event is defined:

- The signal may be broadcast to all monitors in the process simultaneously.
- The signal may be presented to one monitor at a time in the order of their creation (either FIFO or LIFO order).

In either case, the signal is delivered to a set of monitors that is established at the time the signal is received. Once the signal begins to be delivered to a sequence of monitors, no newly created monitors are added to the sequence.

Signals presented sequentially are said to be *propagated* from one monitor to the next. The propagation does not occur until the signal is released from the current signal set of the first monitor as a result of the monitor being reset. Each propagated signal is bound to the next monitor and ultimately becomes part of its current signal set. When the last monitor has released the signal, it is finally discarded.

Event Signal Processing

While a monitor is active, the signals in the current signal set may be processed. The signal processing program executes the EventRetrieve function to get the signal data from each signal in the current signal set. If this event is defined as having sequentially-processed monitors, the signal processing program may use the EventDiscard function to prevent some or all of the current signal set from being propagated to event monitors of lower precedence.

When the signal processing program has finished processing the current signal set, it must reset the monitor either implicitly or explicitly. This allows the current signal set to be released and the monitor to become eligible for activation again, thus enabling additional signals to be processed.

Overview of Event Management Functions

The following functions are available in CMS for managing events:

- EventCreate

  Registers the name of an event and specifies how that event is to be managed. For details of this function, see "EventCreate — Create an Event Definition" on page 128.
- EventDelete

Deletes a previously created event. For details of this function, see "EventDelete — Delete an Event Definition" on page 131.

- EventDiscard

  Prevents signals in the current signal set of an event monitor from being propagated to successive event monitors. It is effective only for events defined to have sequential signal propagation. For details of this function, see "EventDiscard — Inhibit Further Propagation of Signals" on page 133.
- EventEnable

  Enables or disables monitor activation by specific event signals. After disabling for a particular event, signals of that event do not contribute to the activation of any monitor in the invoking process. Upon reenabling for an event, signals of that event bound while the event was disabled may again contribute to the activation of monitors in the invoking process. For details of this function, see "EventEnable — Enable or Disable for Specific Events" on page 135.
- EventModify

  Modifies the characteristics of an event definition previously created by the same process. For details of this function, see "EventModify — Modify an Event Definition" on page 137.
- EventMonitorCreate

  Specifies combinations of event names and keys identifying conditions whose occurrence the invoking process wishes to monitor. For details of this function, see "EventMonitorCreate — Define an Event Handling Environment" on page 139.
- EventMonitorDelete

  Deletes a previously created event monitor. For details of this function, see "EventMonitorDelete — Delete an Event Handling Environment" on page 142.
- EventMonitorEnable

  Enables or disables specific monitors. After a monitor is disabled, the monitor cannot become active. Upon reenabling, the monitor may again become active. For details of this function, see "EventMonitorEnable — Enable or Disable Specific Monitors" on page 144.
- EventMonitorQuery

  Gets information about the definition and status of a previously created event monitor. For details of this function, see "EventMonitorQuery — Obtain Information About an Event Monitor" on page 146.
- EventMonitorReset

  Indicates that processing of the current signal set of an event monitor is complete. For details of this function, see "EventMonitorReset — Reset the State of an Event Monitor" on page 150.
- EventMonitorSelect

  Starts or stops monitoring by specific monitors. After monitoring is stopped, no signals are bound to the specified monitor. On restarting monitoring, loose signals are bound to the specified monitor in accordance with the monitor definition. For details of this function, see "EventMonitorSelect — Start or Stop Monitoring by Specific Monitors" on page 152.
- EventQuery

  Gets information about an existing event definition, including a list of all event monitors defined in the current process that are sensitive to occurrences of the event. The EventMonitorQuery function can obtain further information about a particular event monitor. For details of this function, see "EventQuery — Obtain Information about an Event Definition" on page 154.
- EventQueryAll

  Gets the names of all events and the tokens for all event monitors visible to this process. EventQuery and EventMonitorQuery can obtain further information about events and event monitors. For details of this function, see "EventQueryAll — Obtain All Event Names and Monitor Tokens" on page 157.
- EventRetrieve

Retrieves data from an event signal in the current signal set of an active event monitor. For details of this function, see "EventRetrieve — Retrieve Data From an Event" on page 159.

- EventSelect

  Starts or stops monitoring specific event signals. After monitoring of a particular event is stopped, signals of that event are retained in accordance with the loose signal limit specified when the event definition was created. On restarting monitoring for an event, any retained signal is delivered to qualifying event monitors in accordance with the event and monitor definitions. For details of this function, see "EventSelect — Start or Stop Monitoring for Specific Events" on page 161.

- EventSignal

  Indicates the occurrence of the specified event and optionally passes data associated with the occurrence to any event monitors that have registered an interest in the event. For details of this function, see "EventSignal — Signal the Occurrence of an Event" on page 163.

- EventTest

  Checks the condition of an existing event monitor. For details of this function, see "EventTest — Test for the Occurrence of Events" on page 165.

- EventTrap

  Nominates a routine to receive control asynchronously when the condition defined by an existing event monitor is satisfied. For details of this function, see "EventTrap — Define an Asynchronous Event Handler" on page 167.

- EventWait

  Awaits the satisfaction of the condition defined by an existing event monitor. For details of this function, see "EventWait — Wait for the Occurrence of Events" on page 169.

Event Management Examples

Here are some simple flows indicative of how event management services may be used:

```
CMS or an application                 Application
---------------------                 -----------


EventCreate                           EventMonitorCreate
     (creates goofy event)                 (monitor all goofy events)

                                      Do forever

                                        EventWait (on goofy monitor)
 .
 .                                      When goofy event is signaled,
 .                                      control is returned to invoker
 .                                      of EventWait

EventSignal                             EventRetrieve
     (signal goofy event)                    (get goofy data)

                                        by looping back to EventWait,
                                        monitor is reset and application
                                        is waiting for next goofy event

                                      End
```

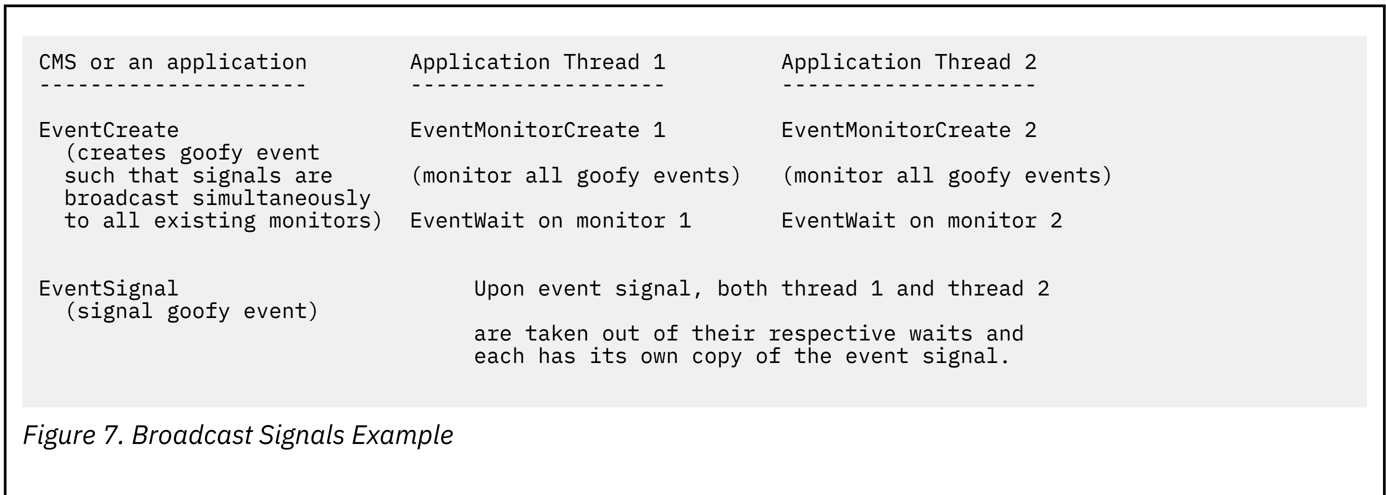
Figure 4. EventWait Example

```
CMS or an
application            Application                    Program X
-----------            -----------                    ---------

EventCreate            EventMonitorCreate
(creates goofy event)   (monitor all goofy events)
                       EventTrap (on goofy monitor,
                        specifying program X)

                       continue processing
 .
 .
 .
 .

EventSignal
(signal goofy event)                                  gets control when
                                                      goofy event signaled

                                                      EventTest (find out
                                                      how much goofy data
                                                      there is)
                                                           .
                                                           .
                                                           .
                                                      EventRetrieve
                                                       (get goofy data)

                                                      upon return from trap
                                                      routine, monitor is
                                                      reset and trap routine
                                                      awaits next goofy event

                                                              return
```

Figure 5. EventTrap Example

```
CMS or an application          Application
---------------------          -----------

EventCreate                    EventMonitorCreate
     (creates goofy event)          (monitor all goofy events where
                                     the goofy data starts with the
                                     letter "A")

EventSignal                    EventTest
     (signal goofy event            (on goofy monitor)
      with data "ABCD")
                               If monitor has been activated by EventTest

                                 EventRetrieve
                                    (get goofy data, in this case "ABCD")

                                 EventMonitorReset
                                      (to allow more goofy events to be
                                       processed)
                               end
```

Figure 6. EventTest Example



```
CMS or an application        Application Thread 1         Application Thread 2
---------------------        --------------------         --------------------

EventCreate                  EventMonitorCreate 1         EventMonitorCreate 2
  (creates goofy event
  such that signals are      (monitor all goofy events)   (monitor all goofy events)
  broadcast simultaneously
  to all existing monitors)  EventWait on monitor 1       EventWait on monitor 2


EventSignal                       Upon event signal, both thread 1 and thread 2
  (signal goofy event)
                                  are taken out of their respective waits and
                                  each has its own copy of the event signal.
```

Figure 7. Broadcast Signals Example

```
CMS or an application       Application Thread 1         Application Thread 2
---------------------       --------------------         --------------------

EventCreate                 EventMonitorCreate 1         EventMonitorCreate 2
   (creates goofy event
   such that signals are    (monitor all goofy events)   (monitor all goofy events)
   propagated to monitors
   sequentially)            EventWait on monitor 1       EventWait on monitor 2


EventSignal                 Upon event signal, thread 1
   (signal goofy event)     is taken out of its wait

                            EventRetrieve goofy data

                            EventMonitorReset monitor 1
                            to allow signal to be
                            propagated to next monitor   Upon reset of monitor 1,
                                                         thread 2 is taken out of its
                                                         wait and it may now process
                                                         the event signal.

                            EventWait on monitor 1       EventWait on Monitor 2

EventSignal                 Upon event signal, thread 1
   (signal goofy event)     is taken out of its wait

                            EventDiscard signal so that
                            this signal is NOT propagated
                            to next monitor

                            EventMonitorReset

                            Because the signal has been
                            discarded, it is NOT
                            propagated to the next
                            monitor, so thread 2 continues
                            to wait.
```

Figure 8. Sequentially Propagated Signals Example

```
CMS or an application                        Application
---------------------                        -----------

EventCreate
     (creates goofy event with a loose
     signal limit of 2)

EventSignal 1
     (signal goofy event)

Because no monitors exist this is a
loose signal.

EventSignal 2
     (signal goofy event)

Because no monitors exist, this is loose
signal number 2.

EventSignal 3
     (signal goofy event)

Because no monitors exist, this is also a
loose signal, but because a loose signal
limit of 2 has been established, signal 1
(the oldest loose signal) is now discarded.
This leaves signals 2 and 3 as the only
loose signals.

EventModify the goofy event to have a loose
signal limit of 3.

EventSignal 4
     (signal goofy event)

Because no monitors exist, this is a loose signal
Because the loose signal limit has been changed
to 3, no signals are discarded.  Signals 2,
3, and 4 are maintained as loose signals.
                                             EventMonitorCreate
                                                  (monitor all goofy events,
                                                  but sets a bound signal
                                                  limit of 3)

                                             Signals 2, 3, and 4 (previously
                                             loose signals) are bound to the
                                             monitor at its creation.

EventSignal 5                                Signal 5 is bound to the monitor,
     (signals goofy event)                   but because the bound signal limit
                                             is 3, signal 2 (the oldest bound
                                             signal) is now discarded.  This
                                             leaves signals 3, 4, and 5 as the
                                             only bound signals.
```

Figure 9. Loose and Bound Signal Limits Example

```
CMS                                 Application
---                                 -----------

EventCreate                         EventCreate
     (creates goofy event as             (creates goofy event as a
     a process level event)               process level event)

                                    EventMonitorCreate
EventSignal                              (monitor goofy events)
     (signal goofy event)

Because this is a process level
event, the signal is delivered
only to the signaling process.
Because there is no monitor for
this event in this process, the
signal becomes loose.
```

Figure 10. Process Level Events Example

Chapter 4. Interprocess Communication

CMS provides queue-based interprocess communication (IPC). The queue operations form a general mechanism for communication and coordination among processes and among threads within a single process. They are intended to aid both distributed processing and object-based programming. To describe how such programming approaches are supported by IPC, a few definitions must be covered.

Queue Definition

To define a queue, first define its elements. A **message** is a data element that is transferred by IPC. A message contains these elements:

- The **prefix**, which contains control information describing the message. The prefix consists of the following:

  text length
  The length of the message text associated with this message.

  key descriptor
  An ordered pair (*offset, length*) indicating the location within the message text of the message key. The message key is used in selective message manipulation.

  sender ID
  The user ID and process ID of the process that sent the message.

- The **message text**, which is the actual data to be conveyed, including a key describing the content or meaning of the data.

A **queue** is simply a list of such messages. Each queue is identified by a name assigned to it by its creator at the time it is created.

When a message is delivered to a recipient, the prefix and the message text are both delivered. The queue API breaks the prefix apart for the recipient — that is, the elements of the prefix are made available to the recipient in separate parameter list entries.

The message text may be up to 16MB-1 bytes long.

Operation

CMS provides a rich API for invoking queue functions. The functions provided fall into several overall categories:

Initialization
Creating a new queue, opening an existing queue, identifying a service queue

Transmission
Sending a message to a queue, sending a reply to a message received from a queue

Monitoring
Watching a queue or set of queues using an event monitor to detect message arrivals

Receipt
Receiving messages from a queue

Support
Identifying a carrier for remote IPC activity

Termination
Closing an open queue, deleting a queue.

The typical paradigm for using queues is roughly as follows:

- A thread creates a queue at a certain visibility scope. Typically the creating thread is the one wishing to receive messages from the queue, but other scenarios are certainly possible.

© Copyright IBM Corp. 1992, 2023

The QueueCreate function returns a numeric token, called a *queue handle*. All threads in the creating thread's process use this handle to identify the queue on all subsequent function calls.

- Threads in other than the creating process must open the queue with the QueueOpen function to use it. This is much like opening a file for I/O. The QueueOpen function also returns a queue handle.
- Threads may send messages to the queue, perhaps choosing to block until the message is received (the QueueSendBlock function) or perhaps choosing to specify that CMS should provide reply assistance (the QueueSendReply function). The former option enables distributed processes to rendezvous. The latter allows client-server interaction.
- Some thread in the creating process issues a receive call to gather up the message waiting in the queue (the QueueReceiveBlock or QueueReceiveImmed function).
- When a sender is finished, it closes the queue (with the QueueClose function). Closing the queue does not disturb the queue's contents.
- When the receiver is finished, it closes the queue and deletes the queue with the QueueDelete function. Only the process that created a queue may delete it.

Properties

CMS queues exhibit the following properties relative to delivery, sequence of arrival, and volatility:

- A successful return code on any of the send functions means that delivery of the message to the target queue has completed without error. This is true regardless of the location of the target queue. This kind of operation is usually called *guaranteed delivery*.
- Messages emanating from a single thread to a single target queue are guaranteed to arrive on the target queue in the order they were sent.
- CMS queues are volatile. If a queue is destroyed, its contents cannot be recovered.

Queue Names

Queue names are composed from single-byte characters chosen from an unrestricted alphabet (all 256 eight-bit patterns are eligible). A queue name may be up to 16MB-1 bytes long.

Queue Name Scopes

CMS defines several name scopes for queues. The operation of placing a queue in one of these scopes is called *exporting* the queue to that scope. This placement takes place at the time the queue is created and is fixed for the life of the queue. The export operation guarantees that the queue name is unique among all queue names exported to the target scope by processes sharing the target scope with the exporter.

The defined queue scopes are as follows:

Level
: **Definition**

Process
: The queue is visible only within a specific process, and the queue name is unique only within that process' process-level queues.

Session
: The queue is visible to any process executing in the same session as the exporting process, and the queue name is unique among all names exported to the session level by processes executing in the same session as the exporting process.

Network
: Like a session-level queue, the queue is visible to any process executing in the same session as the exporting process. The queue name is unique among all names exported to the network level by processes executing in the same session as the exporting process. The queue is visible to processes outside the owner's session, subject to the capabilities of the underlying communication carrier.

Local operation is not penalized by exporting beyond the session level — that is, the cost to access a queue located in the same session as the message sender is the same regardless of whether the target queue has been exported beyond the session level.

Each of these export levels promotes distributed processing. In particular, network-level queues promote interprocess communication from one user ID to another.

Export Level Search Order

Because of the nature of exporting, a queue name may be defined in more than one of a process' name scopes. For example, process A may be able to see different queues named ALPHA at both the process and session levels. This possibility makes it necessary to order the name scopes for search purposes.

The QueueOpen function searches the name scopes in the order requested by the invoker. If the invoker does not request a specific order, then the QueueOpen function searches the name scopes outward from the locus of the function caller; in other words, the function first searches for process queues, then session queues, and so on. The QueueOpen function returns an indication of the level at which the queue to be opened was found.

Primary Queue

When CMS creates a process, it creates a primary queue for the process. The name of the primary queue is the same as the name of the process owning it, and the handle of the primary queue is always 1. CMS sends messages to a process on its primary queue.

A process' primary queue is exported to the session level.

Keys

When a message is sent, the sender associates a byte string, called a *message key*, with the message. The key is embedded in the message text and may identify the content or intent of the message. The specific use of the key is left for the sender and receiver to decide.

For certain receiving and queue control operations, the caller specifies a *match key* to be used in selective manipulation of a queue or the messages in a queue. The requested operation applies only to queue messages whose message key matches the specified match key.

Message Keys

A message key is composed according to these rules:

- The message key must be completely contained in the message text.
- The length of the message key is bounded by the length of the message.
- The characters composing the message key are not restricted.
- It is permissible for the message key length to be zero. This simply means that the message has no message key associated with it.

CMS reserves some message keys for its own use for communicating with processes. All keys in messages sent by CMS are of the form VM*xxxxx*, where *xxxxx* represents an alphanumeric string.

Match Keys

The match key may be an *exact match key* or a *fuzzy match key*, where the terms are defined as follows:

Exact Match Key
: The match key contains no wildcard characters. Message keys against which the match key is compared must match the match key exactly (same length, same data) for the requested operation to have effect.

Fuzzy Match Key
The match key contains wildcard characters. Message keys against which the match key is compared must match the pattern specified by the match key, allowing for wildcards, for the requested operation to have effect.

The rules for composition of a match key are as follows:

- A match key is less than 16MB long.
- A match key containing one or more of the characters * (asterisk), % (percent), or ' (apostrophe) is considered to be a fuzzy match key. These characters, called *wildcard characters*, have special significance when the matching algorithm is applied.

The interpretation of wildcard characters in a fuzzy match key is very similar to the interpretation of those characters in CMS file names and file types. To be precise, the wildcard characters have the following meanings:

% (X'6C')
Matches any single character in a message key. For example, match key a%c matches message keys `abc`, `acc`, and `axc`.

*** (X'5C')**
Matches a variable-length (zero or more characters) substring within the message key. This usually indicates that the match key is actually a series of fragments, all of which must be present in the message key for a match to occur, but that the spacing between the fragments is irrelevant.

For example, message key `abcde` is matched by match keys a*, *de, a*e, and `*a*b*c*d*e*`.

' (X'7D')
Indicates that the next character in the match key should be interpreted literally (that is, without regard to whether it is a wildcard character or not). A character performing this function is commonly called an *escape character*.

Note that a match key may contain more than one kind of wildcard character. For example, message key `abcdefg` is matched by match key *b%d*.

To match any message key, specify either a zero-length match key or a match key of * (these two match keys are functionally equivalent). In the rest of this document, this match key shall be called the *match-all* match key.

Tips on Constructing Keys

For most programming situations, constructing message keys and corresponding match keys is pretty straightforward.

Note: Though this discussion is cast in the context of IPC keys, it also applies to keys used in event data and event monitor creation. Event data keys correspond to message keys, and event monitor keys correspond to match keys.

To build a message key, the programmer places some data in the key, said data being used to qualify the message in some way. To build a corresponding match key, the programmer places that qualifying data in the match key. CMS takes care of comparing the match key to the message key and taking action if a match occurs.

Consider a simple example, using character data. Suppose each message in a queue contains a key whose content is the name of a day of the week. To receive the next message whose key is *Wednesday*, the programmer would build a match key *Wednesday* and call one of the queue receive functions. CMS would deliver the next message having key *Wednesday*, avoiding messages keyed for other days of the week.

Subtle difficulties can arise, though, when the programmer desires to put binary data in a message key and construct binary match keys with the intent of matching said message keys. The difficulty comes not in placing binary data in the message key, but rather in constructing a match key that will have the desired effect. Suppose, for example, that a 4-byte message key is to contain a sequence number identifying the transmitted message. To receive a message having a desired sequence number, the programmer's first impulse might be to build the match key simply by making a 4-byte string whose bytes are the

bytes of the desired sequence number. For example, to receive the message whose sequence number is X'00000001', the programmer might build match key X'00000001', and so on for other sequence numbers. This would work fine for a little while, but it does not work for every sequence number.

The problem with binary match keys is that the binary data might contain a byte corresponding to one of the wildcard characters. Returning to the previous example, this programmer's code will work fine until it desires to receive the message whose sequence number is X'0000005C'. When the programmer uses this number as the match key, he will be asking to receive any message where the first three bytes of the key are X'00' — the rest of the key is irrelevant. This is because X'5C', the last byte of the match key, is the variable-length wildcard character, *.

There are ways of working around this pitfall with binary match keys. The first suggestion, of course, is to refrain from using binary data in message and match keys if possible. [1] When the message key and match key are both composed of character data, it is pretty unlikely that a match key could produce unexpected results. If it is not possible to use keys composed only of character data, then there are a couple of alternatives.

The first alternative is to ensure that binary data used in message and match keys can never contain a byte value that would be interpreted by CMS as a wildcard character. Returning again to the opening example, if the programmer's sequence number assignment scheme skips those numbers where one of the bytes in the 4-byte number is a wildcard character code point, then the sequence number can safely be placed in the message and match keys and all will work fine. In fact, it is pretty simple to generate a translation table which, when used with the TR instruction, produces the next valid sequence number from a known-to-be-valid one. CMS uses this technique in the generation of queue handles, timer tokens, thread IDs, and process IDs. It does this so that these values can be used successfully in event data keys or in message keys.

If it is not possible to avoid wildcard character code points in binary data, then the wildcard escape character must be used in the match key if key matching is to work as expected. In the above example, sequence number X'0000005C' can be matched by match key X'0000007D5C', for X'7D' is the code point for the escape character ('). A general technique for exactly matching an arbitrary binary value in a message key is to build the match key by inserting an escape character before each byte of the desired binary value. This ensures that no unexpected matches will be incurred. If the programmer mentioned above were to employ this technique, he would generate a match key of X'7D007D007D007D5C' for the X'0000005C' sequence number.

Network-Level Queues

Network-level queues allow IPC to occur between processes located in different sessions. Such operation represents IPC at the widest possible scope, that is, from one user ID to another. This facility can be used to let multiple users run a single cooperative application based on queues. It can also be used to implement servers based on IPC.

When a network-level queue is created, it resides in the same session as the process that created it. Other processes in that session access the queue with the same speed as they would have for session-level queues residing in that session. Processes in other sessions access the queue more slowly, because CMS uses a communication carrier on their behalf to reach the remote queue. In either case, the primitives (API calls) used to reach the queue are the same as those used to reach queues of other scopes.

Largely speaking, programs using the queue API need not be sensitive to whether the queues being manipulated are located at the network level. One point to keep in mind, though, is that network-level queues are one of the few constructs in CMS where *inter*process aspects of the CMS process model, that is, those process model traits relating to the interaction of one user process with another, come to light. When two CMS application programs communicate with one another through network-level queues, they are exposed to timing, synchronization, and scope-of-recovery effects not seen within a single CMS application. For example, consider a client-server situation where the client and server components are separate CMS applications, each one residing in its own virtual machine. Each must be written to expect that the partner might not yet be started; CMS may not automatically start a queue-based server in

[1] Patient: *Doctor, it hurts when I do this.* Doctor: *Well then, don't do that.*

response to QueueOpen requests from clients. Programmers need to keep in mind that they must account for such effects when writing distributed applications with CMS.

To facilitate the selection of retry strategies, the queue API provides reason codes indicating the kind of retry an application might choose to attempt. One such reason code indicates that an operation might succeed if the application simply retries it. The *out of storage* indicator is an example of this kind of reason code. Another reason code, returned only when network-level queues are involved, indicates that the connection to the remote CMS application has been lost. In this case, the application's only recourse is to try to re-establish the connection to the remote application by reopening the remote queue. Depending on the nature of the application, this more serious retry attempt might need to be coupled with some application-dependent recovery procedure.

CMS provides code supporting network-level queue operations using APPC/VM as the carrier. This allows applications to send messages to one another's queues within a VM collection (through TSAF) or across collections (through AVS). CMS also contains interfaces that allow customers to write line drivers to support queue operations over other carriers. For more information on writing IPC carriers, see Appendix C, “Remote IPC Support,” on page 307.

For a concrete example that shows how to set up network-level queues based on the following analytical discussion, see “Setting Up Network-Level Queues” on page 44.

Local Access Considerations

When an application attempts to use a network-level queue, CMS searches the local session for the queue, and if it is found, the queue is accessed as if it were of session scope. The application experiences the same access speed and access rights rules it would experience if the queue were truly of session scope.

Remote Access Considerations

CMS uses a names file, $QUEUES$ NAMES, to describe network-level queues of interest to the local session. This names file contains information meaningful to the CMS kernel, and it can also contain information meaningful to communication carriers. The following example of $QUEUES$ NAMES provides an explanation of the tags used in the file and the rules governing default tag values.

CMS requires a $QUEUES$ NAMES entry for each network-level queue residing remotely but being accessed locally. Each such entry lets CMS notify the appropriate carrier that it must initiate activity with some remote instance of CMS. CMS also requires a $QUEUES$ NAMES entry for each network-level queue created in the local session. This entry lets CMS notify the appropriate carrier that it must prepare to handle requests for the queue, that is, requests that might originate remotely.

$QUEUES$ NAMES is kept in storage; names are resolved from the in-storage image. Changes made to the disk image of the file are not recognized until the in-storage image is refreshed. The in-storage image can be refreshed with the SET COMDIR RELOAD command.

When CMS searches $QUEUES$ NAMES for an entry but does not find one, it proceeds with processing as if it had found the entry with all defaults applied. Thus, for many cases, it is not necessary to create $QUEUES$ NAMES at all.

```
***
*
*  $QUEUES$ NAMES -- names file to map IPC queue names to
*  carriers and carrier-specific parameters.
*
*  CMS uses the information it extracts from this file to
*  determine the carrier it should use for IPC operations.
*
*  CMS also passes what it finds here, along with any
*  defaults that may have needed to be filled in, to the
*  carrier for its use.
*
*
*  ENTRY FORMAT
*
*  Format of each entry is:
*
```

```
*     :nick.<queue_name>
*       :scope.NETWORK
*       :qn.<queue_name_at_remote_location>
*       :carrier.<your_carrier_name>
*
*  Note that the :scope tag is required, and its value must
*  be NETWORK (mixed case is OK).
*
*  The entry may also contain carrier-specific tags -- QueueOpen
*  will pass them to the carrier for its use.  For APPC/VM, the
*  following carrier-specific tags are defined:
*
*     Tag             Meaning
*     ---             -------
*     :sdn.           symbolic destination name
*
```

```
*  The APPC/VM carrier uses the value of the :sdn. tag as its
*  index into the CMS Communications Directory.  If a ComDir entry
*  is not found (that is, RC=80 or RC=84 from CMSIUCV RESOLVE),
*  then the APPC/VM carrier uses the following default values:
*
*     ComDir Tag      Default Value If No ComDir Entry Is Found
*     ----------      ----------------------------------------
*
*     :tpn.           VMIPC
*
*     :luname.        *USERID qn, where 'qn' is the first 8
*                     characters of the value of the :qn. tag
*                     from the queue's entry in $QUEUES$ NAMES
*
*     :security.      SAME
*
*  NOTE:  Case is significant in all entries.  (The case of
*  the tag itself is not significant, but the case of the value
*  is.  QueueOpen does not perform any case shifting on the
*  extracted values.)
*
*  DEFAULTS
*
*  If CMS does not find an entry in $QUEUES$ NAMES, or if it
*  finds a partial entry (some tags missing), it fills in
*  defaults as follows:
*
*     Tag             Default Value
*     ---             -------------
*
*     :qn.            Queue name as passed by caller,
*                      truncated to 249 characters
*
*     :carrier.       APPC/VM
*
*     :sdn.           Queue name as passed by caller,
*                      truncated to 8 characters
*
*  Note that CMS does not supply a default :sdn. tag if
*  a :carrier. tag having value other than APPC/VM is found.
*
***
```

```
*
*  Samples
*

:nick.Nick_Only
   :scope.NETWORK

:nick.QName_Only
   :scope.NETWORK
   :qn.My_Qname

:nick.Carrier_Only_TCP/IP
   :scope.NETWORK
   :carrier.TCP/IP

:nick.Carrier_Only_APPC/VM
   :scope.NETWORK
   :carrier.APPC/VM
```

```
:nick.SDN_Only
   :scope.NETWORK
   :sdn.My_SDN

:nick.QName_Missing
   :scope.NETWORK
   :carrier.APPC/VM
   :sdn.My_SDN
  :carrier.Wrap

:nick.Carrier_Missing
   :scope.NETWORK
   :qn.My_Qname
   :sdn.My_SDN
```

```
:nick.SDN_Missing_APPC/VM
   :scope.NETWORK
   :qn.My_Qname
   :carrier.APPC/VM
:nick.SDN_Missing_TCP/IP
   :scope.NETWORK
   :qn.My_Qname
   :carrier.TCP/IP

:nick.All_Data
   :scope.NETWORK
   :qn.Remote_Network_Queue
   :carrier.APPC/VM
   :sdn.RNQ
   :c1.Carrier_Specific_Parm_1
   :c2.Carrier_Specific_Parm_2

:nick.Local_Wrap_Queue
   :scope.NETWORK
   :qn.Remote_Wrap_Queue
```

APPC/VM Carrier Considerations

The APPC/VM carrier provides two basic functions. First, it provides processes in its own session with a means to access queues located in other sessions. Second, it provides a way for processes located in other sessions to access queues located in the local session.

The CMS carrier does not support the use of APPC/VM in wrap fashion. In other words, conversations managed by the APPC/VM carrier cannot originate and terminate at the same virtual machine.

Accessing Remote Queues

The CMS Communications Directory serves as the repository for APPC/VM-related information associated with network-level queues. CMS uses the value of the `:sdn.` tag from $QUEUES$ NAMES as a symbolic destination name for use with the CMS Communications Directory. The CMS Communications Directory gives the APPC/VM naming and addressing information needed to reach the remote queue.

The CMS Communications Directory entries describing remote instances of CMS must be set up as private resource manager entries. The format of these entries varies according to the relative locations (same collection, different collections, and so forth) of the two carriers. For more information on setting up private resource manager entries in the CMS Communications Directory, see *z/VM: Connectivity*.

The APPC/VM carrier uses the value of the `:qn.` tag from $QUEUES$ NAMES as the name of the queue at the remote location. It passes that name to the remote instance of CMS for its use in finding the queue.

Enabling Access to Local Queues

An entry for resource *VMIPC* in the CMS private resource registration file ($SERVER$ NAMES) enables access to network-level queues located in the local session. The entry's `:list.` tag defines those user IDs that may use network-level queues located in the local session. The `:module.` tag gives the name

of the exec or module CMS should start in response to the arriving IPC request. Figure 11 on page 35 shows the required format for the entry in $SERVER$ NAMES.

```
General form:

:nick.VMIPC
   :list.your_access_list
   :module.your_CMS_program

Example:

:nick.VMIPC
   :list.USER1 USER2 USER3
   :module.YOURPGM
```

Figure 11. $SERVER$ NAMES Entries for Network Queues

In addition to setting up $SERVER$ NAMES, the programmer must also ensure that the virtual machine is enabled for incoming APPC/VM conversations. This is most easily done by placing an `IUCV ALLOW` statement in the virtual machine's CP directory entry.

Considerations for Automatic Queue Program Startup

CMS performs automatic startup of the program named by VMIPC's `:module.` tag if and only if:

- SET SERVER is ON
- SET FULLSCREEN is OFF or SUSPEND
- The APPC/VM carrier has not yet initialized itself in the virtual machine.

The APPC/VM carrier initializes itself in the virtual machine as soon as one of these things happens:

- A CMS program creates a network-scope queue for which $QUEUES$ NAMES indicates that the APPC/VM carrier should be used
- A CMS program opens a network-scope queue that does not exist locally and for which $QUEUES$ NAMES indicates that the APPC/VM carrier should be used.

As long as the APPC/VM carrier remains uninitialized, the usual private resource manager behavior continues to apply to resource VMIPC. When a connection pending interrupt arrives for VMIPC, then provided SET SERVER and SET FULLSCREEN are set appropriately, the module or exec named by the `:module.` tag of VMIPC's entry in $SERVER$ NAMES is auto-started.

Once the APPC/VM IPC carrier has initialized itself, the auto-starting feature of CMS private resource management is no longer available for connections to VMIPC (other resource names are still auto-started). Subsequently-arriving connection pending interrupts for resource VMIPC do not cause the module named by VMIPC's `:module.` tag to be auto-started. Further, once it has initialized itself, the APPC/VM carrier is immune to the settings of SET SERVER and SET FULLSCREEN.

No matter whether the APPC/VM carrier is initialized yet or not, all connection pending interrupts for VMIPC are always validated according to the `:list.` tag for the VMIPC entry in $SERVER$ NAMES.

The APPC/VM carrier remains initialized and active in the virtual machine until CMS is re-IPLed.

Authorization

The authorization rules for queues are explained in Table 4 on page 35. Each cell in the table indicates the processes that may perform the given operation at the given level.

Table 4. Queue Authorization Rules

	Create	**Delete**	**Open, Write, Close**	**All Other Operations**
Process	Any process	The creator	The creator	The creator

Table 4. Queue Authorization Rules (continued)				
	Create	**Delete**	**Open, Write, Close**	**All Other Operations**
Session	Any process	The creator	Any process in the creator's session	The creator
Network	Any process	The creator	Any process in the creator's session, plus access from outside the session, subject to the authorization constraints imposed by the communication carrier	The creator

Operations included in the *open, write, close* category are those typically used by a client thread. Specifically, they are:

- QueueOpen
- QueueSend
- QueueSendBlock
- QueueSendReply
- QueueReply
- QueueClose.

Authorization rights for a network-level queue are a function of the communication carrier used to connect sessions together. For the APPC/VM carrier, if a user ID is named in the `:list.` tag of the *VMIPC* entry in $SERVER$ NAMES, then it may access any network-level queues located in that session. This access is limited to the *open, write, close* set of operations.

Replies

Typical IPC functions (send and receive), while suitable for communication among peer cooperating processes, are not particularly suitable for client-server operations, especially when a number of clients and a single server are involved. Consider this:

- To interact with the server, a client opens the server's incoming work queue (a network-level queue, probably) and places messages on it to request service. This implies that a client has an incoming work queue open for each server with which it interacts. This is acceptable.
- A server, to interact with a client, must open a queue owned by the client and send messages to it to indicate completion of service requests. This implies that the server has a client queue open for every client it serves. This is not acceptable. In addition, every client's response queue must be at the network level, because the server would not be able to see it if it resided at a lesser scope. This is not acceptable either.

To alleviate this condition, CMS makes available a send-receive queue function, QueueSendReply. This function lets a process send a message and specify the queue in which it would like the receiver of the message to place a reply. The sender identifies the reply queue by the handle on which it was opened. In response to this, CMS tags the message with a token; this token is passed to the receiver and is used by the receiver in place of a queue handle when it transmits the reply. CMS uses the token to place the message in the queue specified by the sender of the original message.

The sender must have write and read access to the reply queue to specify that a reply be placed in it. Because the access rights to the reply queue are determined at the time the reply queue is opened, the overhead of the QueueSendReply function is reduced. The replying process need not have opened the reply queue, nor need it have write access to the reply queue. In fact, the reply queue need not even be visible to the replying process under normal circumstances.

The use of the QueueSendReply function is as follows:

- A thread sends a message for which it needs a reply by using QueueSendReply.
- CMS assigns a *reply token* to the message and internally associates the reply queue identity and location with the reply token.
- When CMS delivers the message to the receiver, it also delivers the reply token.
- To reply to the message, the recipient uses the QueueReply function, passing CMS the message to be sent and the reply token previously given it by CMS.
- CMS uses the reply token to place the response in the reply queue.

The reply token is valid for only one call to QueueReply.

Service Queues

In client-server relationships, it is sometimes necessary to let a server hide the identity of a service queue from its clients. For example, a server may choose to create and delete work queues dynamically, hiding the true identities of said queues. In addition, it might be desirable to let a diagnostic tool or trace facility place itself between a server and its clients so that the requests and responses might be debugged or otherwise monitored. Insulating clients from the true identity of a queue enables these kinds of functions, among others.

To facilitate this, CMS provides the notions of *service IDs* and *service queues* and integrates these concepts into the queue API. The queue API provides a function for associating a session-level queue with a service ID, and it lets clients use service IDs in place of queue handles in the QueueSendReply and QueueSend functions. With these two features, CMS provides a way for clients to send messages to a server's queue, identifying the service queue by service ID. The client need not open the service queue to send messages to it. In fact, the client does not even need to know the name of the service queue.

The use of service queues goes roughly like this:

- The writer of a service thread makes the thread's service ID known in advance to client writers.
- When it initializes, the service thread calls the QueueIdentifyService function to tell CMS the name of the queue on which requests for this service should be placed.
- Clients wishing to contact the server use the QueueSendReply function, passing the service ID as the queue handle.
- At any time the server may call the QueueIdentifyService function again to change the queue in which requests for its service should be placed. CMS returns the name of the queue previously designated for this purpose. This facilitates the debugging and monitoring example described earlier.
- Subsequent invocations of the QueueSendReply function deliver messages to the new service queue.

A given queue may be pointed to by only one service ID at a time.

Timeouts

To assist in resolving deadlocks, the blocking queue functions (QueueReceiveBlock and QueueSendBlock) allow the caller to specify a timeout value. When the caller specifies a nonzero timeout period, CMS terminates the function when the time period expires, even if the desired operation has not yet completed.

Note: Specifying a timeout period of zero causes the call to block indefinitely (no timeout assistance is provided). For example, a call to QueueReceiveBlock would terminate even though a matching message had not yet arrived at the queue of interest. When a blocking function terminates because of timer expiration, the caller receives a return and reason code indicating that a timeout was the reason for the call's termination.

To keep both thread overhead and system overhead at a bare minimum, CMS uses a coarse-grained, low-overhead timing scheme for managing the timeouts of these blocking functions. The timeout facility is intended only to break deadlocks, ***not*** for precision timing of message exchanges or program throughput

rates. Programs requiring such precision are free to exploit CMS's timer and event APIs directly to achieve such results.

Interactions with Process Management

The following interactions between queues and process management should be noted:

- When a process terminates, any queues created by that process are deleted and messages waiting in those queues are deleted. Threads waiting on the receipt of such messages are notified that the messages were deleted.

Interactions with Event Services

To facilitate the coupling of message arrival waits with other kinds of waits, CMS optionally signals a process-level event, *VMIPC*, when a message arrives on a queue. Whether this event will be signaled or not is controlled by the QueueSignalEvents function. See the description of the QueueSignalEvents function "QueueSignalEvents — Signal Queue Events" on page 216 for more information.

General Queue API Notes

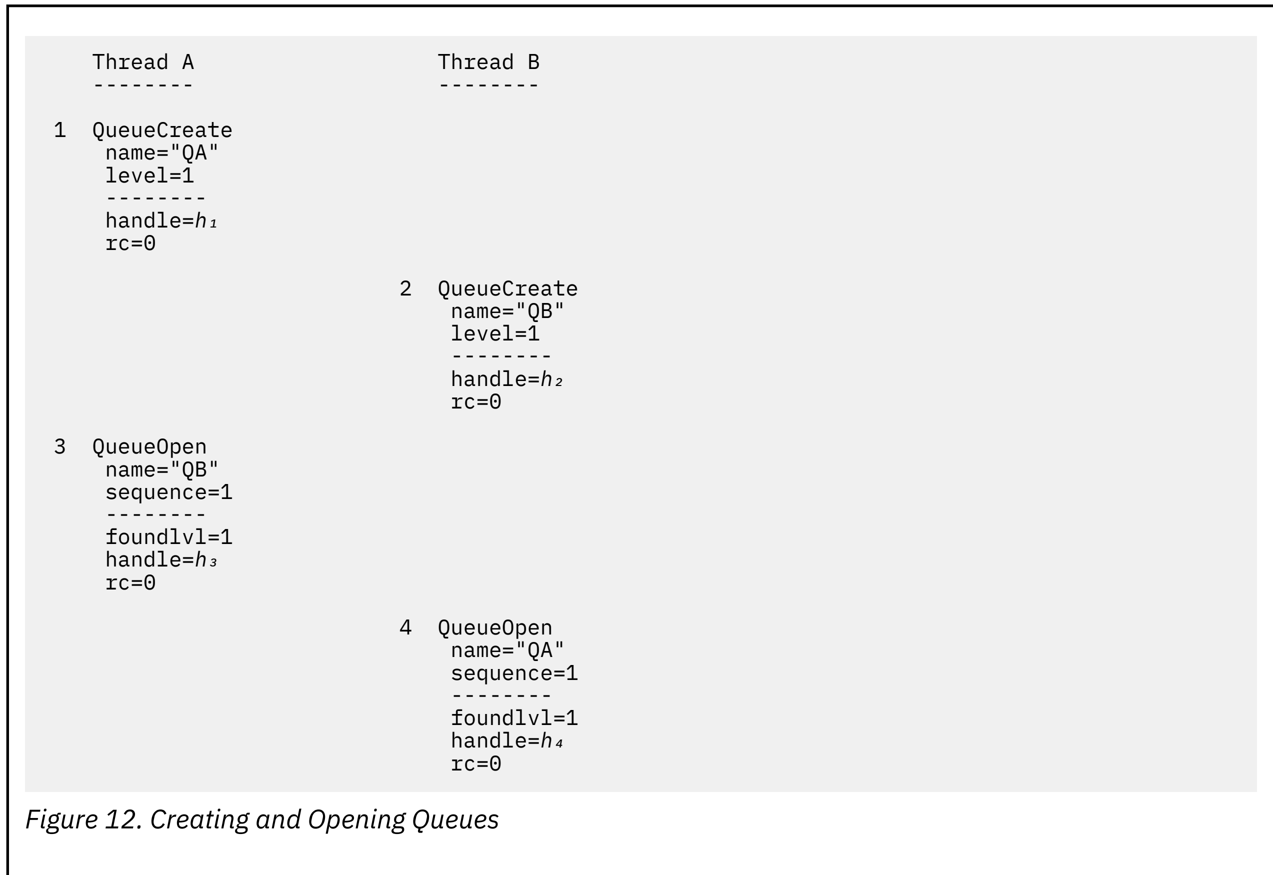
These usage notes apply to the entire Queue API:

- When CMS fills a character string buffer with variable-length return information, the following notes apply:
  - CMS always returns the length of the returned string in a separate signed 4-byte binary variable. The contents of this variable on entry to the function are not relevant.
  - CMS does not pad the returned information on the right.
- When a function fails, output variables (other than return and reason codes) are not set, unless explicitly noted otherwise.
- The interface between CMS and IPC carriers allows the carrier to specify the exact return and reason codes to be returned to the CMS application. If your program is using network queues, and if you are using a carrier other than the APPC/VM carrier provided by IBM, then the following operations may yield return and reason codes other than those documented in this book:
  - QueueOpen
  - QueueSend
  - QueueSendBlock
  - QueueSendReply
  - QueueReply
  - QueueClose.

  If one of these operations yields a return or reason code other than the ones described in this book, then you should consult the author of the carrier to determine how to proceed.

Interprocess Communication Examples

Here are several examples of how CMS interprocess communication works. The examples show different phases of IPC use: startup, message exchange, and shutdown. **Creating and Opening Queues**

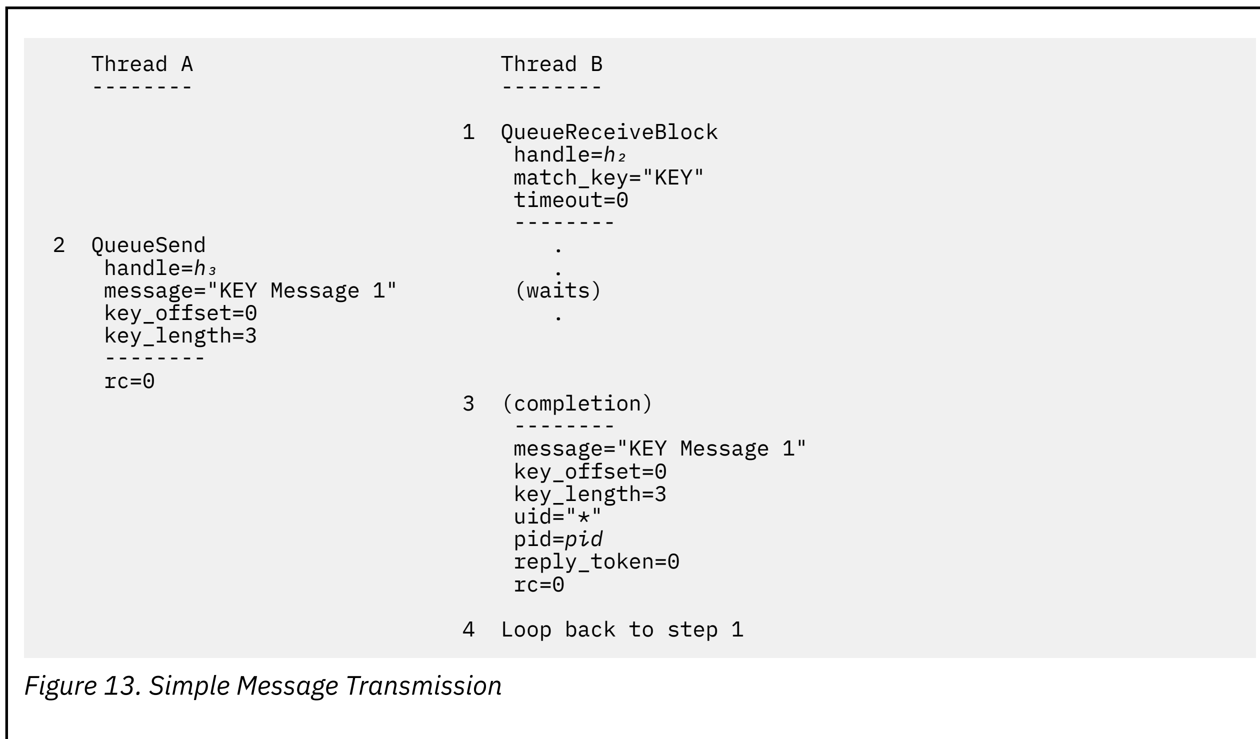


```
    Thread A                    Thread B
    --------                    --------

 1  QueueCreate
     name="QA"
     level=1
     --------
     handle=h1
     rc=0
                             2  QueueCreate
                                 name="QB"
                                 level=1
                                 --------
                                 handle=h2
                                 rc=0
 3  QueueOpen
     name="QB"
     sequence=1
     --------
     foundlvl=1
     handle=h3
     rc=0
                             4  QueueOpen
                                 name="QA"
                                 sequence=1
                                 --------
                                 foundlvl=1
                                 handle=h4
                                 rc=0
```

Figure 12. Creating and Opening Queues

Figure 12 on page 39 shows two processes creating and opening queues for peer-to-peer interaction. The steps are:

1. Thread A creates queue QA at the session level. Handle h_1 is returned to it.
2. Thread B creates queue QB at the session level. Handle h_2 is returned to it.
3. Thread A opens queue QB, specifying that only the session level is to be searched. Handle h_3 and an indication that the queue was found at the session level are returned to it.
4. Thread B opens queue QA, searching the session level. The function returns handle h_4 and an indication that the queue was found at the session level.

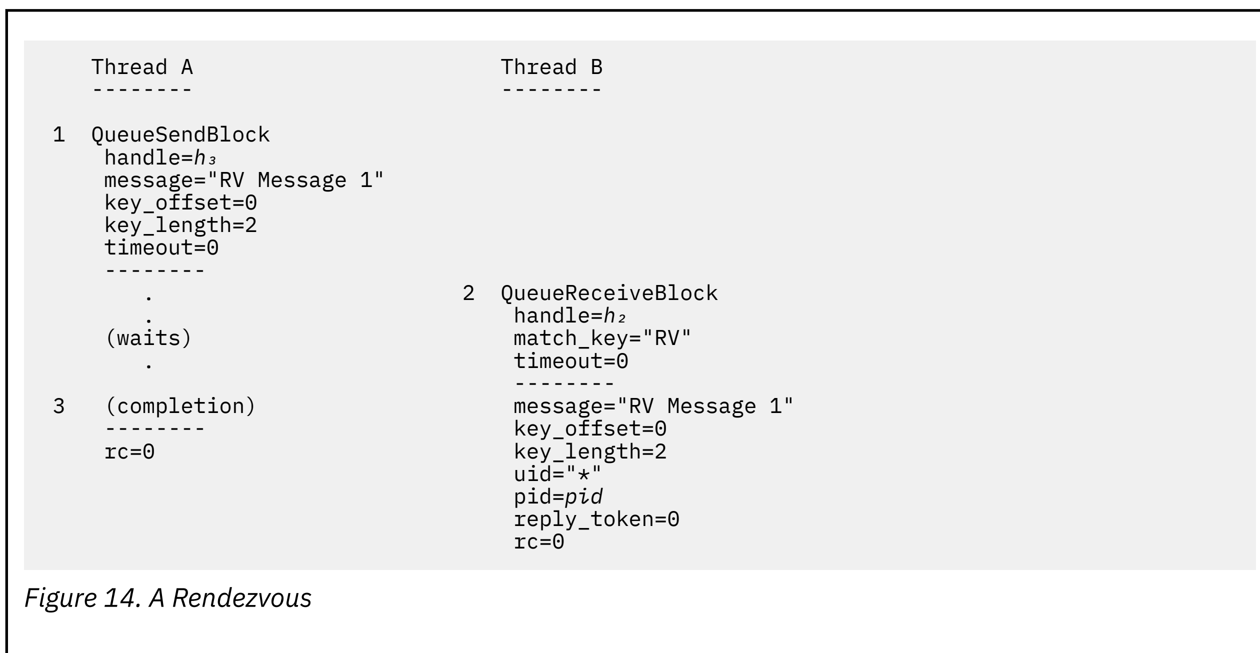


```
   Thread A                         Thread B
   --------                         --------

                                 1  QueueReceiveBlock
                                     handle=h2
                                     match_key="KEY"
                                     timeout=0
                                     --------
2  QueueSend                            .
    handle=h3                           .
    message="KEY Message 1"          (waits)
    key_offset=0                        .
    key_length=3
    --------
    rc=0
                                 3  (completion)
                                     --------
                                     message="KEY Message 1"
                                     key_offset=0
                                     key_length=3
                                     uid="*"
                                     pid=pid
                                     reply_token=0
                                     rc=0

                                 4  Loop back to step 1
```

Figure 13. Simple Message Transmission

Figure 13 on page 40 shows simple message transmission. The steps are:

1. Thread B calls QueueReceiveBlock, waiting for a message to arrive on queue QB.
2. Thread A sends a message, specifying its length and key position, to queue QB.
3. Thread B's QueueReceiveBlock completes.
4. Thread B loops back to wait on another message to arrive.

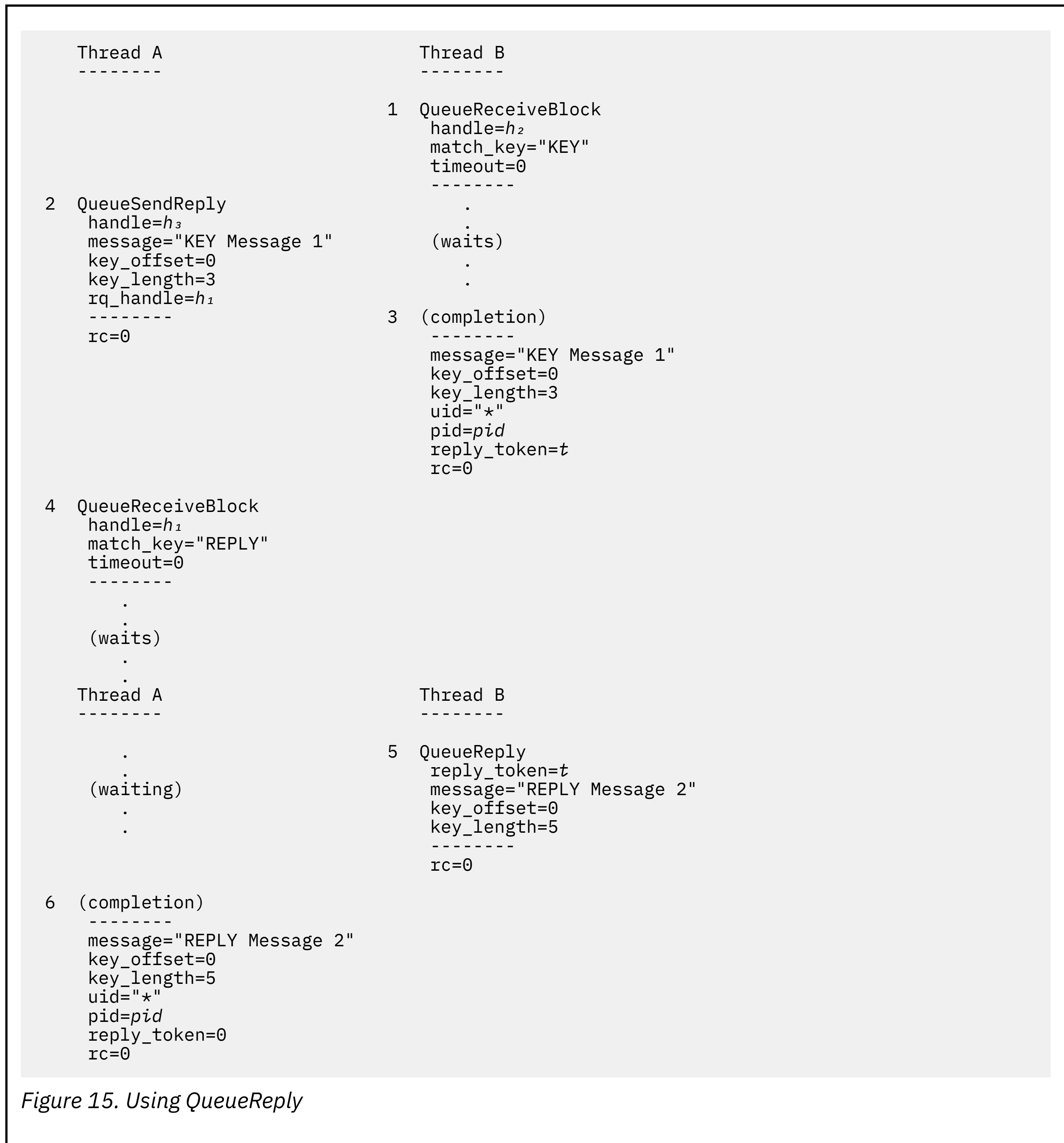


```
   Thread A                         Thread B
   --------                         --------

1  QueueSendBlock
    handle=h3
    message="RV Message 1"
    key_offset=0
    key_length=2
    timeout=0
    --------
       .                         2  QueueReceiveBlock
       .                             handle=h2
    (waits)                          match_key="RV"
       .                             timeout=0
                                     --------
3   (completion)                     message="RV Message 1"
    --------                         key_offset=0
    rc=0                             key_length=2
                                     uid="*"
                                     pid=pid
                                     reply_token=0
                                     rc=0
```

Figure 14. A Rendezvous

Figure 14 on page 40 shows the use of blocking send and blocking receive to achieve a rendezvous. This example shows only the first half of the rendezvous, because the second half is the same as the first except in reverse. The steps are:

1. Thread A sends a message to queue QB with QueueSendBlock. It uses a key indicating that the message is to be interpreted as a rendezvous request and a timeout value indicating that it wishes to wait indefinitely for the message to be received.
2. Thread B issues QueueReceiveBlock to wait for a rendezvous message. The operation completes.
3. Thread A's QueueSendBlock completes, because thread B received the rendezvous message.



```
   Thread A                        Thread B
   --------                        --------

                                1  QueueReceiveBlock
                                    handle=h2
                                    match_key="KEY"
                                    timeout=0
                                    --------
2  QueueSendReply                      .
    handle=h3                          .
    message="KEY Message 1"         (waits)
    key_offset=0                       .
    key_length=3                       .
    rq_handle=h1
    --------                    3  (completion)
    rc=0                            --------
                                    message="KEY Message 1"
                                    key_offset=0
                                    key_length=3
                                    uid="*"
                                    pid=pid
                                    reply_token=t
                                    rc=0

4  QueueReceiveBlock
    handle=h1
    match_key="REPLY"
    timeout=0
    --------
        .
        .
    (waits)
        .
        .
   Thread A                        Thread B
   --------                        --------

        .                       5  QueueReply
        .                           reply_token=t
    (waiting)                       message="REPLY Message 2"
        .                           key_offset=0
        .                           key_length=5
                                    --------
                                    rc=0

6  (completion)
    --------
    message="REPLY Message 2"
    key_offset=0
    key_length=5
    uid="*"
    pid=pid
    reply_token=0
    rc=0
```

Figure 15. Using QueueReply

Figure 15 on page 41 shows how the reply function works. The steps are:

1. Thread B issues QueueReceiveBlock against its request queue, waiting for something to arrive.
2. Thread A sends a message needing a reply. It specifies that its own queue (QA) is the place where the reply should be deposited.
3. Thread B's QueueReceiveBlock completes, the message sent by thread A being delivered.

4. Thread A waits for the reply by issuing QueueReceiveBlock, specifying the key of the expected reply.
5. Thread B responds to the request, using QueueReply and the reply token passed to it when it issued QueueReceiveBlock.
6. Thread A's QueueReceiveBlock completes, the reply having been delivered.

```
   Thread A                         Thread B
   --------                         --------

1  QueueReceiveBlock           2  QueueReceiveBlock
    handle=h1                       handle=h1
    match_key="KEY1"                match_key="KEY2"
    timeout=0                       timeout=0
    --------                        --------
       .                               .
       .                               .
    (waits)                         (waits)
       .                               .
       .                               .


         3  Message of key "KEY1" arrives
               on queue h1

4  (completion)
    --------
    message="KEY1 Message 1"
    key_offset=0
    key_length=4
    uid="*"
    pid=pid
    reply_token=0
    rc=0

         5  Message of key "KEY2" arrives
               on queue h1

                               6  (completion)
7  Processing...                    --------
                                    message="KEY2 Message 2"
                                    key_offset=0
                                    key_length=4
                                    uid="*"
                                    pid=pid
                                    reply_token=0
                                    rc=0


8  QueueReceiveBlock
    handle=h1
    match_key="KEY1"           9  QueueReceiveBlock
    timeout=0                       handle=h1
    --------                        match_key="KEY2"
       .                            timeout=0
       .                            --------
    (waits)                            .
       .                               .
       .                            (waits)
                                       .
                                       .
```

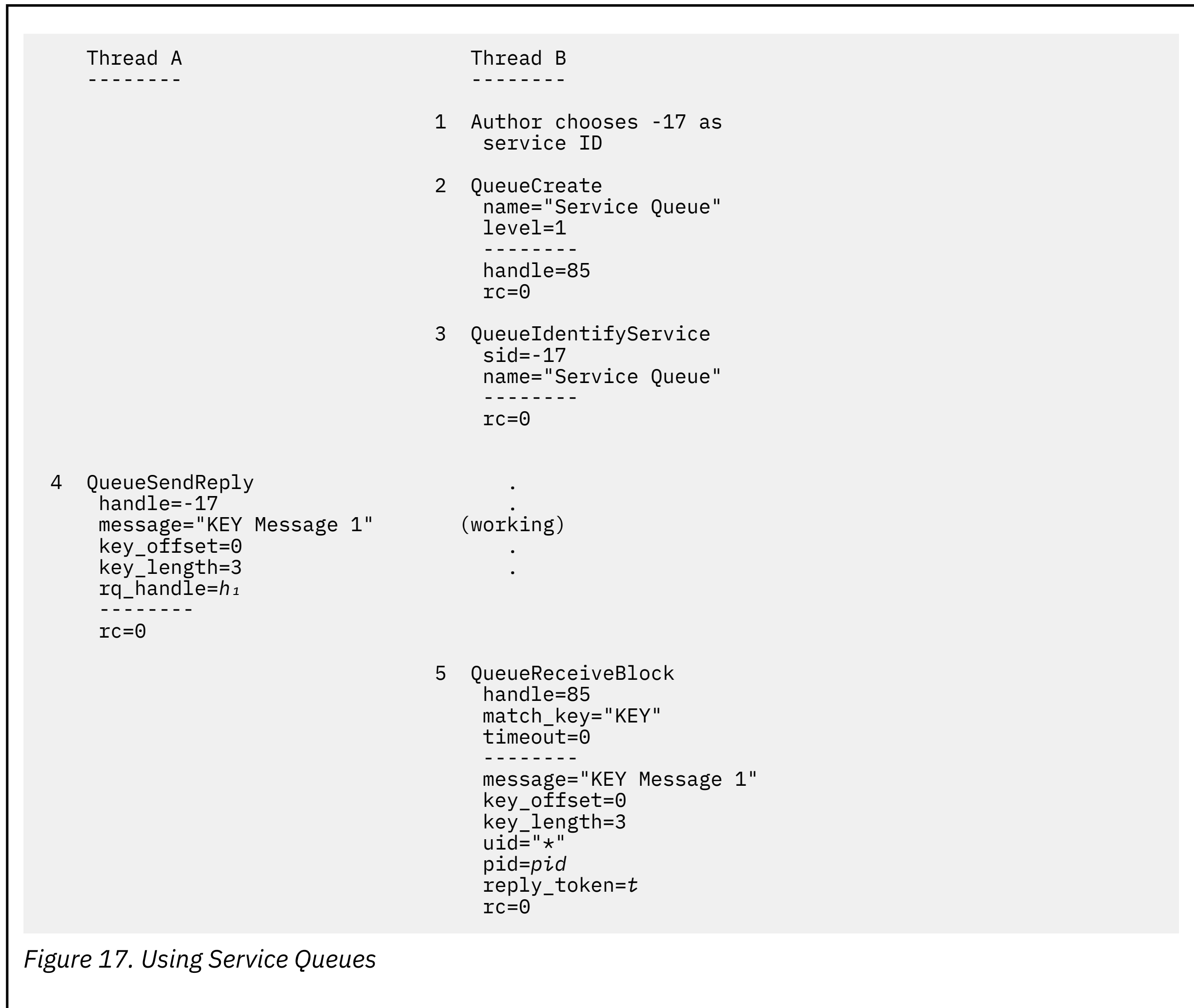
Figure 16. Two Threads Sharing a Queue

Figure 16 on page 42 shows two threads sharing a request queue. Each of the two threads is waiting on messages of a particular key to arrive (a different key for each thread). The key can be imagined to represent the request type. Each thread services a different kind of request.

The steps in the example are as follows:

1. Thread A issues QueueReceiveBlock to wait on messages having the key KEY1.
2. Thread B acts similarly, watching for the key KEY2.

3. A message satisfying thread A's wait arrives in the queue.
4. Thread A returns from QueueReceiveBlock, the matching message having been received.
5. A message satisfying thread B's wait arrives in the queue.
6. Thread B returns from QueueReceiveBlock, its message in tow.
7. Thread A is performing some processing, perhaps due to the message it received.
8. Thread A, having completed the servicing of its first message, issues another QueueReceiveBlock.
9. Thread B, also having completed the processing of its message, issues another QueueReceiveBlock.

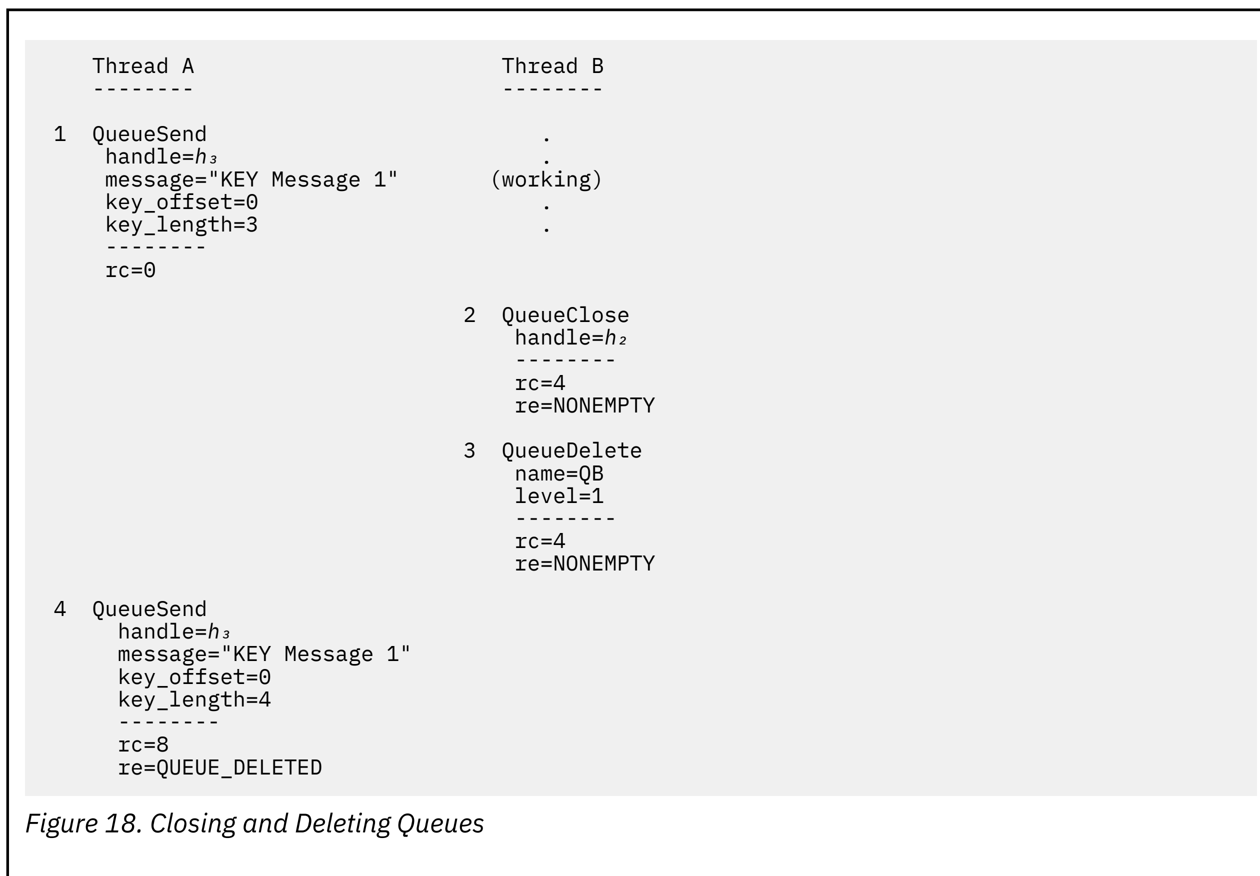


```
    Thread A                        Thread B
    --------                        --------

                                 1  Author chooses -17 as
                                     service ID

                                 2  QueueCreate
                                     name="Service Queue"
                                     level=1
                                     --------
                                     handle=85
                                     rc=0

                                 3  QueueIdentifyService
                                     sid=-17
                                     name="Service Queue"
                                     --------
                                     rc=0


 4  QueueSendReply                      .
     handle=-17                         .
     message="KEY Message 1"       (working)
     key_offset=0                       .
     key_length=3                       .
     rq_handle=h1
     --------
     rc=0

                                 5  QueueReceiveBlock
                                     handle=85
                                     match_key="KEY"
                                     timeout=0
                                     --------
                                     message="KEY Message 1"
                                     key_offset=0
                                     key_length=3
                                     uid="*"
                                     pid=pid
                                     reply_token=t
                                     rc=0
```

Figure 17. Using Service Queues

Service queues are exploited through the QueueIdentifyService function. A service thread author chooses a service ID in advance and associates a queue with the service ID by calling QueueIdentifyService. Clients can then send messages to the server by using the service ID as the queue handle in a call to QueueSendReply. Figure 17 on page 43 shows an example of this and is explained as follows:

1. The service thread author chooses -17 as his thread's service ID.
2. The service thread creates the service queue at session scope.
3. The service thread calls QueueIdentifyService to associate the service queue with the service ID.
4. The client uses QueueSendReply to send a message to the server. Note that the client did not have to open the service queue (it does not even know the name of the service queue).
5. The server receives the message with QueueReceiveBlock.



```
    Thread A                        Thread B
    --------                        --------

 1  QueueSend                           .
     handle=h3                          .
     message="KEY Message 1"        (working)
     key_offset=0                       .
     key_length=3                       .
     --------
     rc=0

                                 2  QueueClose
                                     handle=h2
                                     --------
                                     rc=4
                                     re=NONEMPTY

                                 3  QueueDelete
                                     name=QB
                                     level=1
                                     --------
                                     rc=4
                                     re=NONEMPTY

 4  QueueSend
      handle=h3
      message="KEY Message 1"
      key_offset=0
      key_length=4
      --------
      rc=8
      re=QUEUE_DELETED
```

Figure 18. Closing and Deleting Queues

To finish using a queue, a thread uses the QueueClose function. If its process created the queue and wishes to destroy it, it may also use the QueueDelete function. Other threads using the queue learn of the deletion the next time they attempt to use the queue. Figure 18 on page 44 shows an example of this and is explained as follows:

1. Thread A sends a message to QB.
2. Thread B closes queue QB. The queue still exists, and thread B may even reopen it if it chooses.
3. Thread B deletes queue QB. It receives a return and reason code indicating that messages were in the queue, and the deletion proceeds.
4. On its next attempt to use the queue, thread A learns that the queue was deleted.

Setting Up Network-Level Queues

The following example shows how to set up for network-level queues using the APPC/VM line driver. Some of the setup has to do with queues themselves, and some of it has to do with CMS's use of APPC/VM to implement distributed queues.

In this discussion:

- The word "server" refers to the virtual machine that creates and reads the queue, and the word "client" refers to the virtual machine that opens and sends to the queue.
- Assume the server's user ID is SERVER and the client's user ID is CLIENT.
- Assume that SERVER and CLIENT are user IDs on the same node. (CMS can handle network queues where CLIENT and SERVER are on different VM systems, but that complicates the discussion here. Once you understand how to do this inside one system, you can learn to adjust the configuration so that it works across two systems.)
- Assume that the program running in SERVER is called SPROG.

- Assume that the queue created by SPROG is called SQUEUE (that is, SPROG uses name SQUEUE in its call to QueueCreate).
- Assume that the program running in CLIENT is called CPROG.
- Assume that CPROG will attempt to open SQUEUE by using name CQUEUE in its call to QueueOpen.

When you apply the following instructions to your own situation, substitute appropriate user IDs, program names, queue names, and so on. *Remember that queue names are case-sensitive.*

CLIENT Setup

The basic idea in setting up the client is to inform CMS of the existence of SQUEUE. CMS has to know how to take the token "CQUEUE" and use it to determine the route to SQUEUE. Here is what you have to do on CLIENT:

1. Create a file called $QUEUES$ NAMES and put the following entry in it:

   ```
   :nick.CQUEUE
    :scope.NETWORK
    :qn.SQUEUE
    :carrier.APPC/VM
    :sdn.SCDE
   ```

   This entry indicates that CQUEUE is a network-level queue and that:

   - On the server side, it is known as SQUEUE.
   - The APPC/VM line driver should be used to reach SQUEUE.
   - The symbolic destination name that indexes the CMS Communication Directory (ComDir) entry for SERVER is SCDE.
2. Create a user-level ComDir file (for example, UCOMDIR NAMES) and put the following entry in it:

   ```
   :nick.SCDE
    :tpn.VMIPC
    :luname.*USERID SERVER
    :security.SAME
   ```

   This entry tells CMS's APPC/VM line driver that, to establish an APPC/VM conversation to SCDE, it must connect to transaction program VMIPC at the LU named *USERID SERVER. "VMIPC" is the transaction program name for the CMS APPC/VM network queue line driver. (The meanings of the rest of the tags used in the ComDir entry are outside the scope of this discussion.)
3. Enter the following CMS commands:

   ```
   set comdir file user ucomdir names *
   set comdir reload
   ```

   These commands cause CMS to reread $QUEUES$ NAMES and UCOMDIR NAMES, picking up the entries you just put there.

SERVER Setup

The basic idea in setting up the server is to inform CP that other virtual machines are permitted to connect to SERVER through APPC/VM, and to inform CMS of what to do when some other virtual machine attempts to communicate with it over the APPC/VM network queue line driver. Here is what you have to do:

1. Make sure that the CP directory entry for SERVER includes an IUCV ALLOW record. This permits other virtual machines to connect to SERVER through APPC/VM.
2. In the $SERVER$ NAMES file on SERVER's A-disk, add the following entry:

   ```
   :nick.VMIPC      :module.SPROG     :list.CLIENT
   ```

This entry indicates that only a user whose user ID is CLIENT is allowed to connect to transaction program VMIPC. It also maps VMIPC to your program SPROG (the reason for this mapping is outside the scope of this discussion).

Server API Calls

On SERVER, it is necessary to create SQUEUE at network scope. The following REXX fragment accomplishes this:

```
/* load API definitions */
call apiload(vmrexmtr)
call apiload(vmrexipc)

/* set up constants for QueueCreate */
qn = 'SQUEUE'
qnl = length(qn)

/* create queue */
call csl 'QueueCreate mtrc mtre qn qnl vm_ipc_nlevel qh'
say 'RC='mtrc 'RE='mtre 'creating queue' qn
```

Note that the queue's handle comes back in variable *qh*; you would use *qh* in a subsequent call to QueueReceiveBlock.

Client API Calls

On CLIENT, it is necessary to issue a QueueOpen call. The following REXX fragment should do this, if SPROG is already running and has already issued its QueueCreate:

```
/* load API definitions */
call apiload(vmrexmtr)
call apiload(vmrexipc)

/* set up constants for QueueOpen */
qn = "CQUEUE"
qnl = length(qn)
sv.1 = vm_ipc_nlevel
sv.2 = 0
sv.3 = 0
svl = 1

/* try to open it */
call csl 'QueueOpen mtrc mtre qn qnl sv svl qh el'
say 'RC='mtrc 'RE='mtre 'from QueueOpen of' qn
```

Testing the Queues

You may want to begin by manually starting SPROG and seeing that its QueueCreate call works. From there, we assume that SPROG enters QueueReceiveBlock, waiting for a message.

Then start CPROG. You should see that its call to QueueOpen works. Next, issue QueueSend. You should see the message show up in SQUEUE, and SPROG's call to QueueReceiveBlock will complete.

From this simple example you can grow to a situation where, in response to CPROG's QueueOpen, CP autologs SERVER and CMS automatically starts SPROG. SPROG would do its QueueCreate, and shortly after that, CPROG's QueueOpen would complete.

Chapter 5. Synchronization

Threads within a single process share all resources owned by the process. This implies that threads must coordinate access to these resources by implementing serialization and mutual exclusion policies.

The synchronization functions provided by CMS allow sets of threads to implement coordination and mutual exclusion policies. They are primitives upon which more elaborate and special purpose synchronization mechanisms can be built by programming languages and applications. These primitives are designed to be highly efficient in a parallel execution environment.

The synchronization primitives are based on the following definitions:

Critical Section
A block of code that manipulates a shared resource, such as a data structure or device.

Mutex
A variable with an associated wait queue used to protect a critical section. The typical use of a mutex is as follows:

```
MutexAcquire(mutex_handle)

   /* access the critical section */

MutexRelease(mutex_handle)
```

If more than one thread simultaneously tries to acquire the mutex, only one such thread acquires the mutex. The other threads are queued, waiting to acquire the mutex, and are given the mutex one at a time as the previous holder of the mutex releases it. The order in which these threads acquire the mutex is defined by the order in which they execute the MutexAcquire function.

Condition Variable
A variable representing a state for a mutex-protected shared resource. This state, or condition, can be waited on and signaled as being true.

CMS maintains the condition variable and its associated wait queue. However, it does not determine if the condition is true or false. It is the responsibility of the application to wait on the condition variable when necessary and signal it when the condition is true.

For example, if a stack of length 10 is a shared resource, a condition variable might represent whether there are less than 10 elements in the stack. An element can be pushed onto the stack if it currently contains less than 10 elements.

To use a condition variable, a thread evaluates the condition represented by the condition variable while holding its associated mutex. If the condition is false, the CondVarWait function is issued. When the thread returns from this function, it once again holds the mutex and the condition is true. The other side of the condition variable interface is the CondVarSignal function. It is used by a thread that has changed the state of the shared resource to make the condition true for one thread that may be waiting.

Semaphore
A variable used to perform wait and post operations between threads. It is an integer variable S with an associated wait queue upon which only the following two primary atomic (that is, noninterruptable) operations may be performed:

```
P(S): S := S - 1;
      If (S < 0) then
         Add the thread to the wait queue for S;
      Endif

V(S): S := S + 1;
      If (S ≤ 0) then
         Unblock a thread on wait queue for S;
      Endif
```

© Copyright IBM Corp. 1992, 2023

Note here that S may be interpreted as follows: if S ≥ 0, the number of P(S) operations that are allowed before blocking occurs; if S < 0, the number of blocked threads waiting on S.

The P(S) and V(S) operations are renamed to the more meaningful function names SemWait and SemSignal, respectively. These functions implement general counting semaphores.

Along with the primary semaphore operations defined above, a SemReInit function is included. This function reinitializes the semaphore's value and unblocks all the threads waiting on a semaphore.

There is no notion of establishing ownership of a semaphore as is the case with a mutex. A semaphore should be used to wait for a condition to occur and resume execution when the condition occurs.

The central operational differences between a mutex and a semaphore are the preconditions placed on the operations defined on them. A given thread can signal a semaphore without having previously waited on it, but a given thread cannot release a mutex without having previously acquired it. See Chapter 13, "CMS Multitasking Function Descriptions," on page 95 for a more formal definition of these functions.

Because mutexes and condition variables provide more discipline, semaphores should be used only when these more structured mechanisms cannot be applied. See "Synchronization Examples" on page 48 for examples on the use of mutexes and semaphores.

Event services can alert other threads of important situations, such as the end of a computation phase or an error condition. Signaling, testing, and waiting for events can also be used in place of semaphores; however, semaphore operations are more efficient for such situations.

When a program creates a mutex or semaphore, it assigns it a scope. This scope determines the visibility of the mutex or semaphore relative to other threads in the session. Process-level scope means that the mutex or semaphore is accessible only to the threads in the creating process. Session-level scope means that any thread in any process in the session can access the mutex or semaphore. A condition variable has visibility equal to the visibility of the mutex with which it is associated.

Mutexes, semaphores, and condition variables are assigned handles when they are created. A handle is simply a token used in subsequent function calls to identify the object. The scope of a handle is equal to the scope of the object with which it is associated; in other words, the handle may be used to identify the object wherever the object is known. The scope of the handle is a useful concept. For example, the handle of a mutex can be stored in the data structure it is used to protect and thus be accessible to all threads that use the data structure.

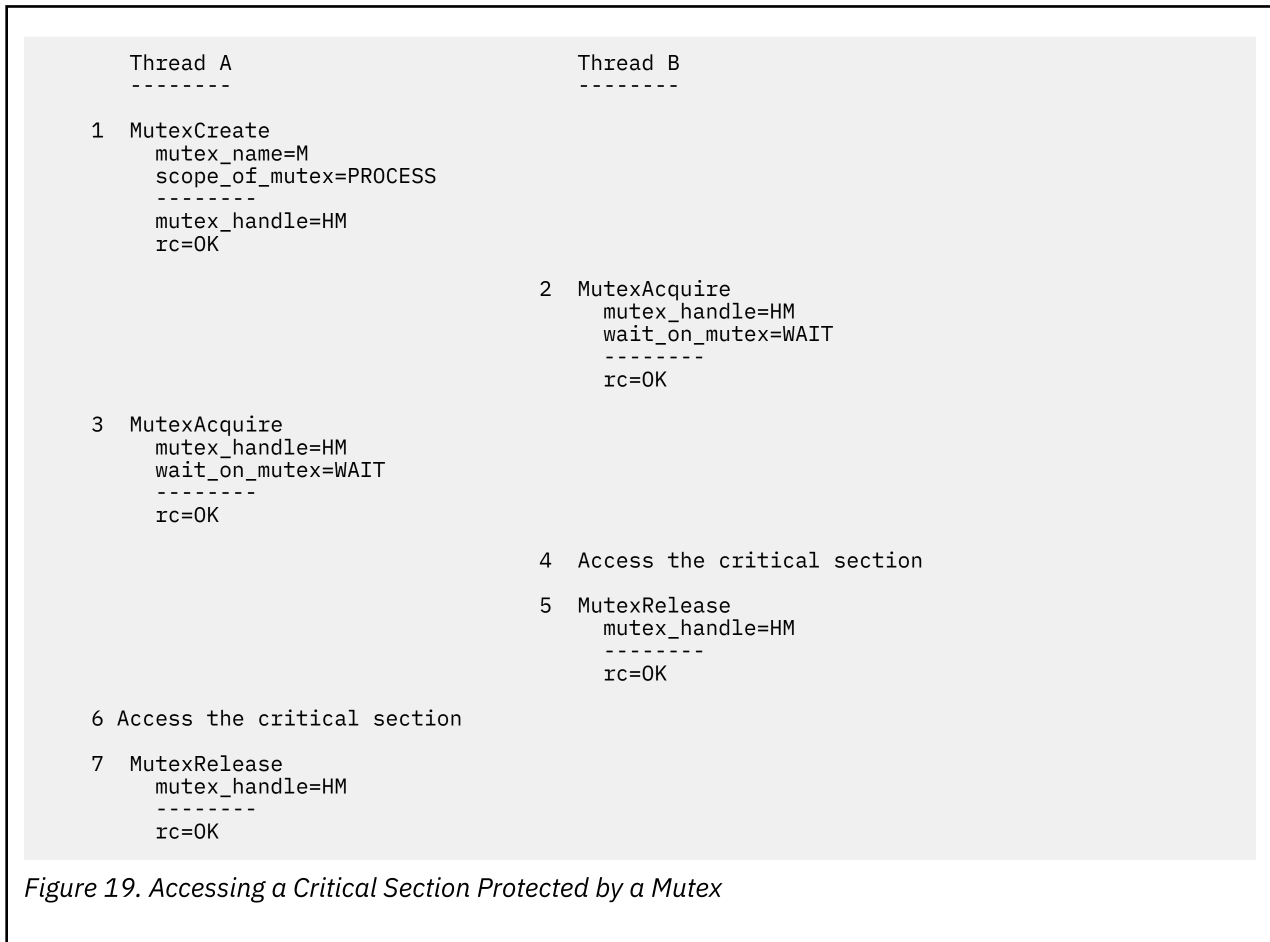
CMS supports up to 32,768 session-scope semaphores, mutexes, and condition variables, altogether. Also, for each process, CMS supports up to 32,768 process-scope semaphores, mutexes, and condition variables, altogether.

Synchronization Examples

This section shows several examples of the synchronization mechanisms available in the CMS environment. The examples include the following:

- Accessing a critical section protected by a mutex
- Replacing OS Wait/Post macros with a semaphore
- Going beyond Wait/Post using semaphores
- Producer/Consumer example using mutexes and condition variables.

Accessing a Critical Section Protected by a Mutex



```
   Thread A                           Thread B
   --------                           --------

1  MutexCreate
     mutex_name=M
     scope_of_mutex=PROCESS
     --------
     mutex_handle=HM
     rc=OK

                                   2  MutexAcquire
                                        mutex_handle=HM
                                        wait_on_mutex=WAIT
                                        --------
                                        rc=OK

3  MutexAcquire
     mutex_handle=HM
     wait_on_mutex=WAIT
     --------
     rc=OK

                                   4  Access the critical section

                                   5  MutexRelease
                                        mutex_handle=HM
                                        --------
                                        rc=OK

6 Access the critical section

7  MutexRelease
     mutex_handle=HM
     --------
     rc=OK
```

Figure 19. Accessing a Critical Section Protected by a Mutex

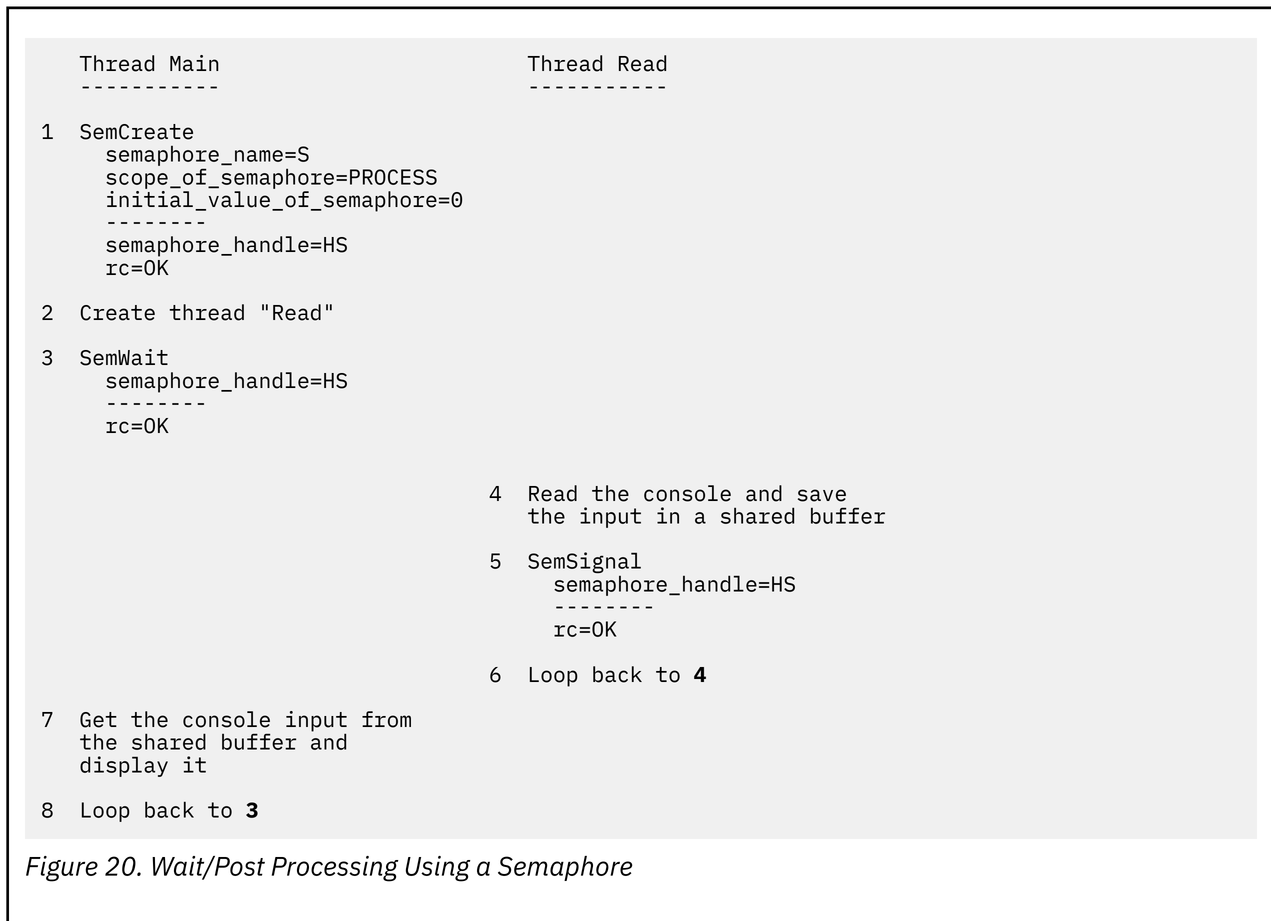
Figure 19 on page 49 shows two threads in a process accessing a critical section protected by a mutex.

The steps are:

1. Thread A creates mutex M with process level scope and saves the handle of the mutex in order that thread B can also use it.
2. Thread B attempts to acquire the mutex to get access to the critical section and is successful.
3. Thread A tries to acquire the mutex to get access to the critical section and has to wait because Thread B already holds the mutex.
4. Thread B accesses the critical section.
5. Having completed accessing the critical section, Thread B releases the mutex.
6. With the mutex now available, Thread A acquires the mutex and accesses the critical section.
7. Having completed accessing the critical section, Thread A releases the mutex.

Basic Semaphore Processing

Semaphores may also be used to protect a critical section. The changes required in Figure 19 on page 49 entail replacing MutexAcquire with SemWait and MutexRelease with SemSignal. However, the responsibility of protecting the critical section is solely upon the application because of the preconditions placed on the operations defined by a semaphore. CMS does not prevent a thread from signaling a semaphore without first having waited on it. Mutexes do not allow such behavior.



```
   Thread Main                        Thread Read
   -----------                        -----------

1  SemCreate
     semaphore_name=S
     scope_of_semaphore=PROCESS
     initial_value_of_semaphore=0
     --------
     semaphore_handle=HS
     rc=OK

2  Create thread "Read"

3  SemWait
     semaphore_handle=HS
     --------
     rc=OK

                                   4  Read the console and save
                                      the input in a shared buffer

                                   5  SemSignal
                                        semaphore_handle=HS
                                        --------
                                        rc=OK

                                   6  Loop back to 4

7  Get the console input from
   the shared buffer and
   display it

8  Loop back to 3
```

Figure 20. Wait/Post Processing Using a Semaphore

Semaphores are more suited to waiting for a condition to occur and resuming execution after the condition occurs. The next example describes the intended use of a semaphore.

Figure 20 on page 50 shows two threads in a process using a semaphore to do wait/post synchronization.

The steps are:

1. Thread Main creates semaphore S with process level scope, the initial value for use as a wait/post mechanism, and saves the handle of the semaphore in order that thread Read can also use it.
2. Thread Main creates Thread Read to read the console.
3. Thread Main issues the ***SemWait*** function which decrements the value of semaphore S by 1. Because the value of semaphore S is now -1, Thread Main is put on the queue for semaphore S.
4. Thread Read waits until something is entered from the console. The console input is saved in a shared buffer.
5. After reading the console, Thread Read signals semaphore S, unblocking Thread Main.
6. Thread Read loops back and waits for something else to be entered from the console.
7. Thread Main gets the console input from the shared buffer and displays it.
8. Thread Main loops back and waits for another line to be read from the console.

Going Beyond WAIT/POST Using Semaphores

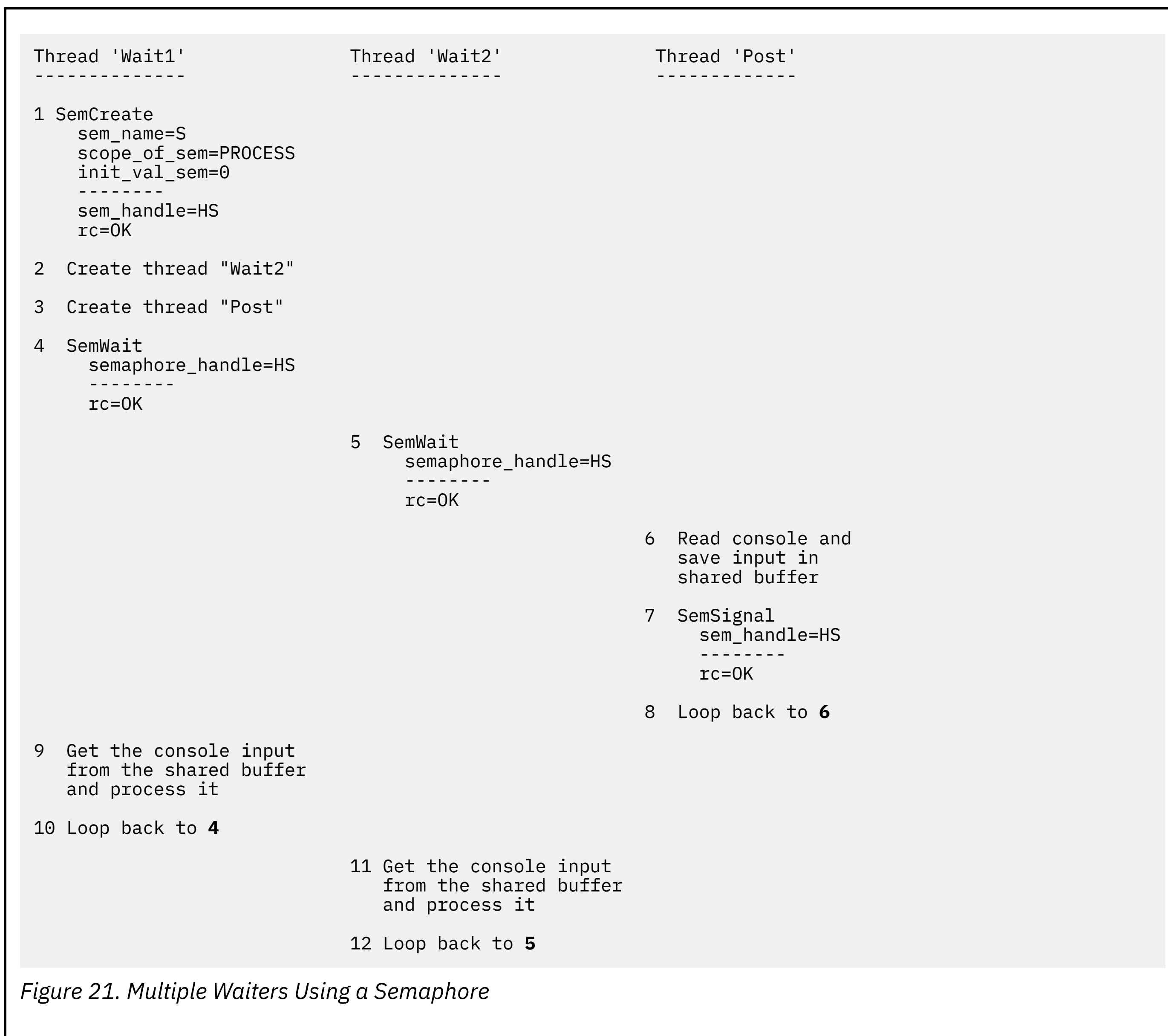
Multiple Waiters

This example shows one of the functions available with a semaphore beyond that available with MVS Wait and Post. ECBs permit only one waiter at a time, whereas semaphores may have multiple threads waiting concurrently.

Figure 21 on page 52 shows three threads in a process doing Wait/Post processing using a Semaphore.

The steps are:

1. Thread Wait1 creates semaphore S with process level scope, the initial value for use as a wait/post mechanism, and saves the handle of the semaphore in order that the other threads can use it.
2. Thread Wait1 creates Thread Wait2 that also waits for the console to be read.
3. Thread Wait1 creates Thread Post to read the console.
4. Thread Wait1 issues the SemWait function, which decrements the value of semaphore S by 1. Because the value of semaphore S is now -1, Thread Wait1 is put on the queue for semaphore S.
5. Thread Wait2 issues the SemWait function, which decrements the value of semaphore S by 1. Because the value of semaphore S is now -2, Thread Wait2 is put on the queue for semaphore S.
6. Thread Post waits until something is entered from the console. The console input is saved in a shared buffer.
7. After reading the console, Thread Post signals semaphore S, unblocking Thread Wait1.
8. Thread Post loops back and waits for something else to be entered from the console.
9. Thread Wait1 gets the console input from the shared buffer and processes it.
10. Thread Wait1 loops back and waits for another line to be read from console.
11. Thread Post reads the console and signals semaphore S, unblocking Thread Wait2. Thread Wait2 now gets the console input from the shared buffer and processes it.
12. Thread Wait2 loops back and waits for another line to be read from console.



```
Thread 'Wait1'              Thread 'Wait2'              Thread 'Post'
--------------              --------------              -------------

1 SemCreate
    sem_name=S
    scope_of_sem=PROCESS
    init_val_sem=0
    --------
    sem_handle=HS
    rc=OK

2  Create thread "Wait2"

3  Create thread "Post"

4  SemWait
     semaphore_handle=HS
     --------
     rc=OK

                             5  SemWait
                                  semaphore_handle=HS
                                  --------
                                  rc=OK

                                                           6  Read console and
                                                              save input in
                                                              shared buffer

                                                           7  SemSignal
                                                                sem_handle=HS
                                                                --------
                                                                rc=OK

                                                           8  Loop back to 6

9  Get the console input
   from the shared buffer
   and process it

10 Loop back to 4

                             11 Get the console input
                                from the shared buffer
                                and process it

                             12 Loop back to 5
```

Figure 21. Multiple Waiters Using a Semaphore

Producer and Consumer Processes

The following programming example demonstrates the use of some of the mutex and condition variable functions in solving the bounded buffer problem. This is accomplished by implementing a monitor with these functions.

In Figure 22 on page 53, a set of producer threads supply messages to a set of consumer threads.

```
Producer: Procedure
    .
    .
Repeat
     Create a new message

     /* Add the new message to the buffer */
     Call Write(message)
     .
Until DONE


Consumer: Procedure
     .
     .
Repeat
      /* Consume a message from the buffer */
      Call Read(message)
      .
Until DONE
```

Figure 22. Producer Threads Supply Messages to Consumer Threads

Shared Monitor

The following synchronization mechanisms are required:

- A mutex, ***M***, to control access to shared circular buffer.
- Two condition variables are needed. Condition variable ***C1*** represents the *buffer is not full* condition while condition variable ***C2*** represents the *buffer is not empty* condition.

Also, the following variables are required:

N
: Number of messages in the shared circular buffer

Read_buffer_index
: Index into the shared circular buffer used to determine where to read a message from.

Write_buffer_index
: Index into the shared circular buffer used to determine where to write a message.

```
Monitor_Init: Procedure
     .
     MutexCreate
       mutex_name=M
       scope_of_mutex=PROCESS
       --------
       mutex_handle=HM
       rc=OK
     CondVarCreate
       condvariable_name=C1
       mutex_name=M
       --------
       condvariable_handle=HC1
       rc=OK
     CondVarCreate
       condvariable_name=C2
       mutex_name=M
       --------
       condvariable_handle=HC2
       rc=OK
     .
End Monitor_Init
```

Figure 23. Monitor Initialization

```
Write: Entry(bufelement)
     /* Gain Access to buffer */
     MutexAcquire
       mutex_handle=HM
       wait_on_mutex=WAIT
       --------
       rc=OK

     /* Is there any space in the buffer to add a message */
     do while n=0

        /* Wait for a space to become available */
        CondVarWait
          condvariable_handle=HC1
          --------
          rc=OK
        /* The loop assures that the buffer has not been refilled */

     end
     /* Add the message to the buffer */
     Buffer(write_buffer_index)=message

     /* Update write_buffer_index and number of messages in buffer */
     write_buffer_index=(write_buffer_index + 1) MOD 10
     n=n+1

     /* Signal that a message has been placed in the buffer */
     CondVarSignal
       condvariable_handle=HC2
       --------
       rc=OK

     /* Give up access to buffer */
     MutexRelease
       mutex_handle=HM
       --------
       rc=OK
End Write
```

Figure 24. Monitor Write Procedure

```
Read: Entry(bufelement)

     /* Gain Access to buffer */
     MutexAcquire
       mutex_handle=HM
       wait_on_mutex=WAIT
       --------
       rc=OK

     /* Are there any messages in the buffer */
     do while n=0

        /* Wait for a message to be placed in the buffer */
        CondVarWait
          condvariable_handle=HC2
          --------
          rc=OK
        /* The loop assures that the buffer has not been emptied */

     end
     /* Consume a message from the buffer */
     message=Buffer(read_buffer_index)

     /* Update read_buffer_index and number of message in buffer
     read_buffer_index=(read_buffer_index + 1) MOD 10
     n=n-1

     /* Signal that a message has been consumed from the buffer */
     CondVarSignal
       condvariable_handle=HC1
       --------
       rc=OK

     /* Give up access to buffer */
     MutexRelease
       mutex_handle=HM
       --------
       rc=OK

End Read
```

Figure 25. Monitor Read Procedure

Chapter 6. Multiprocessor Configuration Control

Many service virtual machines and some computationally intensive applications require **parallel processing** to cope with heavy processing loads. By parallel processing we mean the execution of more than one thread of a particular application at the same time on different real CPUs of the real processor complex. For example, a numeric application may process each column of a table in parallel, if the computations performed on the rows are independent of each other. A different type of example involves a multi-user server like a file server or a database manager. These servers may become saturated with requests. By taking advantage of parallelism they can potentially handle multiple requests at the same time. In general, the goals of parallel processing are to reduce the time a computation takes, increase the capacity of a server, or improve the responsiveness of a server.

CMS provides parallel processing capabilities through the use of CP's virtual multiprocessor support. The CMS virtual machine running the CMS-based application can have more than one virtual CPU, each of which will be used to execute threads. These virtual CPUs, in turn, are dispatched independently by CP. CP can have any one or all of the CPUs for a particular virtual machine in execution at a given time.

The use of virtual CPUs is controlled by CP information and command input. The user directory entry determines how many virtual CPUs can be defined and how many are to be predefined for a particular virtual machine. In the example directory entry, the number given on the MACHINE statement specifies that 64 virtual CPUs can be used by this virtual machine. The CPU statements specify that six of these CPUs should be automatically defined when the user logs on.

```
USER SERVER1 XXXXXXX 5M 048M G 64
MACHINE XC 64
CPU 0 BASE
CPU 1
CPU 2
CPU 3
CPU 4
CPU 5
ACCOUNT 0372 G37/272
IPL CMS  PARM AUTOCR
CONSOLE  01F 3215
SPOOL    00C 2540 READER A
SPOOL    00D 2540 PUNCH A
SPOOL    00E 1403
LINK $MAINT 190 190 RR
LINK CMSGEN 19F 19F RR
      .
      .
      .
MDISK 0191 3390 300 10 CMS888 MR
```

If the application requires more CPUs than those automatically defined, it can use the VCPUCreate function to dynamically define additional virtual CPUs (up to the limit specified in the directory) while the application is running. As soon as the CPU is defined, CMS begins dispatching threads on it within the constraints imposed by dispatching classes. See "Dispatching Classes" on page 11 for an explanation of thread dispatching classes and their implications. Essentially, the number of CPUs that can be used is equal to the number of thread dispatch classes the application has created. If all threads are assigned to different classes, all the CPUs can be exploited to the fullest.

Guidelines for Defining Virtual CPUs

1. After the first multitasking program has started, any additional virtual CPUs required must be defined by means of the VCPUCreate function, not by the CP DEFINE CPU command. CPUs defined using this command after this point will be ignored.
2. For most parallel applications, the number of virtual CPUs should be equal to the number of real CPUs in the real processor complex. However, an application whose threads make use of synchronous CP facilities, such as DIAGNOSE instructions, can efficiently make use of more CPUs. This is because most

© Copyright IBM Corp. 1992, 2023

synchronous CP operations serialize only the issuing CPU. An application may have multiple CPUs in CP-wait while other CPUs are still runable.

3. CP divides a user's SHARE setting evenly among its CPU's. For some high-use servers, the SHARE setting may need to be increased to allow multiprocessor advantages to be felt.
4. Parallel FORTRAN programs must be run before any other multitasking programs are run. FORTRAN run-time support will take over the virtual CPU's for it's own use. If CMS has already run a multitasking program it will have already begun using all the virtual CPU's to dispatch threads.

Chapter 7. Timer Services

CMS provides a comprehensive, callable timer facility. This facility allows timers to be started, stopped and interrogated. Additionally, Timer Services defines the expiration of a timer as an occurrence of the VMTIMER event. Thus, the full power of Event Management may be used in processing the VMTIMER event, as well as combining its processing with other events defined by CMS or an application. The timer functions are:

- DateTimeGet — Return the date, time, time zone, and epoch time.
- DateTimeSubtract — Convert the format or time zone of a time stamp or perform arithmetic on time stamps.
- TimerStartTOD — Start a time of day (TOD) timer.
- TimerStartInt — Start an interval timer based on CPU time or real time. The timer can be single, meaning that it expires after a specified time interval; or it can be cyclical, meaning that it continues to expire at regular intervals until specifically stopped. The interval is 4 bytes in length and may be specified in milliseconds or microseconds.
- TimerStartMicros — Start an interval timer based on CPU time or real time. The timer can be single or cyclical, as above. The interval is 8 bytes in length and is specified in microseconds.
- TimerStop — Cancel a timer previously started by TimerStartInt or TimerStartTOD and return the time remaining.
- TimerStopMicros — Cancel a timer previously started by TimerStartMicros and return the time remaining.
- TimerStopAll — Cancel all previously started timers.
- TimerTest — Return information concerning a timer started by TimerStartInt or TimerStartTOD.
- TimerTestMicros — Return information concerning a timer started by TimerStartMicros.

The MVS simulated timer services, STIMER, STIMERM, TTIMER, and TIME can be used in conjunction with these timer services. An application can start a timer and monitor the VMTIMER event while also using (or invoking other programs that use) the MVS timer services.

Only one active STIMER exit is allowed per simulated MVS task. Simulated MVS task management services do not exploit the native CMS multitasking facilities. The MVS ATTACH service is simulated as a LINK plus the creation of a simulated Task Control Block. They are not related to CMS dispatchable threads of control, but are kept as a single stack of MVS task levels.

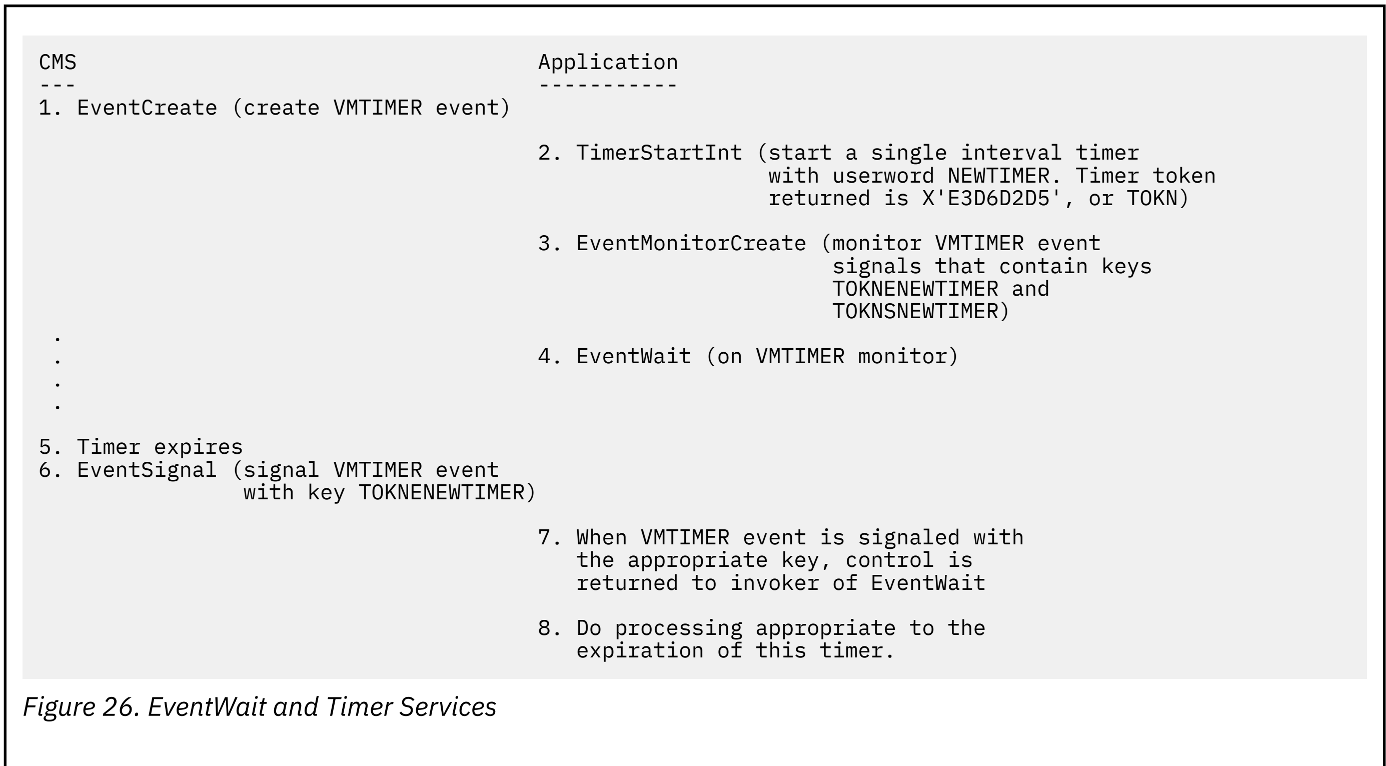
The VMTIMECHANGE system event allows multitasking applications to monitor time zone changes. The signal data associated with VMTIMECHANGE includes:

- offset in seconds from GMT in the time zone you left
- offset in seconds from GMT in the time zone you are presently in.

Timer Services Examples

The comprehensive timer facility of CMS is provided by the interaction of Timer Services and Event Management. During CMS initialization, Timer Services creates the VMTIMER event. Each time a timer expires or is stopped, Timer Services signals the VMTIMER event with a key consisting of the timer token concatenated with either an E (for expiring timers) or an S (for stopped timers) concatenated with the userword specified when the timer was started. For example, the signal key for a stopped timer assigned token X'00000001' and started with userword NEWTIMER, which is X'D5C5E6E3C9D4C5D9', would be X'00000001E2D5C5E6E3C9D4C5D9'. If a userword of all binary zeros is specified, the userword is not included in the signal key when Timer Services signals this VMTIMER event. By using Event Management to monitor and process signals of the VMTIMER event, the application may request notification of a timer expiration. A simple example of this follows.

© Copyright IBM Corp. 1992, 2023



```
CMS                                      Application
---                                      -----------
1. EventCreate (create VMTIMER event)

                                         2. TimerStartInt (start a single interval timer
                                                          with userword NEWTIMER. Timer token
                                                          returned is X'E3D6D2D5', or TOKN)

                                         3. EventMonitorCreate (monitor VMTIMER event
                                                               signals that contain keys
                                                               TOKNENEWTIMER and
                                                               TOKNSNEWTIMER)
 .
 .                                       4. EventWait (on VMTIMER monitor)
 .
 .

5. Timer expires
6. EventSignal (signal VMTIMER event
                with key TOKNENEWTIMER)

                                         7. When VMTIMER event is signaled with
                                            the appropriate key, control is
                                            returned to invoker of EventWait

                                         8. Do processing appropriate to the
                                            expiration of this timer.
```

Figure 26. EventWait and Timer Services

DateTimeSubtract Examples

The DateTimeSubtract function provides a comprehensive facility for the conversion and manipulation of time stamps. The operations that can be performed with DateTimeSubtract include:

- Convert a time stamp from one format to another. For example, the European stamp `21/06/95`—which represents June 21, 1995—can be converted to the USA equivalent `06/21/95`. Formats supported include two-digit-year and four-digit-year character string formats and binary integer formats such as the output of the STORE CLOCK (STCK) instruction.
- Convert a time stamp from one time zone to another. For example, the Eastern Standard Time stamp `06/21/1995 15:30:00` can be converted to the Coordinated Universal Time stamp `06/21/1995 20:30:00`. Precision of up to one second is available for time zone offsets.
- Subtract time stamps of different formats and zones from each other and request the difference in another format and zone. For example, a time of day expressed as Eastern Standard Time in European format can be subtracted from a time of day expressed as Pacific Standard Time in USA format, and the result—an amount of time—can be requested in TOD clock units or a character string form.

DateTimeSubtract accepts a minuend stamp and a subtrahend stamp as inputs and produces a difference stamp as a result. The minuend is the first value in the subtraction expression, and the subtrahend is the value being subtracted. For example, in the expression `5-3=2`, 5 is the minuend, 3 is the subtrahend, and 2 is the difference.

An input stamp is expressed in DateTimeSubtract as follows:

Stamp text
The character string or binary buffer that contains the stamp.

Stamp length
An integer that indicates the length of the stamp in bytes.

Stamp format
An integer that indicates the format of the stamp (character string, binary, or other format).

Stamp bias
If the stamp represents a time-of-day, an integer that indicates the difference in seconds between the zone of the stamp and the UTC zone.

Stamp window type
If the stamp contains a two-digit year, an integer that indicates the type of window being used to provide information about the input year's century digits.

The window is a 100-year span to which the specified two-digit year belongs. For example, two-digit year 93 refers to the year 1993 if the window is [1970,2069], but it refers to the year 2093 if the window is [2000,2099]. DateTimeSubtract supports two kinds of windows:

- A *fixed window* always starts at the same year, no matter what the current year is at the moment of the call. This type of window is useful for manipulating dates known to reside at specific points in time.
- A *sliding window* slides forward in time as the current year moves forward. In other words, the distance between the current year and the beginning of the window is constant. For example, if the distance to the beginning of the window is fifty years and the current year is 1996, the sliding window is [1946,2045]. However, if the same distance is used in the year 2000, the window is [1950, 2049]. A sliding window is useful for manipulating dates known to be no more than a certain number of years old.

Stamp window position
If the stamp contains a two-digit year, an integer that indicates where the window begins. If a fixed window is being used, this is the first year of the window. If a sliding window is being used, this is the distance from the current year to the beginning of the window.

The difference stamp is expressed in DateTimeSubtract as follows:

Stamp buffer
The buffer into which DateTimeSubtract should place the difference stamp.

Stamp buffer size
An integer that specifies the size of the stamp buffer in bytes.

Stamp length
An output produced by DateTimeSubtract that indicates how many bytes of the stamp buffer were filled.

Stamp format
An integer that indicates the format in which DateTimeSubtract should express the difference stamp.

Stamp bias
An integer that indicates the zone in which DateTimeSubtract should express the difference stamp. This input is in seconds and expresses the difference between the desired zone and UTC.

Stamp window type
If the difference is requested in a two-digit-year format, an integer that indicates whether the result should be expressed using a fixed or sliding window. Knowing the type of window being used allows the caller to unambiguously interpret the result.

Stamp window position
If the difference is requested in a two-digit-year format, an integer that indicates the location of the window in which the caller expects the computed result to reside. If a fixed window is being used, this is the first year of the window. If a sliding window is being used, this is the distance from the current year to the beginning of the window.

Because format and zone conversions are just a special case of subtraction, separate zone conversion and format conversion functions are not provided. To accomplish a simple conversion, the caller specifies a relative-format subtrahend of zero, and DateTimeSubtract returns the converted stamp as the difference.

The following examples illustrate how to use DateTimeSubtract to perform various calculations and conversions. For information on DateTimeSubtract syntax and possible parameter values, see "DateTimeSubtract -- Compute Time Differences" on page 115.

Example 1

Problem

Convert 12/31/1995 09:00:00 in Eastern Standard Time (EST) to Pacific Standard Time (PST) and express the result in the same format.

Solution

The inputs and outputs to DateTimeSubtract are shown in Table 5 on page 62.

Table 5. DateTimeSubtract Example 1. In this table, cell contents in ***this font*** are outputs; the rest are inputs.

	Minuend	Subtrahend	Difference
Stamp text	12/31/1995 09:00:00	0/00:00:00	***12/31/1995 06:00:00.000000***
Stamp buffer size	n/a	n/a	32
Stamp length	19	10	26
Stamp format	vm_tmr_format_usa	vm_tmr_format_met	vm_tmr_format_usa
Stamp bias	-18000	ignored	-28800
Stamp window type	ignored	ignored	ignored
Stamp window position	ignored	ignored	ignored

Example 2

Problem

Convert 12/31/1995 09:00:00 in Eastern Standard Time to the format supported by CMS Pipelines in Eastern Standard Time.

Solution

The inputs and outputs to DateTimeSubtract are shown in Table 6 on page 62.

Table 6. DateTimeSubtract Example 2. In this table, cells contents in ***this font*** are outputs; the rest are inputs.

	Minuend	Subtrahend	Difference
Stamp text	12/31/1995 09:00:00	X'0000000000000000'	***19951231090000000000***
Stamp buffer size	n/a	n/a	32
Stamp length	19	8	***20***
Stamp format	vm_tmr_format_usa	vm_tmr_format_tod_relative	vm_tmr_format_pipe
Stamp bias	-18000	ignored	-18000
Stamp window type	ignored	ignored	ignored
Stamp window position	ignored	ignored	ignored

Example 3

Problem

Subtract three hours from the absolute TOD clock value X'0025613602932E00' and express the result in Central Standard Time in the format supported by CMS Pipelines.

Solution

Note that the TOD clock value mentioned corresponds to January 1, 1995, midnight UTC. The inputs and outputs to DateTimeSubtract are shown in Table 7 on page 63.

Table 7. DateTimeSubtract Example 3. In this table, cell contents in ***this font*** are outputs; the rest are inputs.

	Minuend	**Subtrahend**	**Difference**
Stamp text	X'0025613602932E00'	0/03:00:00	***19941231150000000000***
Stamp buffer size	n/a	n/a	32
Stamp length	8	10	***20***
Stamp format	vm_tmr_format_tod_absolute	vm_tmr_format_met	vm_tmr_format_pipe
Stamp bias	ignored	ignored	-21600
Stamp window type	ignored	ignored	ignored
Stamp window position	ignored	ignored	ignored

Example 4

Problem

Convert `12/31/95 09:00:00` in Eastern Standard Time (EST) to Pacific Standard Time (PST) and express the result in ISO format.

Solution

Because the input has a two-digit year, a window must be used to precisely specify the date (that is, identify the century). In this example, a fixed window of [1900,1999] is used to assert that `12/31/95` means "12/31/1995". The inputs and outputs to DateTimeSubtract are shown in Table 8 on page 63.

Table 8. DateTimeSubtract Example 4. In this table, cell contents in ***this font*** are outputs; the rest are inputs.

	Minuend	**Subtrahend**	**Difference**
Stamp text	12/31/95 09:00:00	0/00:00:00	***1995-12-31 06:00:00.000000***
Stamp buffer size	n/a	n/a	32
Stamp length	17	10	***26***
Stamp format	vm_tmr_format_usa_short	vm_tmr_format_met	vm_tmr_format_iso
Stamp bias	-18000	ignored	-28800
Stamp window type	vm_tmr_window_fixed	ignored	ignored
Stamp window position	1900	ignored	ignored

Example 5

Problem

Subtract three hours from `12/31/1995 09:00:00` Eastern Standard Time (EST) and express the result in Pacific Standard Time (PST) in USA two-digit format.

Solution

Because the output is requested in a format that uses a two-digit year, the caller must specify the window in which the result of the calculation is expected to reside. If the calculated difference falls within that window, DateTimeSubtract fills in the difference buffer and indicates success. If the difference falls outside the window, DateTimeSubtract indicates failure.

In this example, a sliding window of [-50,+49] is used; assume the call takes place in 1996. The inputs and outputs to DateTimeSubtract are shown in Table 9 on page 64.

Table 9. DateTimeSubtract Example 5. In this table, cell contents in ***this font*** are outputs; the rest are inputs.

	Minuend	**Subtrahend**	**Difference**
Stamp text	12/31/1995 09:00:00	0/03:00:00	***12/31/95 03:00:00.000000***
Stamp buffer size	n/a	n/a	32
Stamp length	19	10	***24***
Stamp format	vm_tmr_format_usa	vm_tmr_format_met	vm_tmr_format_usa_short
Stamp bias	-18000	ignored	-28800
Stamp window type	ignored	ignored	vm_tmr_window_sliding
Stamp window position	ignored	ignored	-50

Chapter 8. Accounting Services

To facilitate the accounting for multithread servers that service multiple users, CMS adds native accounting services. CMS lets an application access and control the processing of the following types of accounting information:

- Communication requests
- CPU utilization.

Accounting for resource utilization is an optional facility which is enabled and controlled by the application. Accounting information is gathered by CMS and presented by means of an accounting event, VMACCOUNT. Accounting information is then gathered by the application by handling this event by means of the CMS event services. These services are described in the section Chapter 3, “Event Management,” on page 17. To start, stop, or alter the selectivity of accounting, the application calls the service *AccountControl*. This function also allows the application to control how often the accounting records are generated.

CMS also provides the means to associate a given thread or process with an application-defined *account ID*. Among other things, this capability lets servers *charge* requestors with server processing performed for the requestors. For example, the user ID of a virtual machine on whose behalf a set of threads is working can be used as an account ID and assigned to each thread in the set. This account ID is part of the accounting record and so can be used to relate work done by these threads to the requester they were serving. When a thread begins work for another requester, the account ID can be reset. The *AccountIdentify* function can be used to do this.

For details of the AccountControl and AccountIdentify functions, see “AccountControl — Define and Query Accounting Attributes” on page 99 and “AccountIdentify — Identify an Accounting Entity” on page 102, respectively.

The accounting records produced by CMS have the format given in the following structure. It is defined in the language binding files VMCACT H, VMASMACT MACRO, and VMREXACT COPY.

Table 10. Accounting Record

Hex	Dec	Type	Len	Name	Description
00	0	-	28	vm_acct_hdr_fixed	fixed header section
00	0	Signed	4	vm_acct_hdrlen	Length of header data
04	4	Character	16	vm_acct_acctid	Account ID associated with this thread or process as specified by AccountIdentify
14	20	Signed	4	vm_acct_typeid	Type of Accounting record
18	24	Signed	4	vm_acct_datalen	Length of actual accounting data
1C	28	Character	*	vm_acct_vardata	Variable length accounting data
1C	28	Signed	8	vm_acct_cputime	CPU Time (for CPU records)
1C	28	Signed	4	vm_acct_qsends	Number of queue sends (for Communication records)
20	32	Signed	4	vm_acct_qrec	Number of queue receives (for Communication records)

When CMS creates the VMACCOUNT event, the attributes defined for the event are:

- Session scope (all processes in the session can both monitor and signal the event)

© Copyright IBM Corp. 1992, 2023

- Broadcast signals, in that a signal is simultaneously delivered to all qualifying monitors
- Asynchronous signals, so that the signaling thread is allowed to continue executing.

CMS accumulates accounting information for accounting types supported and signals the accounting event. The event signal data is the entire accounting record, with the accounting header record as the key beginning at displacement X'00' and extending up to displacement X'1C' (see previous format). The VMACCOUNT event is signaled either when a user issues AccountControl requesting a generate and set function, or when a timer has expired (interval specified by user).

An accounting setting cannot be set off except by the process that set it on. If multiple processes set on a setting, it is not actually set off until all the processes that had set it on set it off. When a process terminates, its modifications to the accounting settings are reset.

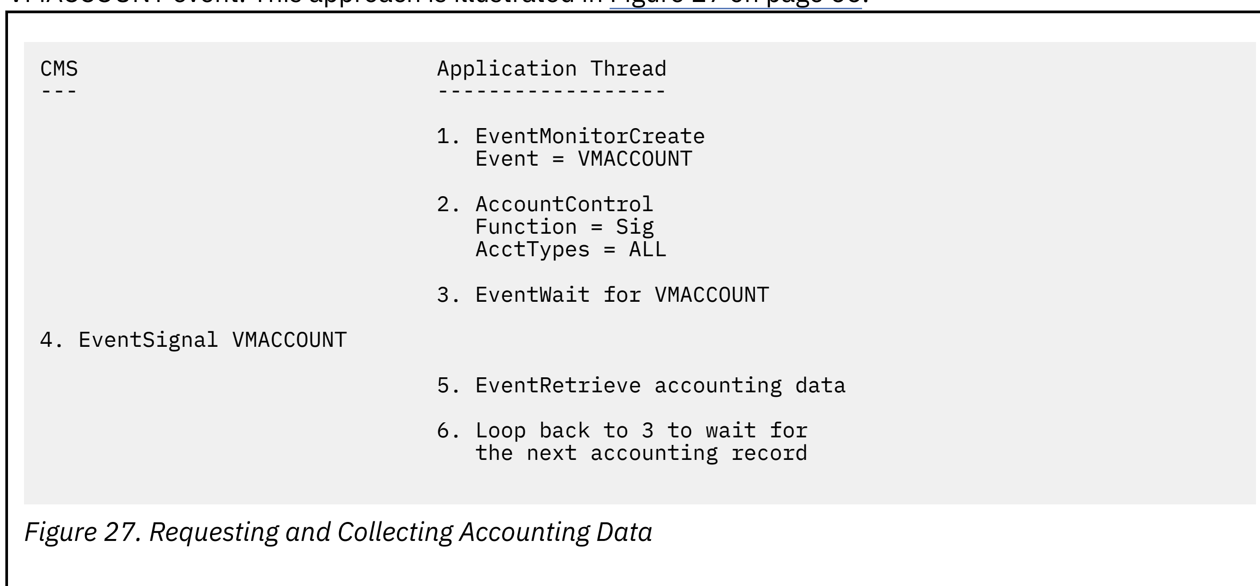
If a time interval has been set by some process, another process can set the interval shorter but not longer. A time interval of zero (no automatic signaling) is considered to be the longest interval. When a process ends the time interval is not altered, unless no accounting settings remain on, in which case the accounting timer is cancelled.

Since the interval can be shortened, a routine that handles the VMACCOUNT event should always assume that there could be multiple events bound to its event monitor and be prepared to process them in a loop.

Applications may also set up their own trace records and account for any other information that is of interest to that application. Using the Event Services API, the application may also signal the accounting event, and set up accounting event monitors to retrieve and process the data. The accounting records can then be processed in real time, rather than waiting for offline processing. In this way, the application can react to accounting information that has been obtained.

Accounting Services Examples

A simple way of using the accounting services is to devote a thread to collecting accounting data. This thread would use AccountControl to select the types of data to be collected and then use event services to wait for and retrieve the accounting records, which are presented to the application through the VMACCOUNT event. This approach is illustrated in Figure 27 on page 66.



```
CMS                             Application Thread
---                             ------------------

                                1. EventMonitorCreate
                                   Event = VMACCOUNT

                                2. AccountControl
                                   Function = Sig
                                   AcctTypes = ALL

                                3. EventWait for VMACCOUNT

4. EventSignal VMACCOUNT

                                5. EventRetrieve accounting data

                                6. Loop back to 3 to wait for
                                   the next accounting record
```

Figure 27. Requesting and Collecting Accounting Data

Chapter 9. Abend Services

During the execution of an application, a serious error may be detected by CMS, or the application itself. These errors start abnormal termination (abend) processing. CMS provides facilities for the application to request abends and to attempt to recover from any abends that do occur.

An abend occurs on a thread when it requests abnormal termination with the AbnormalEnd function, when CMS detects an unrecoverable error performing one of its services, or the thread receives a program check. In any of these cases, CMS lets the application try to recover from the error and continue execution. If the abend is not handled successfully by the application, the application process is abnormally ended and its child processes are deleted. All abend notification and recovery facilities are provided by means of event signaling and monitoring. See Chapter 3, “Event Management,” on page 17 for more information on event handling.

To inform error handlers that an abend has occurred, CMS defines the VMERROR event, which it signals to activate and control error recovery. An error handler is defined by the application by creating an event monitor for the VMERROR event, and an event trap routine can handle the error. Multiple monitors can be defined for the event and the error handlers associated with these monitors are run in LIFO order.

If a process does not handle the error, an additional error event named VMERRORCHILD is signaled in the ancestor processes of the abending process to allow further recovery attempts or first failure data capture by these processes for the abending process.

The data presented to an error handler by the VMERROR event is given in the following structure. It is defined in VMCABN H, VMASMABN MACRO, and VMREXABN COPY.

Table 11. VMERROR Data

Hex	Dec	Type	Len	Name	Description
00	0	Signed	4	vm_errorcode	Error completion code
04	4	Signed	4	vm_errortype	Error type - system/user
08	8	Signed	4	vm_errthread_id	Abending program's thread ID
0C	12	Character	8	vm_errpgm_name	Program name for SVC level
14	20	Address	4	vm_errmod_data_ptr	Pointer to modifiable data area
18	24	Signed	4	vm_errmod_data_len	Length of modifiable data
1C	28	Address	4	vm_error_cmssdwa	Pointer to CMS System Diagnostic Work Area
1C	28	Signed	4	vm_errorprocess_id	Abending program's process ID

The values of the error type field are represented by the constants *vm_abn_type_user* and *vm_abn_type_system* for user abends and system abends, respectively. *Vm_error_cmssdwa* points to the CMS System Diagnostic Work Area, which is mapped by CMSSDWA and contains additional abend information.

The modifiable data area (pointed to from displacement X'14' of the error data) has the following format:

Table 12. Modifiable Data Area

Hex	Dec	Type	Len	Name	Description
00	0	Signed	64	vm_errgprs[16]	General registers 0-15 at time of abend

© Copyright IBM Corp. 1992, 2023

Hex	Dec	Type	Len	Name	Description
40	64	Signed	32	vm_errfprs[4]	Floating point registers 0,2,4, and 6 at time of abend
60	96	Address	4	vm_err_rtepaw	Runtime environment anchor for process owning abending thread
64	100	Address	4	vm_err_rtetaw	Runtime environment anchor for abending thread
68	104	Character	24	-	Unused
80	128	Signed	64	vm_erraccregs[16]	Access registers 0-15 at time of abend
C0	192	Address	4	vm_errpsw	PSW address at time of abend
C4	196	Address	4	vm_erretry_ptr	Amode ANY retry address or 0
C8	200	Signed	4	vm_errnext_inst	Next Sequential Instruction indicator
CC	204	Address	4	vm_erruserdata_ptr	Pointer to optional user data (from AbnormanEnd call)
D0	208	Signed	4	vm_erruserdata_len	Length of optional user data (from AbnormalEnd call)

Table 12. Modifiable Data Area (continued)

Monitoring Error Events

The VMERROR event is defined with the following attributes:

- Process-level, synchronous event (synchronizes the process)
- LIFO propagation

  This implies that each error handler is given a chance to recover, starting from the most recently defined and proceeding back to the first. Once an error handler has determined that it can successfully recover from the error, it can stop the propagation of the event.
- Event key is abend code followed by the type of abend (system or user) and thread ID.

  If a system abend of code X'0C2' is suffered on a thread whose thread ID is X'00000005', the key of the error event would be X'000000C20000000100000005'. When the application created an event monitor for this event, it could have specified a match key of X'5C00000005' to get all abends for thread ID 5, X'000000C25C' for abend X'0C2' on any thread, or X'5C' for any abend on any thread. (X'5C' is the asterisk character, *.)

  For all user abends, any abend code, any thread, the match key may be specified as X'6C6C6C6C000000005C'. (X'6C' is the percent character, %.)

Using event keys, error handlers can be error-specific, thread-specific or both. The handlers can be defined by issuing EventTrap for the error event monitor.

Error Recovery

An event trap routine should be used as an error event handler instead of an event wait routine due to timing considerations when running handlers for the error event. The error event handler can examine the event data included in the VMERROR event and attempt a recovery procedure. While an error event handler is running, all other threads in the process are suspended. Therefore, it can examine and manipulate the data structures of the application safely. If it determines that processing can be restarted, it can specify either that execution of the abending thread should proceed at its next sequential

instruction, or that the abending thread should be restarted at a particular restart address, known as a retry routine. In either of these cases, the registers in the error event data can be altered to correspond to how and where processing is to resume.

If the error handler cannot find a way to recover, it simply resets the event monitor, either by normal return of a trap routine or by issuing EventMonitorReset. Then, other error event handlers can try to recover. If none of the error event handlers specify a retry address, indicate that processing is to resume at the next sequential instruction, or deletes the failing thread, then abend processing proceeds and the process is ended. If the error handler does recover, it can stop the LIFO running of other error handlers by calling EventDiscard. So, the outline of an error event handler is:

1. Issue EventRetrieve to get the error data
2. Attempt recovery
3. Specify one of the following recovery actions:
   - Fill in the retry address in the modifiable section of the error data
   - Set the *next sequential instruction* indicator (by assigning the field vm_errnext_inst the constant vm_abn_next_true)
   - Issue ThreadDelete to delete the failing thread.
4. Issue EventDiscard to stop the running of error event handlers
5. Resets the monitor.

If this error routine could not find a way to recover, it could let other error event handlers get a chance to try recovery by not issuing EventDiscard. CMS continues to run error event handlers until all which are monitoring the particular abend have been run or one of them issues EventDiscard.

If the abend occurs in a program that was invoked from the tasking module by means of CMSCALL (the program is at a later CMS SVC-level), the exit might not have sufficient knowledge of the program to take recovery action. In this case, the error event handler can recover by deleting the failing thread. This will eliminate the failing SVC-level.

Retry Routines

The retry routine is the entry point which gains control as a result of the error event handler filling in a retry address in the error event data. For XA- or XC-mode virtual machines, the high-order bit of the retry address determines the addressing mode of the retry routine. Either the error event handler or the retry routine must assure that the call stack (or save area stack) of the recovered thread is in a valid state. Often, the easiest and safest approach is to reset the call stack to its initial routine (by following the chain of register 13 save areas back to the initial routine) and restart the thread's execution at a well-defined state.

Recovery in the High-Level Language Environment

The recovery mechanism provided by CMS is oriented toward assembler programmers and compiler and language run-time writers. However, these functions can still be used by C programmers directly to perform more tasking-oriented error recovery than that provided by current language support. Essentially, the C programmer has two choices for error recovery using the CMS facilities:

- Delete the failing thread, thus canceling the error condition and allowing the other threads to proceed, or
- Specify that execution should resume on the next sequential instruction of the abending thread.

The first case, deleting the failing thread, is the most direct. First, the error handler would assure that critical data structures are in an appropriate state. Then it would take the thread ID from the error event data and use it as the *thread_ID* parameter on ThreadDelete. Finally, it would issue EventDiscard and EventMonitorReset to end error processing.

The second approach is most applicable to the case in which a thread detects a problem and issues AbnormalEnd. The error event handler could then adjust data structures or take other measures to

assure that processing can proceed and then indicate that processing should continue at the next sequential instruction of the abending thread. Finally, this handler should also issue an EventDiscard and EventMonitorReset to end error processing.

In either case, the handler should discard the event signal before resetting the monitor if it wants to stop propagation of the signal to other event handlers. If a trap routine resets the monitor using EventMonitorReset instead of returning, its execution may be resumed even if none of the error event handlers have recovered. The trap routine is then responsible for determining what, if any, action to take.

Advanced Error Recovery

By using the capabilities of event management, more advanced methods of error recovery are possible.

- Exception Promotion

  If an error event handler determines that, given the state of the processing at the time of the abend, the situation is worse than the abend code would indicate, it can promote the exception. This is done by signaling the VMERROR event with a different (more serious) abend code in the error data. The event data should be a pointer to the error data provided by CMS for the original abend. You should then signal the event, which will result in a synchronous signal because of the event definition, so that the error event handler that promoted the error can see the result of the attempt at recovery of the promoted error. Then, if recovery was successful, the promoting error handler should issue EventDiscard on the original error event.

- Cooperative Recovery

  If an error event handler does not issue EventDiscard, other eligible event handlers will be driven. So, the handler can set such values as the retry routine address and let later error event handlers have a chance to overrule the recovery decision. The state of the error data after all error event handlers have run determines what CMS does with the abend.

- Child Process Recovery

  When a process' error handlers do not recover, the VMERRORCHILD event is signaled in each of its ancestor processes, beginning with its creator and proceeding up the process tree until one of them successfully recovers. This event is of session level with LIFO signal propagation, but unlike the VMERROR event, it does not synchronize the process. The abending process, however, remains suspended during this processing. An additional difference is that event handlers for the VMERRORCHILD event are not restricted to being trap routines. The event data presented to the VMERRORCHILD handlers is identical to that of the VMERROR event and can be manipulated in the same way.

  Because the EventMonitorCreate function allows multiple events to be specified on one monitor, the application need only have one event monitor and one event handler to handle both events.

Interactions with ABNEXIT and Simulated MVS Recovery

The other abend recovery mechanisms in CMS, the ABNEXIT service and the simulated MVS recovery services, do not allow for the multiple process, multiple thread environment. These services can still be used by multitasking programs but they will act as though no multitasking is in effect. The only situation where they cannot be used is in a multiprocessor virtual machine.

The order in which abend recovery routines are invoked is determined by the class of abend. The following list defines the order for each class:

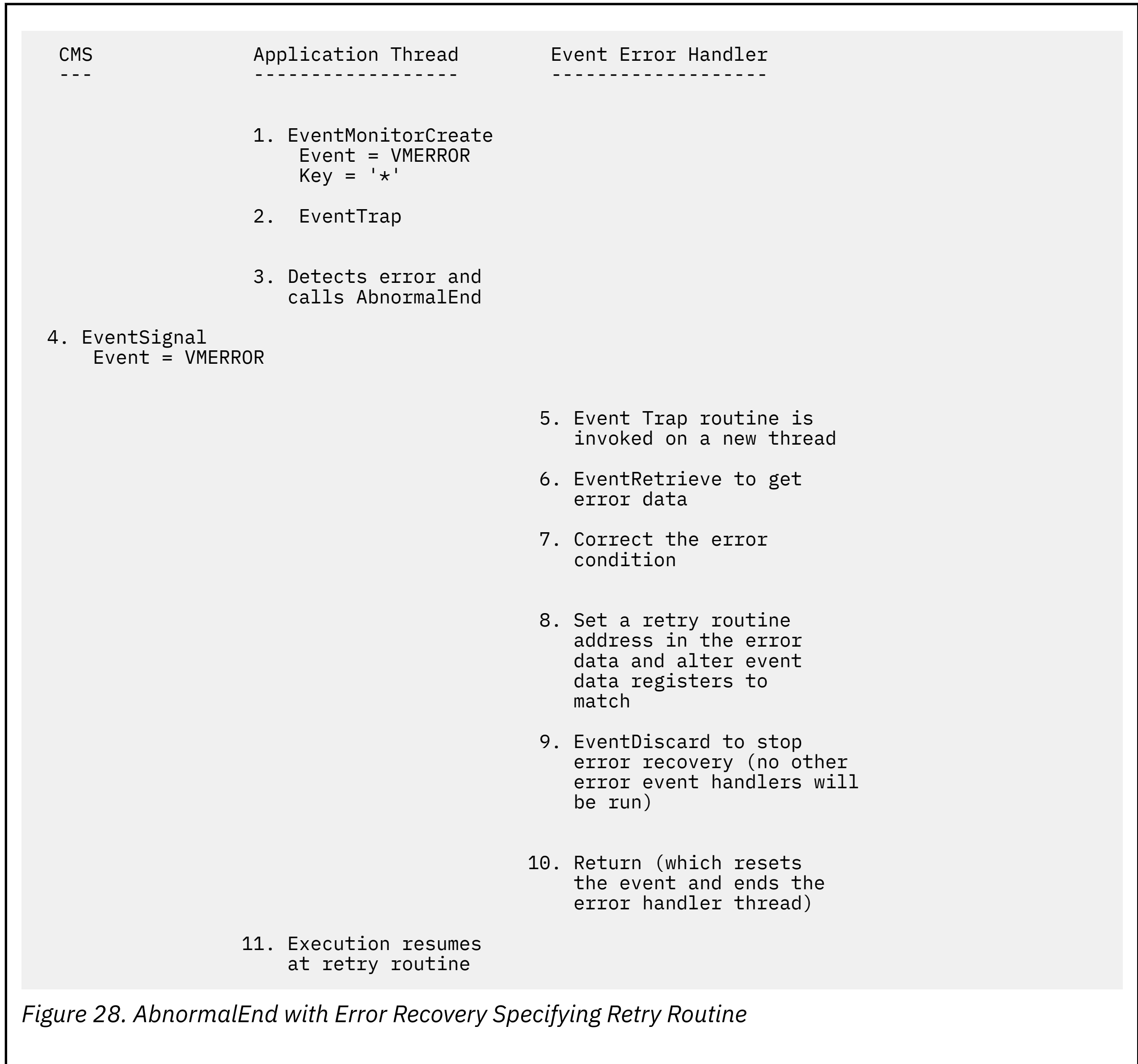
- Program Check

  1. SPIE and ESPIE exits
  2. STAE and ESTAE exits
  3. VMERROR event handlers
  4. VMERRORCHILD event handlers
  5. ABNEXIT routines

- MVS ABEND macro
  1. STAE and ESTAE exits
  2. VMERROR event handlers
  3. VMERRORCHILD event handlers
  4. ABNEXIT routines
- DMSABN macro
  1. VMERROR event handlers
  2. VMERRORCHILD event handlers
  3. ABNEXIT routines
  4. STAE and ESTAE exits
- AbnormalEnd call
  1. VMERROR event handlers
  2. VMERRORCHILD event handlers
  3. ABNEXIT routines
  4. STAE and ESTAE exits

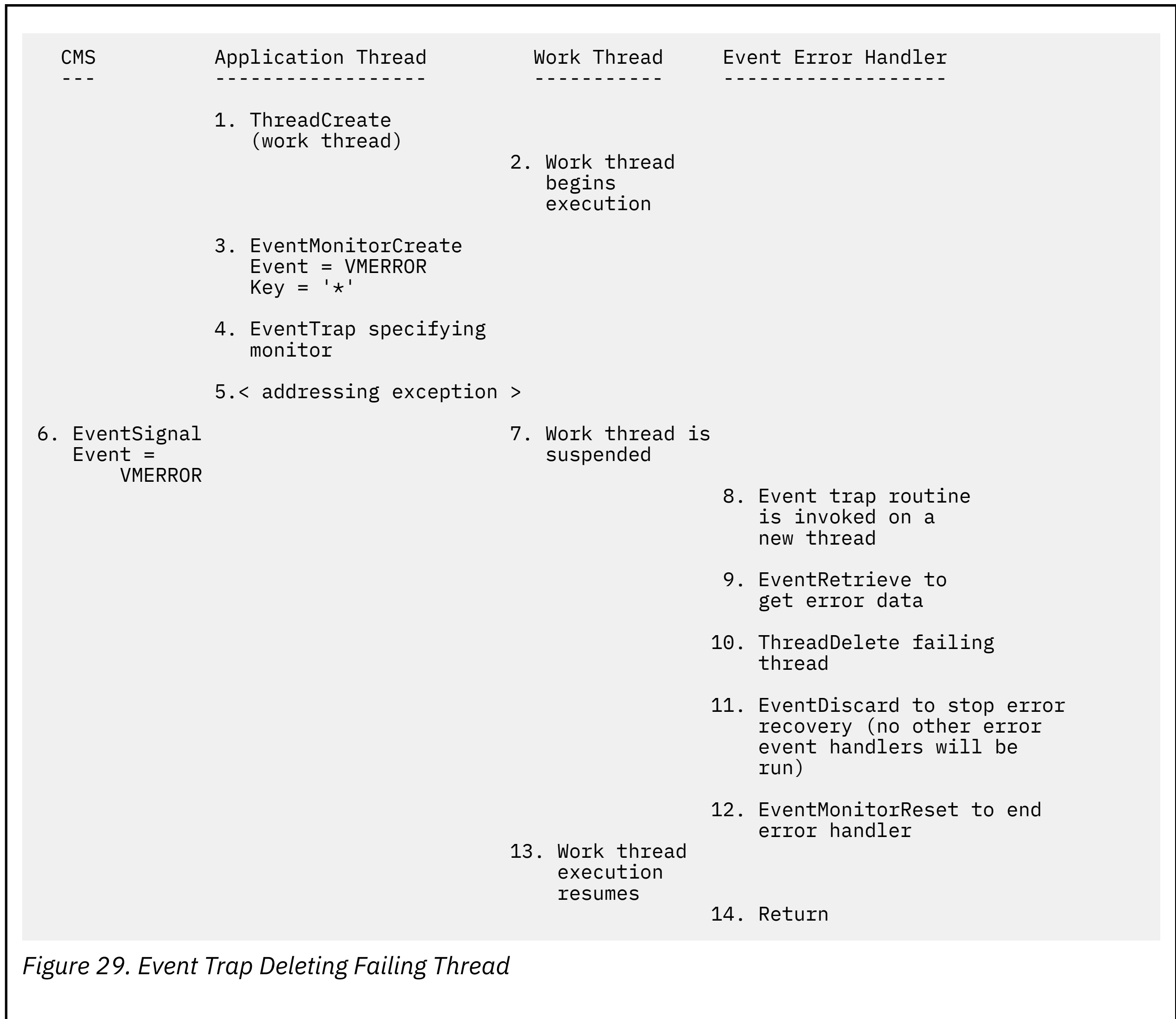
  Only those STAE, ESTAE, SPIE, and ESPIE exits defined in the abending process are driven. ABNEXITs are driven regardless of the process that created them.

Abend Services Examples



```
  CMS              Application Thread        Event Error Handler
  ---              ------------------        -------------------


                   1. EventMonitorCreate
                       Event = VMERROR
                       Key = '*'

                   2.  EventTrap


                   3. Detects error and
                      calls AbnormalEnd

 4. EventSignal
     Event = VMERROR


                                            5. Event Trap routine is
                                               invoked on a new thread

                                            6. EventRetrieve to get
                                               error data

                                            7. Correct the error
                                               condition


                                            8. Set a retry routine
                                               address in the error
                                               data and alter event
                                               data registers to
                                               match

                                            9. EventDiscard to stop
                                               error recovery (no other
                                               error event handlers will
                                               be run)


                                           10. Return (which resets
                                               the event and ends the
                                               error handler thread)

                  11. Execution resumes
                      at retry routine
```

Figure 28. AbnormalEnd with Error Recovery Specifying Retry Routine



```
  CMS          Application Thread         Work Thread     Event Error Handler
  ---          ------------------         -----------     -------------------

               1. ThreadCreate
                  (work thread)
                                         2. Work thread
                                            begins
                                            execution

               3. EventMonitorCreate
                  Event = VMERROR
                  Key = '*'

               4. EventTrap specifying
                  monitor

               5.< addressing exception >

6. EventSignal                           7. Work thread is
   Event =                                  suspended
       VMERROR
                                                          8. Event trap routine
                                                             is invoked on a
                                                             new thread

                                                          9. EventRetrieve to
                                                             get error data

                                                         10. ThreadDelete failing
                                                             thread

                                                         11. EventDiscard to stop error
                                                             recovery (no other error
                                                             event handlers will be
                                                             run)

                                                         12. EventMonitorReset to end
                                                             error handler
                                         13. Work thread
                                             execution
                                             resumes
                                                         14. Return
```

Figure 29. Event Trap Deleting Failing Thread

Chapter 10. Trace Services

This section discusses the CMS trace table and provides user application information.

The CMS Trace Table

For good serviceability, CMS must accumulate and maintain trace information to permit trace entries to be:

- Written without significantly affecting the load and timing characteristics of the system
- Processed in real time
- Processed selectively over a broad range of criteria
- Processed without loss of data.

To accomplish these goals, CMS defines its trace table and a trace API in terms of the CMS Event Services. See Chapter 3, "Event Management," on page 17 for a more detailed description.) The trace information structure maintained by CMS corresponds (loosely) to conventional trace information structures in the following manner:

- A trace event is defined by CMS (named VMTRACE) and corresponds to a trace table.
- Signals of the trace event correspond to trace entries.
- The loose signal limit of the trace event corresponds to the trace table size or *wrapsize*.
- Creation of event monitors sensitive to the trace event allows for selective collection of trace entries.
- EventWait, EventTrap, and EventTest allow for processing of trace entries in real time.

Specifically, CMS provides the following services over and above Event Services for the maintenance of trace information:

- The TraceControl function provides a high-level language interface to the internal trace parameters.
- The TraceSignal function allows applications to build a trace header record and signal the VMTRACE event with the caller's trace data, using the entire trace record (trace header and trace data) as the event key.
- The TRACECTL and QUERY TRACECTL commands set and query trace parameters. See the *z/VM: CMS Commands and Utilities Reference*.

The general flow of this support is as follows:

1. When CMS initializes, it defines the VMTRACE event as asynchronous, session scope, with broadcast signals. The wrapsize is initially set to no limit, subject to the virtual storage size, and the tracing is set off.
2. CMS, at appropriate points in its processing, signals trace events where the event data is the trace entry and the entire trace record, including both the trace header and trace data, is the key. As long as no EventMonitorCreate functions are issued for the trace event, the maximum number of trace entries kept in storage is equal to the wrapsize. When the wrapsize is exceeded, the oldest trace entry is thrown away. Thus, this sort of trace table wraps based on the number of trace entries instead of the number of bytes allotted for a conventional trace table.
3. User program issues EventMonitorCreate to select the trace entries this program is interested in processing. The full power of key specification provided by Event Services is available in selecting trace entries for processing. Trace entries selected by this mechanism do *not* count against the wrapsize defined by TraceControl. In fact, an additional wrapsize is specified for the selected trace entries by the bound signal limit specified on EventMonitorCreate.
4. User program issues EventWait to wait for trace entries selected by the monitor.

© Copyright IBM Corp. 1992, 2023

5. The waiting routine is awakened by the arrival of a selected trace entry and issues EventRetrieve to obtain the trace entry. Once a trace entry is obtained, it is no longer kept in the trace table, so it no longer counts against the bound signal limit specified on EventMonitorCreate.
6. The trace entry is processed and EventWait is reissued to wait for the next selected trace entry.

Of course, this is just one possible flow. Many other flows are possible by exercising both the specific trace functions and the Event Services functions in a variety of ways. Note that the processing of the trace entries is not limited to writing to CPTRAP, or writing to DASD. For example, trace entries may be simply placed on a queue to a service machine, where the overhead of processing them takes place independent of the machine being traced, allowing maximum processing of trace information with minimum impact to the load and timing characteristics of the machine being traced.

On the other end of the spectrum, it seems reasonable to assume that, if trace entries are indeed signaled at significant processing points, the trace signals may be good places to stop machine processing and look around.

Trace entries built by CMS are in the following format:

Table 13. Trace Entry

Hex	Dec	Type	Len	Name	Description
00	0	Signed	4	vm_trc_hdrlen	Length of header data
04	4	Signed	4	vm_trc_typeid	Type of trace event
08	8	Signed	4	vm_trc_subtype	Subtype of trace event
0C	12	Character	16	vm_trc_acctid	Account ID associated with this thread or process as specified by AccountIdentify to categorize trace entries
1C	28	Character	8	vm_trc_userid	User ID of virtual machine issuing the trace signal
20	32	Signed	4	vm_trc_processid	Process ID of process on whose behalf entry is signaled
24	36	Signed	4	vm_trc_threadid	Thread ID of thread on whose behalf entry is signaled
28	40	Fixed	2	vm_trc_cpuid	Address of CPU that signaled the event
2A	42	Fixed	2	vm_trc_res	Reserved
2C	44	Character	8	vm_trc_timestamp	Time stamp when entry is being signaled (TOD clock form)
34	52	Character	8	vm_trc_formatrtn	Formatting routine name
3C	60	Signed	4	vm_trc_datalen	Length of actual trace data
40	64	Character	*	vm_trc_vardata	Variable length trace data

See Appendix B, “CMS Trace Record Formats,” on page 303 for actual format of trace data for specific subtypes of tracing information.

User Application Information

This support interacts heavily with the Event Services.

1. Because the trace table is kept as signals through Event Services instead of as a contiguous area of storage reserved for trace information, the Dump Viewing Facility should be used to look at the trace entries.
2. The *wrapsize* specified on the TraceControl function is for signals with no eligible monitor (loose signals). For monitored trace signals, the bound signal limit constitutes a wrapsize for the trace entries being monitored by that monitor. The signals kept in loose signal wrapsize do not overlap with the signals kept in any bound signal wrapsize.

   As a result of this setup, there are potentially an unlimited number of conceptual trace tables, one for each eligible monitor and one for unmonitored trace events. By specifying different wrapsizes for each monitor and with TraceControl, the trace tables can be biased. For example, if one wishes to keep the last two dispatch entries, the last 20 communication entries, and the last 10 of any other type of entry, one would do the following:

   - Issue TraceControl with a wrapsize of 10
   - Monitor dispatch trace entries with a bound signal limit of 2
   - Monitor communication entries with a bound signal limit of 20.

   Note that this does not require that any of these entries be processed. It only determines the trace entries that CMS keeps in storage.
3. Although the conceptual trace table created by TraceControl does not overlap with the conceptual trace table created by a monitor, two or more monitors may have overlapping trace tables. That is, more than one copy of a given trace signal may exist, depending on the number of monitors to which a given signal is bound. Although no processing is required of bound signals, this capability lets a trace entry be both processed *and* kept in storage. This is accomplished by having two monitors of a selected trace entry, one monitor that has an EventWait, EventTrap, or EventTest associated with it, and one monitor that does not.
4. Allowing trace entries to be processed in real time is potentially an extremely powerful capability. Although trying to list all the new possibilities is impossible, here are a few examples just to give an idea of this power.

   - Writing selected trace data to a real file system
   - Sending selected trace data from one virtual machine to another for processing
   - Sending selected trace data from several virtual machines to another for processing
   - Sending selected trace data from several virtual machines to another real machine (a workstation perhaps) for processing
   - Monitoring processes' use of resources for simple tracking
   - Monitoring processes to adjust thread and process dispatching priority
   - User exit processing
   - Debugging exit processing.

Chapter 11. CMS Monitor Data

As part of its normal processing, the CMS dispatcher counts certain events that are relevant to multithreading and the use of CMS POSIX support. The events counted are:

ThreadCreate count
The number of threads created

ThreadCreate time
The amount of time spent creating threads (TOD clock units)

ThreadDelete count
The number of threads deleted

ThreadDelete time
The amount of time spent deleting threads (TOD clock units)

Slow switch count
The number of times the "slow path" through the CMS dispatcher was taken

Fast switch count
The number of times the "fast path" through the CMS dispatcher was taken

Blocked threads
The current number of blocked threads

Process watermark
The greatest number of processes that have existed concurrently

Thread watermark
The greatest number of threads that have existed concurrently

Process limit failures
The number of times an attempt to create a POSIX process has failed because the process limit was reached

Counting of these events begins when the first multithreaded or POSIX application executes, and the counts accrue in a data structure called the CMS monitor data area.

CMS itself does nothing with the counts that accrue in the monitor data area, but applications desiring to use the information collected can obtain the address of the monitor data area by calling CSL routine MonitorBufferGet.

CMS provides C and assembler language bindings that map the monitor data area. A program written in one of these languages can use the binding to simplify its references to the data area.

© Copyright IBM Corp. 1992, 2023

Chapter 12. Writing Multitasking Applications

The previous chapters in this book introduced the basic multitasking concepts and discussed the various types of multitasking services. This chapter describes how to write a multitasking application in C, assembler, or REXX/VM. You code calls to CMS multitasking functions in your application source as if they were ordinary C functions, assembler subroutines, or REXX calls.

This chapter contains the following sections:

- "VMMTLIB Callable Services Library" on page 81
- "Programming Language Binding Files" on page 81
- "Writing Multitasking Applications in C" on page 83
- "Writing Multitasking Applications in Assembler" on page 85
- "Writing Multitasking Applications in REXX/VM" on page 89
- "General CMS API Considerations" on page 92

VMMTLIB Callable Services Library

CMS multitasking functions are provided by callable services library (CSL) routines. These routines reside in the VMMTLIB callable services library. VMMTLIB is contained within the CMS nucleus and is automatically loaded during CMS initialization before the system profile (SYSPROF EXEC) is run. CMS multitasking CSL routines cannot be dropped by using the RTNDROP command, although calls to them can be intercepted by routines that are loaded by the RTNLOAD command.

Programming Language Binding Files

For each supported programming language (C, assembler, and REXX), CMS provides a set of programming language binding files that declare external functions, constants, and return and reason codes. These are included in the C program source through the `#include` statement, in the assembler language source by a macro invocation, or in a REXX exec by a call to the APILOAD function. (For information about APILOAD, see the *z/VM: REXX/VM Reference*.)

The following header files are provided for C programs:

Table 14. Header Files for C Programs

Header File	Contents
VMCPRO	Process Management
VMCIPC	Interprocess Communication
VMCSYN	Synchronization
VMCEVN	Event Services
VMCCPU	Processor Configuration
VMCTRC	Trace Services
VMCACT	Accounting Services
VMCTMR	Timer Services
VMCABN	Abend Services
VMCMON	CMS Monitor Data
VMCMTR	Return and Reason Codes

© Copyright IBM Corp. 1992, 2023

Header File	Contents
VMCMT	Includes all of the above header files

Table 14. Header Files for C Programs (continued)

The following macros are provided for assembler programs:

Table 15. Macros for Assembler Programs

Macro	Contents
VMASMPRO	Process Management
VMASMIPC	Interprocess Communication
VMASMSYN	Synchronization
VMASMEVN	Event Services
VMASMCPU	Processor Configuration
VMASMTRC	Trace Services
VMASMACT	Accounting Services
VMASMTMR	Timer Services
VMASMABN	Abend Services
VMASMMON	CMS Monitor Data
VMASMMTR	Return and Reason Codes
VMASMMT	Includes all of the above macros

The following COPY files are provided for REXX programs:

Table 16. COPY Files for REXX Programs

COPY File	Contents
VMREXPRO	Process Management
VMREXIPC	Interprocess Communication
VMREXSYN	Synchronization
VMREXEVN	Event Services
VMREXCPU	Processor Configuration
VMREXTRC	Trace Services
VMREXACT	Accounting Services
VMREXTMR	Timer Services
VMREXABN	Abend Services
VMREXMON	CMS Monitor Data
VMREXMTR	Return and Reason Codes
VMREXMT	Includes all of the above COPY files

The program source must include at least one of the language binding files for multitasking services to be usable from the application. You must ensure that all the function definitions used are included in the program. A safe approach is to include VMCMT, VMASMMT, or VMREXMT, which defines all the functions

at once. If you include function definitions selectively, you must also explicitly include the return and reason code definitions (VMCMTR, VMASMMTR, or VMREXMTR).

Writing Multitasking Applications in C

To write a multitasking application in C, either the IBM IBM C/C++ for z/VM compiler or the IBM IBM C for VM/ESA compiler can be used. The C programmer can choose between two entry linkage conventions:

- The entry linkage provided through the run-time library support for POSIX threading
- The entry linkage provided by the CMS multitasking `applmain()` convention

Using the C POSIX Entry Linkage

Using the POSIX entry linkage means that the programs are treated by CMS as OpenExtensions applications, even if they do not use POSIX services. Because they are OpenExtensions applications, they are compiled and built using the `c89` command.

The `c89` command performs the compile, prelink, and build steps and uses the appropriate C TXTLIBs without referencing the GLOBAL TXTLIB list. To compile and build a C multitasking program, issue the following command:

```
c89 //pgmname.c -l//vmmtlib
```

This will produce the file *pgmname* MODULE. This is only a simple example of `c89`. It can use CMS minidisks, SFS, or the OpenExtensions byte file system (BFS) for input or output. For a full description of how to use `c89` to create OpenExtensions applications, see the *z/VM: OpenExtensions Commands Reference* and the *XL C/C++ for z/VM: User's Guide*.

A C multitasking program looks like a standard C application:

```
    /* specify POSIX(ON) so C can handle multiple threads */

    #pragma runopts(POSIX(ON))

    /* include multitasking library constants and functions */

    #include "vmcmt.h"

    /* use main() as in a standard C program */

    int main(int argc, char *argv)
    {
        .
        .
        .
     /* rest of the application */
        .
        .
        .
    }
```

One important difference between the POSIX(ON) mode of operation and the POSIX(OFF) mode is how the file ID is interpreted by the C library routines that accept a file ID as a parameter. With POSIX(ON), the file ID is assumed to indicate a BFS file unless the file ID is preceded by two slashes (//). For example, to refer to file DATA FILE A, you would specify `//data.file.a` instead of `data.file.a`.

Another important difference between POSIX(ON) mode and POSIX(OFF) mode is in the behavior of the `system()` function. For POSIX(ON) applications, `system()` passes the specified string to the OpenExtensions shell for execution, unless the environment variable `__POSIX_SYSTEM` is set to no, No, or `NO`. In the POSIX(OFF) mode, the specified command string is passed to the CMS command interpreter.

Using the CMS Multitasking applmain() Linkage

The entry point from the CMS multitasking application is the `applmain()` function. (C programmers should *not* provide a `main()` function.) CMS calls `applmain()` after process initialization is complete and the language environment is active. `applmain()` must be defined to use OS linkage.

Four parameters are passed to `applmain()`:

Parameter	Usage
ext_plist	A pointer to the extended parameter list CMS passed to the tasking module, or zero if no extended parameter list was provided.
tok_plist	A pointer to the tokenized parameter list CMS passed to the multitasking module.
scblok	A copy of the R2 value CMS passed to the multitasking module. If the multitasking module is loaded as a nucleus extension, then this pointer points to the nucleus extension's SCBLOCK.
usersave	A pointer to a copy of the USERSAVE area CMS provided for the module. This copy is provided so that the programmer can interrogate the USERINFO doubleword in USERSAVE.

C programmers can choose whether to take advantage of these parameters — they need not declare `applmain()` as taking any parameters at all. Note that IBM does not provide C structure definitions for the extended parameter list, tokenized parameter list, SCBLOCK, or USERSAVE area. For information on how these blocks are organized, see the *z/VM: CMS Macros and Functions Reference*.

The program must contain one or more `#include` statements specifying the binding files to be used. See Table 14 on page 81 for a list of the C header files and their contents.

A C multitasking program using `applmain()` might look like this:

```
/* include multitasking library constants and functions */

#include "vmcmt.h"
#pragma linkage(applmain,OS)

/* use applmain instead of main to get started... */

int applmain ( ext_plist, tok_plist, scblok, usersave )
   void * ext_plist;
   void * tok_plist;
   void * scblok;
   void * usersave;
{
     .
     .
     .
  /* rest of the application */
     .
     .
     .
}
```

Building an applmain() Enabled Program

To build an `applmain()` enabled multitasking program, the programmer must compile the source so that the language binding files are included, then combine the resultant text files with a special multitasking initialization routine and a language environment selector text file. The binding files reside on the CMS system disk, so C will pick them up automatically.

When the multitasking application is bound into a module through the normal CMS LOAD, INCLUDE, GENMOD, or LKED commands, the multitasking initialization entry routine (VMSTART) and the language environment selector (DMSCES) are also included. You should specify the text libraries DMSCMT, DMSCENV, and VMMTLIB on a GLOBAL TXTLIB command to access the VMSTART routine, the C programming language environment exits, and the CSL stub routines, respectively:

```
GLOBAL TXTLIB DMSCMT DMSCENV SCEESPC SCEELKED VMMTLIB
```

Then, proceed with the usual module generation procedure:

```
LOAD    file1
INCLUDE file2
INCLUDE file3
```

```
 .
 .
 .
INCLUDE filen
INCLUDE VMSTART ( LIBE RESET VMSTART
GENMOD pgmmod
```

You can specify any options on the LOAD and INCLUDE commands as long as you include the options shown above.

Restrictions when Using applmain()

Overall, the C programmer using multitasking has the z/VM API, the C language constructs, and the C run-time library available for use. However, for CMS to create and maintain a multitasking environment, and to allow the application to obtain expected results from C library functions, a few restrictions apply.

The functions of the C run-time library can be used by a multitasking application. Each thread or event trap routine executes in its own private instance of the C run-time environment. While this allows each thread to use the run-time library without interference between threads, this separation has implications on thread cooperation:

- Although the C heap storage is usually tied to an instance of the run-time environment, CMS C multitasking language support intercepts requests for heap storage and allows sharing of the heap. This means, for example, that one thread can allocate a block of storage and a second thread can release it. CMS performs this interception for calls to `malloc()`, `calloc()`, `realloc()`, and `free()`.
- Because `applmain()` employs the system programmer C facilities, the same restrictions apply as are documented for the system programming environment in *z/OS®: XL C/C++ Programming Guide*. Threads are executed as persistent C environments with access to the C Specific Library. The chief restriction in this environment is that the application cannot use the C-PL/I Common Library. Consult the C documentation for a full list of restrictions.

Calling Multitasking Functions from C

The format for calling multitasking routines from C is:

```
                      ┌─── , ───┐
                      ▼         │
►► rtnname ── ( ──────── parm ──┴── ) ── ; ─►◄
```

Note: You must use the direct call format. You cannot use DMSCSL to call CMS multitasking functions.

Although the C header files provided by CMS specify OS linkage on the `#pragma linkage` statement for each function, the usual C call-by-value approach is maintained for the C programmer. All input parameters are passed by value; for output parameters, addresses of variables to receive values are passed by value. For example, the ThreadYield function takes one input parameter, the thread ID, and returns two values, the return code and the reason code. So a call to ThreadYield from C would look like this:

```
ThreadYield(&retcode,&reascode,tid);
```

Notice that the *address of* operator & is used with the two output parameters.

For an extended example of a C program, showing calls to a number of multitasking functions, see Appendix D, “Example of a C Multitasking Program,” on page 319.

Writing Multitasking Applications in Assembler

The main entry point of the assembler program must have the external name APPLMAIN. CMS calls APPLMAIN after process initialization is complete and the language environment is active. The linkage used for APPLMAIN is exactly the linkage that would have been used by CMS had it called APPLMAIN directly. The following register conventions apply:

Register	Usage
R0	A pointer to the extended parameter list CMS passed to the tasking module, or zero if no extended parameter list was provided.
R1	A pointer to the tokenized parameter list CMS passed to the tasking module.
R2	A copy of the R2 value CMS passed to the tasking module. If the tasking module is loaded as a nucleus extension, then this pointer points to the nucleus extension's SCBLOCK.
R12	The address of APPLMAIN
R13	A pointer to a copy of the USERSAVE area CMS provided for the module. This copy is provided so that the programmer can interrogate the USERINFO doubleword in USERSAVE.
R15	The address of APPLMAIN

The program must contain one or more macro invocations specifying the binding files to be used. See "Programming Language Binding Files" on page 81 for a list of the assembler macros and their contents.

Calling Multitasking Functions from Assembler

The services provided by CMS can be treated as external subroutines to the tasking application. They are invoked using BALR 14,15 linkage with R1 containing the address of a list of addresses of parameters, as defined by standard Type-1 linkage conventions:

Register	Usage
R1	Address of a list of addresses of the parameters
R14	Return address
R15	Address of the entry point for the function

The external symbols and parameter address list DSECTs are defined in the assembler binding files. The application can call CMS functions without using the parameter list DSECTs provided. They are provided only as a convenience, in case the programmer wishes to use symbolic names for the entries in the address list. The names of the DSECT fields are documented only in the binding files themselves. However, the important point is that the order of parameters is defined in the reference documentation for each function. That is all you need to know to build the parameter list.

An assembler call to the ThreadYield function would look like:

```
L     R1,address of data area
USING VMTHRYI_PLIST,R1
MVC   VMTHRYI_PLIST_RC,=A(RC)       Address of return code
MVC   VMTHRYI_PLIST_RE,=A(RE)       Address of reason code
MVC   VMTHRYI_PLIST_TID,=A(TID)     Address of Thread ID
L     R15,=A(THREADYIELD)
BALR  R14,R15                       Invoke ThreadYield
```

The assembler language technique presented above produces only serially-reusable code. This is because parameters RC, RE, and TID all reside in storage locations whose addresses are constants computed by the loader at load time (note the use of "=A" in referring to those parameters). If reentrant code is required, then all parameters and the parameter list must reside in dynamically-obtained storage. This means that the parameter list must be built at run time using LA and ST instructions. For example, if we assume that R1 points to a fragment of dynamically-obtained storage for the parameter list, and if we assume that a base register for dynamically-obtained storage for the parameters themselves (for example, R13) has already been set up, then the following code fragment produces reentrant code:

```
USING    VMTHRYI_PLIST,R1
LA       R2,RC
ST       R2,VMTHRYI_PLIST_RC
LA       R2,RE
```

```
      ST       R2,VMTHRYI_PLIST_RE
      LA       R2,TID
      ST       R2,VMTHRYI_PLIST_TID
      L        R15,=A(THREADYIELD)
      BALR     R14,R15
```

If a section of code is shared among threads in a single dispatch class, and if that code contains a point at which control could be lost, then that code must be reentrant. If a section of code is shared among threads in multiple dispatch classes, then that section of code must be multiprocessor-capable.

A simpler way to handle the construction of the parameter address list and the function invocation is to use the CALL macro provided by CMS. It builds the parameter list automatically and generates the BALR instruction. The CALL macro does not support long names, so instead of specifying the routine name, use the register form.

The format of the CALL macro is:

```
                             ┌─ , ─┐
▶▶─ CALL ── (reg) ── , ── ( ─┴ parm ┴─ ) ─▶◀
```

Note: You must use the direct call format of the CALL macro. You cannot use CALL DMSCSL or the CSLFPI macro to call CMS multitasking functions.

So the call to ThreadYield would look like this:

```
     L      R15,=A(THREADYIELD)
     CALL   (15),(RC,RE,TID)
```

If you use the parameter list DSECTs provided, you can determine both the length of a particular parameter list DSECT and the largest parameter list DSECT included in your program. The length of a particular parameter list DSECT is in a variable named *funcname*_PLIST_LENGTH. The parameter list DSECT for ThreadYield looks like:

```
VMTHRYI_PLIST            DSECT
VMTHRYI_PLIST_RC         DS     F
VMTHRYI_PLIST_RE         DS     F
VMTHRYI_PLIST_THREAD_ID  DS     F
VMTHRYI_PLIST_LENGTH     EQU    *-VMTHRYI_PLIST
```

Determining the largest parameter list DSECT included in a program requires a few more assembler instructions. The length of the largest parameter list DSECT is found in a global macro variable called &DMAX. To use this length:

1. Include the assembler binding files at the end of the program.
2. Declare the global macro variable &DMAX before the assembler binding files.
3. Define the variable with the largest parameter list DSECT length after the assembler binding files. The largest length is the address of &DMAX. A CSECT statement is required between the assembler binding files and the largest length variable.

Finding the largest length in the parameter list DSECTs provided would look like this:

```
TESTASM           CSECT
          .
          .
          .
                  GBLA      &DMAX
                  VMASMMT
TESTASM           CSECT
MAX_PLIST_LENGTH  DC        A(&DMAX)
                  END
```

Outline of an Assembler Application

The outline and structure of an assembler tasking application is the same as that of a C application. This example points out the assembler specifics.

```
*-----------------------------------------------------------------*
* CMS will pass control to the entry point APPLMAIN               *
*-----------------------------------------------------------------*

APPLMAIN CSECT
         STM     R14,R12,12(R13) Save input registers
         LR      R12,R15         Register 15 has entry point address
         USING   APPLMAIN,R12
         .
         .
         .
    process parameters passed in CMS tokenized and extended PLISTs,
    if any
         .
         .
         .
*-----------------------------------------------------------------*
* Create a thread to start execution at label THREAD1.            *
*-----------------------------------------------------------------*

         L       R15,=A(THREADCREATE)
         CALL    (15),(RC,RE,TID,FLAGARRAY,FLAGSIZE,PRIORITY,THRADDR,  X
                       PLIST, PLISTLEN)
         .
         .
         .
         LM      R14,R12,12(R13) Restore regs
         BR      R14             End of initial thread
```

```
THREAD1  DS      0H
         .
         .
         .
    Perform concurrent work
         .
         .
         .
         LM      R14,R12,12(R13) Restore regs
         BR      R14             End of Thread1
         .
         .
THRADDR  DC      A(THREAD1)
         .
         .
    other data definitions
         .
         .
         VMASMMT                 Binding file macro invocation
         .
         .
         END     APPLMAIN
```

Building an Assembler Multitasking Program

To build an assembler multitasking program, the programmer must compile the source so the language binding files are included, then combine the resultant text files with a special tasking initialization routine and a language environment selector text file. The binding files for assembler are in DMSGPI MACLIB, so you should specify this library on a GLOBAL MACLIB command to make them available to the assembler:

```
GLOBAL MACLIB DMSGPI
```

When the multitasking application is bound into a module, through the normal CMS LOAD, INCLUDE, GENMOD, or LKED commands, the tasking initialization entry routine (VMSTART) and the language environment selector (DMSAES) are also included. You should specify the text libraries DMSAMT and VMMTLIB on a GLOBAL TXTLIB command to access the VMSTART routine and CSL stub routines, respectively:

```
GLOBAL TXTLIB DMSAMT VMMTLIB
```

Then, proceed with the usual module generation procedure:

```
LOAD    file1
INCLUDE file2
INCLUDE file3
 .
 .
 .
INCLUDE filen
INCLUDE VMSTART ( LIBE RESET VMSTART
GENMOD pgmmod
```

You can specify any options on the LOAD and INCLUDE commands as long as you include the options shown above.

Writing Multitasking Applications in REXX/VM

Execs written in REXX/VM can be used as an integral part of a multitasking application. REXX procedures can execute concurrently on multiple threads and call multitasking functions. A REXX exec itself cannot be the main entry point for a process and cannot directly create threads. Instead, REXX execs can be invoked by multitasking programs in different threads and can invoke programs bound with the VMSTART initialization routine that result in new processes which can in turn create additional threads.

The REXX/VM interpreter does not itself take advantage of the multitasking environment. This implies that although there can be execs active on multiple threads, only one of them can be dispatched at a time. If a REXX exec issues a call that blocks the thread, such as QueueReceiveBlock or EventWait, another thread that could itself be running a REXX exec can be dispatched. The only limitation arises from the fact that it is not possible to generate an entry point address corresponding to a procedure or label in an exec. Therefore, you cannot use the ThreadCreate function to create a new thread whose entry point is in an exec. Similarly, you cannot use the *trap_routine_address* parameter of the EventTrap function to designate an event trap routine in an exec.

The EXECCOMM environment for an exec is local to the thread on which it is running. This means that two concurrent REXX execs cannot affect each other's variable pool, and that the EXECCOMM can only be used to communicate with execs or programs running on the same thread. The CMS terminal input buffer and program stack are shared by all threads so care must be taken when using them from multiple concurrent execs. This could include using a mutex to serialize access to the stack.

To use the services provided in the VMMTLIB callable services library, the exec must invoke the APILOAD function to initialize REXX variables with the values defined in the REXX language binding files. To make the entire multitasking API available, the exec would issue:

```
Call APILOAD 'VMREXMT'
```

See "Programming Language Binding Files" on page 81 for a list of the REXX binding files and their contents. The APILOAD function is described in the *z/VM: REXX/VM Reference*.

Calling Multitasking Functions from REXX

In REXX/VM, CSL routines can be called as REXX functions. However, the preferred method is to call them as subroutines using the CALL instruction:

REXX function call

```
>>─ CSL ── ( ── '── rtnname ──┬─ parm ─┬── ' ── ) ─><
                               └───◄────┘
```

REXX CALL instruction

```
>>─ CALL CSL ── '── rtnname ──┬─ parm ─┬── ' ─><
                               └───◄────┘
```

For multitasking functions requiring pointer values, the actual name must be specified. The following example shows a call to the EventMonitorCreate function from a REXX exec:

```
/*  Call the EventMonitorCreate service  */

Call APILOAD 'VMREXMT'

monitor_flag.1 = vm_evn_no_auto_delete
monitor_flag.2 = vm_evn_async_monitor
monitor_flag.3 = vm_evn_bind_loose_signals

monitor_flag_size = 3

number_of_events = 1

event_name_address.1 = 'VMCONINPUT'
event_name_length.1 = Length(event_name_address.1)

event_key_address.1 = '*'
event_key_length.1 = Length(event_key_address.1)

bound_signal_limit.1 = -1

event_count = 1

Call CSL 'EventMonitorCreate retcode reascode monitor_token',
         'monitor_flag monitor_flag_size number_of_events',
         'event_name_address event_name_length event_key_address,
         'event_key_length bound_signal_limit event_count'

say '  retcode =' retcode
say '  reascode =' reascode

Exit
```

A number of multitasking functions use parameter lists that contain arrays of integers. In REXX these are expressed with stem variables. The following example shows a call to the TraceControl function from a REXX exec. This function contains two arrays, one containing trace types and another giving the settings for the corresponding values.

```
/* Call the TraceControl service */

Call APILOAD 'VMREXTRC'

wrap = 5
num_types = 2
tracetypes.1 = vm_trc_comm
tracetypes.2 = vm_trc_disp
tracetype_settings.1 = vm_trc_on
tracetype_settings.2 = vm_trc_on

Call CSL 'TRACECONTROL RETCODE REASCODE VM_TRC_ITRACE WRAP',
          'NUM_TYPES TRACETYPES TRACETYPE_SETTINGS'

Say 'Return Code =' RETCODE
Say 'Reason Code =' REASCODE
```

Because of a CSL REXX restriction, several arrays need to be handled in a special manner. Table 17 on page 90 has the list of these special cases:

Table 17. Special case functions when called from REXX/VM

Function Name	Parameter	Number of Elements
EventCreate	event_flag	3
EventMonitorCreate	monitor_flag	3
EventModify	event_flag	1
EventMonitorQuery	monitor_flag	4
QueueOpen	search_sequence	3

Table 17. Special case functions when called from REXX/VM (continued)		
Function Name	**Parameter**	**Number of Elements**

To use these functions in a REXX/VM exec, you **must** define a stem variable **and** define **all** the elements of the array. For example, when calling QueueOpen, you must define a stem variable for the *search_sequence* parameter. You must also assign values to all 3 elements of the array even if you are only going to use one of them. For example, if you call the stem variable *search_sequence.* and you want to search the process export level only, then you must assign one of the legal search sequence values to *search_sequence.1* and assign zeros to *search_sequence.2* and *search_sequence.3*. Using all of the above, the REXX/VM snippet for this looks like:

```
…
search_sequence.1 = vm_ipc_plevel
search_sequence.2 = 0
search_sequence.3 = 0

search_sequence_length = 1
…
```

When using the TimerStartMicros, TimerStopMicros, or TimerTestMicros functions, you need to manipulate numbers to make the calls work. These three functions all require as input an 8-byte, signed, binary number in the *interval* variable. Now, REXX deals in character data and the REXX CSL cannot handle an 8-byte, signed, binary number. So, you need to convert the data in the *interval* variable into something REXX and CSL can handle. The following code snippet shows how to do this:

```
interval = 100
interval = D2X(interval,16)
interval = X2C(interval)
```

These instructions can be combined into a single statement. They are shown here as separate statements to clearly illustrate the conversion needed to feed an 8-byte, signed, binary variable into a REXX CSL function.

Using Binding Files with REXX Procedures

If your REXX program contains any internal subroutines that are introduced by "procedure" statements, it is important for you to remember that language bindings you load with APILOAD in the main routine will not automatically be visible inside these subroutines. This is illustrated by the following example:

```
/* SLOWSAY EXEC */

Call APILOAD 'VMREXMTR'
Call APILOAD 'VMREXPRO'
x = saywithwait('This is a test')
return 0

saywithwait: procedure
 parse arg what
 say what
 /* VMREXPRO bindings are invisible here, so the   */
 /* following WILL NOT WORK... 'ThreadDelay' will  */
 /* be undefined                                   */
 interval = 1000
 call csl 'ThreadDelay cslrc cslre interval'
 return 0
```

If your procedure needs to use bindings loaded with APILOAD, you have three choices:

- Load the bindings in your mainline program and use EXPOSE to expose the specific constants and procedure names your procedure will need. For example:

```
/* SLOWSAY EXEC */

Call APILOAD 'VMREXMTR'
Call APILOAD 'VMREXPRO'
```

```
x = saywithwait('This is a test')
return 0

saywithwait: procedure expose,
 ThreadDelay

 parse arg what
 say what
 interval = 1000
 call csl 'ThreadDelay cslrc cslre interval'
 return 0
```

- Avoid using the REXX "procedure" statement altogether; just use a label. For example:

```
/* SLOWSAY EXEC */

Call APILOAD 'VMREXMTR'
Call APILOAD 'VMREXPRO'
x = saywithwait('This is a test')
return 0

saywithwait:             /* no procedure statement */
 parse arg what
 say what
 interval = 1000
 call csl 'ThreadDelay cslrc cslre interval'
 return 0
```

- If performance considerations will permit it, load the bindings inside your procedure. For example:

```
/* SLOWSAY EXEC */

x = saywithwait('This is a test')
return 0

saywithwait: procedure
 call APILOAD 'VMREXMTR'
 call APILOAD 'VMREXPRO'
 parse arg what
 say what
 interval = 1000
 call csl 'ThreadDelay cslrc cslre interval'
 return 0
```

You will have to choose one of these approaches based on your program's particular needs.

General CMS API Considerations

The following restrictions and considerations apply to assembler applications, assembler subroutines of C applications, or assembler subroutines called by REXX execs. CMS provides improved, tasking-oriented alternatives to the services mentioned below. While all of the CMS API can be used by the tasking application, in some cases a choice must be made between using certain CMS functions and taking full advantage of the multitasking capabilities.

- The application cannot issue HNDEXT to trap SIGP interrupts.
- ABNEXIT, STAE, ESTAE, SPIE, ESPIE and SETRP should not be used if multiprocessor-mode is being used. They generate unpredictable results. In a virtual machine with one virtual processor these services can be used, although they do not adequately address the multi-threaded case.
- DMSKEY should not be used in multiprocessor-mode, because CMS does not maintain key stacks for more than one virtual processor.
- Multitasking functions can be called by a program running in access register mode in an ESA/XC or z/XC virtual machine, but all parameters must be in the primary address space. These functions do not use access registers to reference storage in a data space.
- CMS does not manage vector registers or IEEE floating point registers as part of the thread context.
- When running in multiprocessor mode, use only the ENABLE macro to enable or disable interrupts. Do not directly manipulate control register 6 to change the I/O interruption-subclass mask.
- CMS DOS mode does not support multitasking programs.

- Application interrupt handlers established by CMS services run as extensions to CMS interrupt handling. As is the case with other CMS services, some multitasking services should not be used in such an interrupt exit. The rules for what multitasking processing can be done in an interrupt exit are described in Chapter 15, "Suggestions for Server Writers," on page 287.

  The supported multitasking operations can be called from second-level interrupt handlers established through the HNDIO, HNDINT, HNDEXT, HNDIUCV, or CONSOLE services.

- Simulated MVS task management services do not exploit the native CMS multitasking facilities. The MVS ATTACH service is simulated as a LINK plus the creation of a simulated Task Control Block. They are not related to CMS dispatchable threads of control, but are kept as a single stack of MVS task levels.

Chapter 13. CMS Multitasking Function Descriptions

This chapter describes each of the CMS multitasking services. To request these services in your application program, you use procedure calls. CMS procedure calls are designed in such a way that each call performs a single, well-defined function. This results in a large number of calls compared to what would be seen in an equivalent assembler macro API design. However, each of these calls tends to be simple in itself, performing one function and requiring few parameters.

These services, also referred to as functions, can be grouped into three areas:

- General multitasking services
- Services to allow customization of the product
- Services to obtain diagnostic information.

The majority of the calls fall into the first category. Calls that fall primarily into one of the other two categories are identified as such in the function descriptions. In C programs, you call the multitasking functions just as any invocation of a function that returns *void*. In assembler programs, you use BALR 14,15 linkage to invoke the functions.

Notation Used in Parameter Descriptions

The description of each parameter for a function begins with the three-part notation:

(*usage*,*type*,*length*)

In this notation:

usage
is **input**, **output**, or **input/output**, indicating how the variable is used by the called function.

type
is **INT** or **CHAR**, indicating whether the variable contains binary integer or character type data. (All INT parameters are signed unless otherwise indicated.)

length
is the length of the variable, specified as one of the following:

- A number (such as **8**), a choice of two numbers (such as **0 or 9**), or a range of numbers (such as **1–8**), indicating the number of bytes or characters (depending on the data type) or the number of equal-length elements in an array

  Note: When a range is indicated for an output variable and you do not know the length of the value to be returned, you should set the variable to the maximum length to ensure that it will hold the complete returned value. Otherwise, if the returned value does not fit, you will receive an error indication. In that case, you will have to increase the length of the variable and call the function again.

- The name of another parameter variable (such as ***length1***) that specifies the number of bytes, characters, or elements.

Using the Online HELP Facility

You can receive online information about the CMS multitasking routines described in this book by using the z/VM HELP Facility. For example, to display a menu of CMS multitasking routines, enter:

```
help multitsk menu
```

To display information on a specific multitasking routine, (ThreadCreate in the following example), enter:

```
help routine threadcr
```

Because of the length of some of the routine names, typing the first eight characters of a routine's name may not provide help for the desired routine. For example, entering

```
help routine threadde
```

could mean you would like help for ThreadDelete or ThreadDelay. In this case, you can try an abbreviation for the routine name (ThreadDelay in this example) by entering:

```
help routine threaddy
```

Or get a list of all the routines in a particular group (all Thread routines in the following example) by entering:

```
help routine thread
```

For more information about using the HELP Facility, see the *z/VM: CMS User's Guide*. To display the main HELP Task Menu, enter:

```
help
```

For more information about the HELP command, see the *z/VM: CMS Commands and Utilities Reference* or enter:

```
help cms help
```

AbnormalEnd - Terminate a Process Abnormally

AbnormalEnd
errorcode
type
error_userdata_pointer
error_userdata_length

Purpose

Use the AbnormalEnd function to initiate abnormal termination processing.

Parameters

errorcode
(input/output,INT,4) is a variable for specifying an error completion code associated with this particular abend. A predefined set of codes used by the system are documented in the section on CMS abend codes in *z/VM: CMS and REXX/VM Messages and Codes*.

type
(input,INT,4) is a variable for specifying the type of error code that is defined either by the system or the user. Usually the caller specifies that the abend is of user type, unless the application needs to simulate a system abend. Valid values are as follows:

vm_abn_type_user
Error code is defined by the user.

vm_abn_type_system
Error code is defined by the system.

error_userdata_pointer
(input,INT,4) is a variable for specifying the address of a buffer containing optional user data, which gives information to any error event handlers about the abend. This may be used for specific information about I/O at the time of abends, save areas, parameter lists, or anything else defined by the user. The format of the information in this area is defined by the user and the data is provided by the caller of AbnormalEnd. If no user data is provided, this parameter should be set to zero.

error_userdata_length
(input,INT,4) is a variable for specifying the length of the user data contained in the buffer pointed to by *error_userdata_pointer*.

Usage Notes

1. AbnormalEnd uses Event Services to signal an error event, sending the error data mapped by the *vm_errevent* structure in the language binding file. A modifiable data area is provided for purposes of specifying recovery methods and modifying registers for recovery routines. The user may define an area containing additional information which is pointed to by *error_userdata_pointer*.
2. Information in the area pointed to by *error_userdata_pointer* can be in any format as defined by the user.
3. If the value specified for *type* is not valid, a type of *vm_abn_type_user* is assumed.
4. This function does not return to the caller unless an error event handler performed recovery and specified that execution should resume at the next instruction after the point of the abend. See Chapter 9, “Abend Services,” on page 67 for more information on error recovery.

5. If both a retry address and the next sequential instruction indicator are set by an error event handler, the next instruction indicator takes precedence and the retry routine is not invoked. An invalid next sequential instruction indicator value is ignored and recovery is determined by the retry address.
6. The other abend recovery mechanisms in CMS, the ABNEXIT service and the simulated MVS recovery services, do not allow for the multiple process, multiple thread environment. These services can still be used by multitasking programs but their behavior follows special rules intended to maintain compatibility with non-tasking behavior.

   The order in which abend recovery routines are invoked is determined by the class of abend. The following list defines the order for each class:

   - Program Check
     a. SPIE and ESPIE exits
     b. STAE and ESTAE exits
     c. VMERROR event handlers
     d. VMERRORCHILD event handlers
     e. ABNEXIT routines
   - MVS ABEND macro
     a. STAE and ESTAE exits
     b. VMERROR event handlers
     c. VMERRORCHILD event handlers
     d. ABNEXIT routines
   - DMSABN macro
     a. VMERROR event handlers
     b. VMERRORCHILD event handlers
     c. ABNEXIT routines
     d. STAE and ESTAE exits
   - AbnormalEnd call
     a. VMERROR event handlers
     b. VMERRORCHILD event handlers
     c. ABNEXIT routines
     d. STAE and ESTAE exits

   Only those STAE, ESTAE, SPIE, and ESPIE exits defined in the abending process are driven. ABNEXITs are driven regardless of the process that created them.
7. Portions of the event data key associated with the VMERROR event are binary data. See "Tips on Constructing Keys" on page 30 in *z/VM: CMS Application Multitasking* for a discussion of the use of binary data in event keys.

AccountControl — Define and Query Accounting Attributes

AccountControl
retcode
reascode
function
timer
num_types
accttypes
acct_settings

Purpose

Use the AccountControl function to initiate and query the collection of accounting information, alter accounting selectivity, or request immediate generation of accounting records created by CMS.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

function
(input,INT,4) is a variable for specifying the operation to be performed in response to the subsequent arrays. Valid values are as follows:

vm_act_query
The *accttypes* are queried and their setting is returned in the *acct_settings* array.

vm_act_set
The *accttypes* are set according to the values in the *acct_settings* array.

vm_act_sig
An account event is signaled for each accounting type currently set on, and then the values in the *accttypes* array are set according to the values in the *acct_settings* array.

timer
(input/output,INT,4) is a variable that contains the time interval for CMS to generate accounting records. The *timer* variable is either input or output, depending on the function:

- An input variable for the *vm_act_set* and *vm_act_sig* functions
- An output variable for the *vm_act_query* function.

The integer value is in milliseconds, in the range of 0 - 2,147,483,647 (maximum positive number of milliseconds that can be represented in 4 bytes). If the maximum time is used, this is equivalent to approximately 24.85 days. If 0 is specified for the *vm_act_set* or *vm_act_sig* functions, then accounting records are not automatically generated based on a time interval, but a user may explicitly request records by using the *vm_act_sig* function of AccountControl. For the *vm_act_query* function, the last time interval in effect is returned.

num_types
(input,INT,4) is a variable for specifying the number of account types to be set, signaled, or queried, (that is, the number of elements in the *accttype* and *acct_settings* arrays). The value must be greater than 0.

accttypes
(input/output,INT,*num_types*) is an array of 4-byte variables for specifying the account types to be set, signaled, or queried. The size of the array is specified by the *num_types* parameter. Valid values are as follows:

vm_act_all
All account types

vm_act_comm
Communication requests

vm_act_cpu
CPU utilization

acct_settings
(input/output,INT,*num_types*) is an array of 4-byte variables that specifies the settings corresponding to the *accttypes* array. This array is either input or output, depending on the setting of the *function* parameter:

- An input array for the *vm_act_set* and *vm_act_sig* operations
- An output array for the *vm_act_query* operation.

Valid values are as follows:

vm_act_off
Corresponding account type is set OFF

vm_act_on
Corresponding account type is set ON

vm_act_unchg
Corresponding account type is left UNCHANGED.

When accounting is set on for any of the account types, CMS will begin accumulating accounting data for that particular type. When an accounting record is requested by the *vm_act_sig* function or when a requested timer interval has expired, the accumulated data as well as any new data collected will be included in the accounting record that is signaled for the VMACCOUNT event.

Usage Notes

1. At initialization time, all account types are set off.
2. The *accttypes* and *acct_settings* arrays are processed in array order. For example, if values corresponding to *vm_act_all* and *vm_act_off* are the first elements in the respective arrays, all account types are set off before processing subsequent elements in the arrays. As a result, in general, an array element may be nullified by a subsequent array element.
3. The account type *vm_act_all* is invalid for the *vm_act_query* function, because individual elements may have been set after the *vm_act_all* setting. The *vm_act_all* setting is returned in the *acct_settings* array, but a warning return code indicates that individual account types should be queried.
4. If an invalid *accttype* is specified for the *vm_act_query* function, the corresponding *acct_settings* array will return a value of *vm_act_off* for that *accttype*, and a warning return code will be given. The rest of the array will be processed.
5. When the *vm_act_sig* function is specified, accounting data will be collected and signaled based on the accounting types that were previously set on and are currently in effect. After records are signaled, the *accttypes* and *acct_settings* arrays will be used to change any settings.
6. If the *timer* value is invalid, 0 will be assumed and a warning return code will be given. When all accounting is off, any *timer* value specified is ignored.
7. The AccountControl function uses Event Services to collect trace data and process it to produce accounting information.
8. Accounting information is accumulated and signaled based on account IDs.

9. The information being accounted for by CMS when accounting is set on for communication requests is the number of queue sends and number of queue receives that have occurred during an accounting period. These accumulated amounts include the various queue functions related to sending and receiving; for example, *queue receives* includes QueueReceiveBlock as well as QueueReceiveImmed. The format is defined in the VMCACT H and VMASMACT MACRO language binding files.
10. The information being accounted for by CMS for CPU utilization is the amount of CPU time utilized. The format is defined in the VMCACT H and VMASMACT MACRO language binding files.
11. Portions of the event data key associated with the VMACCOUNT event are binary data. See "Tips on Constructing Keys" on page 30 for a discussion of the use of binary data in event data keys.
12. An accounting setting cannot be set off except by the process that set it on. If multiple processes set on a setting, it is not actually set off until all the processes that had set it on set it off. When a process terminates, its modifications to the accounting settings are reset.

    If a time interval has been set by some process, another process can set the interval shorter but not longer. A time interval of zero (no automatic signalling) is considered to be the longest interval. When a process ends the time interval is not altered, unless no accounting settings remain on, in which case the accounting timer is cancelled.

    Since the interval can be shortened, a routine that handles the VMACCOUNT event should always assume that there could be multiple events bound to its event monitor and be prepared to process them in a loop.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_act_success	vm_act_success	AccountControl completed successfully
vm_act_error	vm_act_bad_func	*Function* is invalid
vm_act_error	vm_act_bad_numtype	*Num_types* is invalid
vm_act_warning	vm_act_array_bad_value	*Accttype* or *acct_settings* array value is invalid
vm_act_warning	vm_act_bad_time	*Timer* value invalid
vm_act_error	vm_act_insufficient_storage	No more storage available

Programming Language Bindings

Language	Language Binding File
C	VMCACT H
Assembler	VMASMACT MACRO
REXX	VMREXACT COPY

AccountIdentify — Identify an Accounting Entity

AccountIdentify
retcode
reascode
acctid
acctflg

Purpose

Use the AccountIdentify function to define an identifier to be associated with either an individual thread or all the threads in a process.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

acctid
(input,CHAR,16) is a variable for specifying the account identifier of the accounting data collected for this thread or process.

acctflg
(input,INT,4) is a variable for specifying whether the *acctid* applies only to this thread or to all the threads in this process. Valid values are as follows:

vm_act_id_thrd
ID for this thread only

vm_act_id_prc
ID for all the threads in this process

Usage Notes

1. If an account identifier has not been assigned to a process or a given thread, the accounting data collected for that thread has an account ID of 16 bytes of binary zeros.
2. If an account identifier is assigned to a process, any subsequently created threads will be assigned that identifier when they are created.
3. The account identifier is included in the header of an accounting record. See Chapter 8, “Accounting Services,” on page 65 for a description of the format of an accounting record.
4. It is suggested that the account ID not contain bytes whose values correspond to the code points for key wildcard characters. This is so that the account ID can be used easily in an event monitor sensitive to the VMACCOUNT or VMTRACE events. For more information, see “Tips on Constructing Keys” on page 30.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_act_success	vm_act_success	AccountIdentify completed successfully
vm_act_error	vm_act_bad_id_flag	*acctflg* is invalid

Programming Language Bindings

Language	Language Binding File
C	VMCACT H
Assembler	VMASMACT MACRO
REXX	VMREXACT COPY

CondVarCreate — Create a Condition Variable

CondVarCreate

retcode
reascode
condvariable_handle
condvariable_name
condvariable_name_length
mutex_handle

Purpose

Use the CondVarCreate function to establish a condition variable and associate it with a mutex.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

condvariable_handle
(output,INT,4) is a variable where the function returns the handle of the condition variable.

condvariable_name
(input,CHAR,*condvariable_name_length*) is a variable for specifying the name of the condition variable.

condvariable_name_length
(input,INT,4) is a variable for specifying the length of *condvariable_name*. It must be greater than 0 and less than 16MB in length.

mutex_handle
(input,INT,4) is a variable for specifying the handle of the mutex to be associated with the condition variable. This value is returned by the MutexCreate or MutexGetHandle function.

Usage Notes

1. The scope of access for a condition variable is the same as that defined by the associated mutex, which is defined when the mutex is created (by the MutexCreate function). This implies that a condition variable has either process or session level scope that is fixed for the life of the condition variable.
2. For uniqueness, the scope of a condition variable name is the mutex with which it is associated. This guarantees that a condition variable name is unique among all the condition variables of a mutex. For example, mutex M and mutex N in process P can each have a condition variable named C, even though they are different condition variables.
3. All condition variables created by a process are deleted when a process terminates. Any threads in other processes waiting on such condition variables are unblocked and given a return code indicating that the condition variable and mutex associated with this condition variable have been deleted.
4. The thread creating the condition variable must be in the process that created the associated mutex.
5. CMS supports up to 32,768 session-scope semaphores, mutexes, and condition variables, altogether. Also, for each process, CMS supports up to 32,768 process-scope semaphores, mutexes, and condition variables, altogether.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_syn_success	vm_syn_success	CondVarCreate completed successfully
vm_syn_error	vm_syn_insufficient_storage	Condition variable not created because storage is not available
vm_syn_error	vm_syn_handle_not_found	Mutex handle does not exist
vm_syn_error	vm_syn_bad_cnv_name_len	*Condvariable_name_length* is out of range.
vm_syn_error	vm_syn_not_mutex_creator	Condition variable not created because thread not in process that created mutex
vm_syn_error	vm_syn_cnv_already_exists	Condition variable not created because a condition variable of the same name already exists for this mutex
vm_syn_error	vm_syn_limit_reached	Selected scope's limit on total number of synchronization objects has been reached.

Programming Language Bindings

Language	Language Binding File
C	VMCSYN H
Assembler	VMASMSYN MACRO
REXX	VMREXSYN COPY

CondVarDelete — Delete a Condition Variable

CondVarDelete
retcode
reascode
condvariable_handle

Purpose

Use the CondVarDelete function to delete a condition variable from a mutex.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

condvariable_handle
(input,INT,4) is a variable for specifying the handle of the condition variable to be deleted. This value is returned by the CondVarCreate or CondVarGetHandle function.

Usage Notes

1. Two conditions must exist for a thread to delete a condition variable. They are:
   - Thread must be in the process in which the condition variable was created.
   - Thread must hold the mutex associated with the condition variable.

   If at least one of these conditions is not met, an error is returned.
2. If a condition variable is deleted and threads are waiting on this condition variable, the blocked threads are released and a return code is given to each thread indicating that the condition variable has been deleted.
3. All condition variables created by a process are deleted when a process terminates. Any threads in other processes waiting on such condition variables are unblocked and given a return code indicating that the condition variable has been deleted.
4. If a mutex is deleted, the condition variables associated with this mutex are deleted. The threads waiting on such condition variables are unblocked and given a return code indicating that the condition variable and the mutex have been deleted.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_syn_success	vm_syn_success	CondVarDelete completed successfully
vm_syn_error	vm_syn_handle_not_found	*Condvariable_handle* does not exist
vm_syn_error	vm_syn_not_condvar_creator	Condition variable is not deleted because process is not condition variable creator
vm_syn_error	vm_syn_mutex_not_held	Condition variable is not deleted because thread does not hold mutex

Programming Language Bindings

Language	Language Binding File
C	VMCSYN H
Assembler	VMASMSYN MACRO
REXX	VMREXSYN COPY

CondVarGetHandle — Get the Handle of a Condition Variable

CondVarGetHandle

retcode
reascode
condvariable_handle
condvariable_name
condvariable_name_length
mutex_handle

Purpose

Use the CondVarGetHandle function to get a handle for an existing condition variable.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

condvariable_handle
(output,INT,4) is a variable where the function returns the handle of the condition variable.

condvariable_name
(input,CHAR,*condvariable_name_length*) is a variable for specifying the name of the existing condition variable.

condvariable_name_length
(input,INT,4) is a variable for specifying the length of *condvariable_name*. It must be greater than 0 and less than 16MB in length.

mutex_handle
(input,INT,4) is a variable for specifying the handle of the mutex associated with the condition variable. This value is returned by the MutexCreate or MutexGetHandle function.

Usage Notes

1. A condition variable must be created by CondVarCreate before this function can get its handle. If the condition variable is not created, an error is returned.
2. If the threads using a condition variable share memory, the handle of a condition variable may be stored in the shared memory by the thread creating the condition variable. When the other threads in an application require the handle to manipulate the condition variable, it may be retrieved from the shared memory. However, if threads using a condition variable in an application do not share memory, the CondVarGetHandle function should be used to get the handle of the condition variable.
3. Condition variable handles are kept either per-process or per-session, depending on the level at which the mutex that is associated with the condition variable was created. The technique for sharing handles by storing them in shared memory will work only if the threads execute in the same scope, process or session, that the condition variable has as its level.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_syn_success	vm_syn_success	CondVarGetHandle completed successfully
vm_syn_error	vm_syn_name_not_found	*Condvariable_name* does not exist
vm_syn_error	vm_syn_bad_cnv_name_len	*Condvariable_name_length* is out of range
vm_syn_error	vm_syn_handle_not_found	Mutex handle does not exist

Programming Language Bindings

Language	Language Binding File
C	VMCSYN H
Assembler	VMASMSYN MACRO
REXX	VMREXSYN COPY

CondVarSignal — Signal a Condition Variable

CondVarSignal
retcode
reascode
condvariable_handle

Purpose

Use the CondVarSignal function to indicate that the condition represented by the condition variable is true. This results in one thread waiting on this variable being unblocked as soon as the associated mutex is also released.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

condvariable_handle
(input,INT,4) is a variable for specifying the handle of the condition variable to be signaled. This value is returned by the CondVarCreate or CondVarGetHandle function.

Usage Notes

1. If no threads are waiting on a condition variable and it is signaled, the condition represented by the condition variable stays at *true*. The next thread that checks the condition by issuing CondVarWait does not have to wait and may proceed to perform its operation.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_syn_success	vm_syn_success	CondVarSignal completed successfully
vm_syn_error	vm_syn_handle_not_found	*Condition_variable_handle* does not exist
vm_syn_error	vm_syn_mutex_not_held	Condition variable is not signaled because thread does not hold the mutex associated with this condition variable

Programming Language Bindings

Language	Language Binding File
C	VMCSYN H
Assembler	VMASMSYN MACRO
REXX	VMREXSYN COPY

CondVarWait — Wait on a Condition Variable

```
CondVarWait
        retcode
        reascode
        condvariable_handle
```

Purpose

Use the CondVarWait function to wait for a condition variable to be signaled. Upon entry to this function, the mutex associated with the condition variable must be held. The mutex is released while waiting for the condition variable to be signaled. When the condition variable is signaled, the mutex is reacquired.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

condvariable_handle
(input,INT,4) is a variable for specifying the handle of the condition variable to be waited on. This value is returned by the CondVarCreate or CondVarGetHandle function.

Usage Notes

1. If a thread waiting on a condition variable is terminated, the thread is removed from the wait list.
2. If a condition variable is deleted, any thread waiting on that variable is given an error return code when it resumes running. The mutex is not reacquired in this case.
3. When a thread resumes running after CondVarWait completes successfully, it is guaranteed that the condition is still true and the thread may proceed to perform its operation.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_syn_success	vm_syn_success	CondVarWait completed successfully
vm_syn_error	vm_syn_handle_not_found	*Condvariable_handle* does not exist
vm_syn_error	vm_syn_cnv_deleted	Condition variable was deleted
vm_syn_error	vm_syn_cnv_mutex_deleted	Condition variable and mutex were deleted
vm_syn_error	vm_syn_mutex_not_reacquired	The mutex associated with this condition variable could not be reacquired.
vm_syn_error	vm_syn_mutex_not_held	Condition variable is not waited on because thread does not hold the mutex associated with this condition variable
vm_syn_warning	vm_syn_indeterminate_state	Resource protected by mutex may be in an indeterminate state

Programming Language Bindings

Language	Language Binding File
C	VMCSYN H
Assembler	VMASMSYN MACRO
REXX	VMREXSYN COPY

DateTimeGet — Query Time and Date

DateTimeGet

retcode
reascode
format
date
time
zone
epoch

Purpose

Use the DateTimeGet function to return the date, time, time zone, and epoch time (the number of seconds since the system's standard epoch). The date and time formats for the United States, Europe, and the International Standards Organization (ISO) are supported.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

format
(input,INT,4) is a variable for specifying the format in which the date and time should be represented. Values are as follows:

Variable	Format	Date	Time
vm_tmr_format_usa	United States	MM/DD/YYYY	HH:MM:SS
vm_tmr_format_eur	European	DD.MM.YYYY	HH.MM.SS
vm_tmr_format_iso	ISO	YYYY-MM-DD	HH.MM.SS

date
(output,CHAR,10) is a variable where the function returns the date in the format specified on the *format* parameter.

time
(output,CHAR,8) is a variable where the function returns the time, in 24-hour clock notation, in the format specified on the *format* parameter.

zone
(output,CHAR,3) is a variable where the function returns the time zone (as returned by CP QUERY TIME, for example, EDT or PST).

epoch
(output,INT,4) is a unsigned variable where the function returns the number of seconds since the standard epoch (as computed from the value returned by the STORE CLOCK (STCK) instruction).

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_tmr_success	vm_tmr_success	DateTimeGet. completed successfully
vm_tmr_error	vm_tmr_cpqtime_failed	CP QUERY TIME failed
vm_tmr_error	vm_tmr_format_invalid	Invalid *format* specification

Programming Language Bindings

Language	Language Binding File
C	VMCTMR H
Assembler	VMASMTMR MACRO
REXX	VMREXTMR COPY

DateTimeSubtract -- Compute Time Differences

DateTimeSubtract

retcode
reascode
minuend_stamp
minuend_stamp_length
minuend_stamp_format
minuend_stamp_bias
minuend_stamp_window_type
minuend_stamp_window_position
subtrahend_stamp
subtrahend_stamp_length
subtrahend_stamp_format
subtrahend_stamp_bias
subtrahend_stamp_window_type
subtrahend_stamp_window_position
difference_stamp_buffer
difference_stamp_buffer_size
difference_stamp_length
difference_stamp_format
difference_stamp_bias
difference_stamp_window_type
difference_stamp_window_position

Purpose

Use the DateTimeSubtract function to compute differences of times. DateTimeSubtract performs time format conversions and time zone conversions as part of the calculation process.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

minuend_stamp
(input,CHAR,*minuend_stamp_length*) is a variable for specifying the minuend stamp. The minuend is the first term in the time subtraction expression.

minuend_stamp_length
(input,INT,4) is a variable for specifying the length of *minuend_stamp*.

minuend_stamp_format
(input,INT,4) is a variable for specifying a value that indicates the format of *minuend_stamp*.

minuend_stamp_bias
(input,INT,4) is a variable for specifying a bias value to be applied to *minuend_stamp* as part of the time calculation.

minuend_stamp_window_type
(input,INT,4) is a variable for specifying a value that indicates the type of window for the minuend year. The values are:

vm_tmr_window_fixed
0

vm_tmr_window_sliding
1

minuend_stamp_window_position
(input,INT,4) is a variable for specifying a value that indicates the position of the window for the minuend year.

subtrahend_stamp
(input,CHAR,*subtrahend_stamp_length*) is a variable for specifying the subtrahend stamp. The subtrahend is the term to be subtracted from the minuend in the time subtraction expression.

subtrahend_stamp_length
(input,INT,4) is a variable for specifying the length of *subtrahend_stamp*.

subtrahend_stamp_format
(input,INT,4) is a variable for specifying a value that indicates the format of *subtrahend_stamp*.

subtrahend_stamp_bias
(input,INT,4) is a variable for specifying a bias value to be applied to *subtrahend_stamp* as part of the time calculation.

subtrahend_stamp_window_type
(input,INT,4) is a variable for specifying a value that indicates the type of window for the subtrahend year. The values are:

vm_tmr_window_fixed
0

vm_tmr_window_sliding
1

subtrahend_stamp_window_position
(input,INT,4) is a variable for specifying a value that indicates the position of the window for the subtrahend year.

difference_stamp_buffer
(output,CHAR,*difference_stamp_buffer_size*) is a variable where the function stores the difference stamp, which is the value that results from the time calculation.

difference_stamp_buffer_size
(input,INT,4) is a variable for specifying the size in bytes of *difference_stamp_buffer*.

difference_stamp_length
(output,INT,4) is a variable where the function stores the length of the difference stamp.

difference_stamp_format
(input,INT,4) is a variable for specifying a value that indicates the format which the difference stamp should be expressed.

difference_stamp_bias
(input,INT,4) is a variable for specifying a bias value to be applied to the difference stamp after the time calculation is complete.

difference_stamp_window_type
(input,INT,4) is a variable for specifying a value that indicates the type of window for the difference year. The values are:

vm_tmr_window_fixed
0

vm_tmr_window_sliding
1

vm_tmr_window_none
2

difference_stamp_window_position
(input,INT,4) is a variable for specifying a value that indicates the position of the window which the difference year is expected to reside.

Usage Notes

General Information

1. In a subtraction expression, the *minuend* is the first term, the *subtrahend* is the second term, and the *difference* is the result. For example, in the expression `5-3=2`, 5 is the minuend, 3 is the subtrahend, and 2 is the difference.
2. Leap seconds are not accounted for in any of DateTimeSubtract's calculations.
3. DateTimeSubtract supports all the REXX DATE() formats except for `Century`, `Days`, `Month`, and `Weekday`. CP, CMS, CSL, CMS Pipelines, and certain other selected formats are also supported.

   For more information on a related routine that performs operations, see the DATECONVERT stage in *z/VM: CMS Pipelines User's Guide and Reference*.
4. For examples of using DateTimeSubtract, see “DateTimeSubtract Examples” on page 60.

Formats

1. The minuend, subtrahend, and difference can each be expressed or requested in a variety of formats. Certain formats are *absolute*; that is, they accommodate a reference to a specific *moment* in time. Certain other formats are *relative*; that is, they accommodate a reference to a certain *amount* of time. Input parameters *minuend_stamp_format*, *subtrahend_stamp_format*, and *difference_stamp_format* accept format identifiers as identified in Table 18 on page 117.

   The abbreviations used in the table heading have these meanings:

   ID
   Format identifier. All format IDs begin with *vm_tmr_format*.

   RT
   Record type: fixed or variable.

   SL
   Stamp length.

   FT
   Format type: absolute or relative.

   Input stamps must comply with the syntax rules given in Table 18 on page 117, in particular:

   - Extraneous blanks (leading, trailing, or intermediate) are not accepted. For example, a stamp of format *vm_tmr_format_iso* must have exactly one space between the date and the time; multiple spaces will produce a conversion error.
   - If a format does not accept leading zeroes in a field, supplying leading zeroes in that field will result in a conversion error. For example, format *vm_tmr_format_rexx_date_n* requires January 1, 1997 to be expressed as `1 Jan 1997`, not `01 Jan 1997`.

Table 18. Formats and Format Identifiers

ID	Syntax	RT	SL	FT
_csl	*yyyyyyy/mm/dd hh:mm:ss.uuuuuu*	V	29	A
_db2	*yyyyyyy-mm-dd-hh.mm.ss.uuuuuu* This is the date format used by DB2® for VM and VSE.	V	29	A
_eur	*dd.mm.yyyyyyy hh:mm:ss.uuuuuu*	V	29	A

Table 18. Formats and Format Identifiers (continued)

ID	Syntax	RT	SL	FT
_iso	*yyyyyyy-mm-dd hh:mm:ss.uuuuuu*	V	29	A
_julian	*yyyyyyy.ddd hh:mm:ss.uuuuuu*	V	27	A
_normal	*dd mmm yyyyyyy hh:mm:ss.uuuuuu* Where: ***dd*** specifies the day of the month. Leading zeroes in the day field are accepted on input and will be supplied on output. ***mmm*** specifies the three-letter text month. The accepted values are: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec Case is not significant. ***yyyyyyy*** specifies the one- to seven-digit year.	V	30	A
_rexx_date_e_long	*dd/mm/yyyyyyy hh:mm:ss.uuuuuu*	V	29	A
_usa	*mm/dd/yyyyyyy hh:mm:ss.uuuuuu*	V	29	A
_rexx_date_j_long	*yyyyddd hh:mm:ss.uuuuuu*	V	23	A
_pipe *_rexx_date_s*	*yyyymmddhhmmssuuuuuu*	V	20	A
_rexx_date_n	*dd mmm yyyy hh:mm:ss.uuuuuu* Where: ***dd*** specifies the day of the month. The day does not accept leading zeroes on input and does not produce leading zeroes on output. ***mmm*** specifies the month. The accepted values for the month are the ones defined for format *_normal*, with the additional restriction that case is significant. ***yyyy*** specifies the four-digit year.	V	27	A
_csl_short *_rexx_date_o*	*yy/mm/dd hh:mm:ss.uuuuuu*	V	24	A
_db2_short	*yy-mm-dd-hh.mm.ss.uuuuuu*	V	24	A

Table 18. Formats and Format Identifiers (continued)

ID	Syntax	RT	SL	FT
_eur_short	*dd.mm.yy hh:mm:ss.uuuuuu*	V	24	A
_iso_short	*yy-mm-dd hh:mm:ss.uuuuuu*	V	24	A
_julian_short	*yy.ddd hh:mm:ss.uuuuuu*	V	22	A
_pipe_short	*yymmddhhmmssuuuuuu*	V	18	A
_rexx_date_e	*dd/mm/yy hh:mm:ss.uuuuuu*	V	24	A
_rexx_date_j	*yyddd hh:mm:ss.uuuuuu*	V	21	A
_rexx_date_n_short	*dd mmm yy hh:mm:ss.uuuuuu* Where: ***dd*** specifies the day of the month. ***mmm*** specifies the month. ***yy*** specifies the two-digit year. The *dd* and *mmm* fields are as defined for format *_rexx_date_n*.	V	25	A
_usa_short *_rexx_date_u*	*mm/dd/yy hh:mm:ss.uuuuuu*	V	24	A
_rexx_date_b	*dddddddddd hh:mm:ss.uuuuuu*	V	26	A
_scientific_absolute	Eight-byte value containing two signed four-byte binary integers. The first integer is the number of whole days (that is, 24-hour units) that have elapsed between January 1, 4713 BC, noon, Coordinated Universal Time (UTC) and the moment being expressed. (This is called the *Julian day number*.) The second integer is the number of milliseconds to be added to the Julian day number to reach the moment being expressed. The first integer must be greater than or equal to zero. The second integer must be greater than or equal to zero and less than the number of milliseconds in a day (86400000).	F	8	A

Table 18. Formats and Format Identifiers (continued)

ID	Syntax	RT	SL	FT
_tod_absolute	Unsigned doubleword indicating the number of TOD clock units that have elapsed between the standard epoch and the moment being expressed. A *TOD clock unit* is a unit of time, just as minutes, seconds, and hours are units of time; there are 4096 TOD clock units in a microsecond. The standard epoch is the moment at which the TOD clock would have read X'00000000 00000000'. On z/VM systems the standard epoch is January 1, AD 1900, 00:00:00 UTC. This format is exactly the format of the output of the STORE CLOCK (STCK) instruction.	F	8	A
_posix	Unsigned doubleword indicating the number of seconds since the POSIX epoch (this is defined according to the IEEE Standard 1003.1). The POSIX epoch is January 1, AD 1970, 00:00:00 UTC.	F	8	A
_met	*[-]ddddddddd/hh:mm:ss.uuuuuu*	V	27	R
_hms	*hhhhhhhhh:mm:ss.uuuuuu*	V	23	R
_rexx_time_l	*hh:mm:ss.uuuuuu*	V	15	R
_rexx_time_e	*ssssssssss.uuuuuu*	V	17	R
_scientific_relative	Eight-byte value containing two signed four-byte binary integers and specifying an *amount* of time rather than a specific date. The first integer is a number of whole days (that is, 24-hour units). The second integer represents a fractional day and is the number of milliseconds to be added to the whole day count. There are no constraints on the value of the first integer. The second integer must be greater than or equal to zero and less than the number of milliseconds in a day (86400000).	F	8	R
_tod_relative	Signed doubleword specifying an *amount* of time rather than a specific date. Specified in TOD clock units.	F	8	R

2. For fixed-length formats, the minuend and subtrahend stamps must be supplied with at least the length shown in Table 18 on page 117. Also, the difference buffer must be at least as long as the length shown, and the produced stamp will have exactly the length shown.
3. For varying-length formats, the minuend and subtrahend can be supplied with any positive length. The caller can omit any part of the *hh:mm:ss.uuuuuu* portion of the stamp; however, if any field is omitted, all fields to the right of it must also be omitted. Omitted fields are taken as zero.
4. When the difference is to be expressed in a varying-length format, as much of the computed difference as can fit is returned in the difference buffer, subject to the constraint that the last integer field supplied in the difference buffer is never truncated to fit. If the entire difference stamp cannot

be placed in the difference buffer, a warning is returned. The length of a full-length difference is shown in Table 18 on page 117.

5. For the string formats (such as *vm_tmr_format_usa_short*), a minuend or subtrahend can be specified with leading zeroes omitted. For example, `4/2/1995 0:0:0` is accepted input for the format *vm_tmr_format_usa*.
6. For the string formats, DateTimeSubtract usually produces a difference that contains the appropriate number of leading zeroes in each field. The exceptions are these fields, which never contain leading zeroes on output:
   - A seven-digit year field
   - The day fields of formats *vm_tmr_format_rexx_date_n* and *vm_tmr_format_rexx_date_n_short*
   - The day fields of *vm_tmr_format_met* and *vm_tmr_format_rexx_date_b*
   - The hours field of *vm_tmr_format_hms*
   - The seconds field of *vm_tmr_format_rexx_time_e*
7. For all of the following fields, the caller may supply up to ten digits of information, provided the number supplied fits without overflow in a signed fullword:
   - Day fields of *vm_tmr_format_rexx_date_b* and *vm_tmr_format_met*
   - Hours field of *vm_tmr_format_hms*
   - Seconds field of *vm_tmr_format_rexx_time_e*
8. Table 19 on page 121 gives the lower and upper bounds for each supported format.

Table 19. Format Limits

Date Format All format IDs begin with *vm_tmr_format*	**Lower Bound**	**Upper Bound**
DateTimeSubtract's internal limits (absolute formats)	1 Jan 4713 BC 00:00:00 UTC	4 Jun AD 5874898 00:00:00 UTC minus one TOD unit
_scientific_absolute	X'00000000 00000000' 1 Jan 4713 BC 12:00:00 UTC	X'7FFFFFFF 02932DFF' 3 Jun AD 5874898 23:59:59.999 UTC
_tod_absolute	X'00000000 00000000' 1 Jan AD 1900 00:00:00 UTC	X'FFFFFFFF FFFFFFFF' 17 Sep AD 2042 23:53:47.370495 UTC plus 4095 TOD units
_posix	X'00000000 00000000' 1 Jan AD 1970 00:00:00 UTC	X'0000A88E E75BEDFF' 3 Jun AD 5874898 23:59:59 UTC
Absolute string formats containing seven-digit year fields	1 Jan AD 0001 00:00:00 UTC	31 Dec AD 5873999 23:59:59.999999 UTC
Absolute string formats containing four-digit year fields	1 Jan AD 0001 00:00:00 UTC	31 Dec AD 9999 23:59:59.999999 UTC
Absolute string formats containing two-digit year fields **Note:** You must use a window to achieve these limits.	1 Jan AD 0001 00:00:00 UTC	31 Dec AD 5873999 23:59:59.999999 UTC
_rexx_date_b	639796 00:00:00 UTC 14 Sep AD 1752	2145762221 23:59:59.999999 3 Jun AD 5874898
DateTimeSubtract's internal limits (relative formats)	-2147483648 days	2147483648 days minus one TOD unit

Table 19. Format Limits (continued)

Date Format All format IDs begin with *vm_tmr_format*	Lower Bound	Upper Bound
_tod_relative	X'80000000 00000000' -(2**51) microseconds (approximately -26062.5 days)	X'7FFFFFFF FFFFFFFF' 2**51 microseconds minus one TOD unit (approximately 26062.5 days)
_scientific_relative	X'80000000 00000000' -2147483648 days	X'7FFFFFFF 05265BFF' 2147483647 days plus 23:59:59.999
_met	-2147483648 00:00:00.000000	2147483647 23:59:59.999999
_rexx_time_l	00:00:00.000000	99:59:59.999999
_rexx_time_e	0000000000.000000	2147483647.999999
_hms	0:00:00.000000	2147483647:59:59.999999

9. The accepted formats for the subtrahend stamp and difference stamp are a function of the format in which the minuend stamp is expressed. In other words, some combinations of time formats are meaningless in the context of time arithmetic. For example, the expression "23 days - (minus) April 15, 1936" means nothing. Table 20 on page 122 illustrates the combinations of relative and absolute formats that are accepted.

   Table 20. Supported Combinations of Formats

Minuend	Subtrahend	Difference
Absolute	Absolute	Relative
Absolute	Relative	Absolute
Relative	Relative	Relative

   Attempts to use any other combinations of absolute and relative formats result in an error return and reason code being produced and no arithmetic being performed.

10. The only format capable of expressing a BC year is *vm_tmr_format_scientific_absolute.* An attempt to place a BC year in any other format results in a conversion error being returned.

Using Bias

1. Nearly all of the absolute formats express dates and times relative to a certain time zone. For example, format *vm_tmr_format_iso* gives us a date and a time, but it still fails to express an **exact moment** in time because it doesn't tell us the time zone with respect to which the date and time should be interpreted. To see this ambiguity more clearly, note, for example, that the moment "1997-11-03 00:00:00" occurs one hour later in Chicago than it does in Washington.

   To overcome this, DateTimeSubtract requires most absolute formats to be qualified with an integer, called the *bias value*. The bias value is expressed in seconds and reflects the difference between the time zone of the stamp and Coordinated Universal Time (UTC); negative values are west of UTC, and positive values are east of UTC. For example, if *minuend_stamp* is meant to express an Eastern Standard Time (EST) stamp, then you should set *minuend_stamp_bias* to `-18000` (which is -5*60*60), because EST is five hours west of UTC. Similarly, if you want DateTimeSubtract to produce the difference stamp in Pacific Standard Time, then you should set *difference_stamp_bias* to `-28800` (which is -8*60*60).

   Whenever DateTimeSubtract requires you to supply a bias value, you may set the bias value to constant *vm_tmr_bias_local* to cause DateTimeSubtract to use the bias value for the time zone of your system. This feature gives you an easy way to work with stamps expressed in local time. When you use

this feature, DateTimeSubtract uses the time zone offset returned by DIAG X'00' as the stamp's bias value.

Of course, the notion of bias value does not apply to relative formats. Further, certain absolute formats are defined in such a way that time zone phenomena do not come into play. To summarize, bias value is meaningless for all of the following formats:

- Any relative format (such as *vm_tmr_format_met*)
- Format *vm_tmr_format_tod_absolute*
- Format *vm_tmr_format_scientific_absolute*
- Format *vm_tmr_format_posix*

Using a Window

1. To cope with two-digit years, DateTimeSubtract supports two different window schemes. These techniques let the caller who provides a two-digit input year (such as 12/25/93) specify an additional numeric value from which DateTimeSubtract can deduce the year's century digits. The two techniques are *fixed window* and *sliding window*.

   - **Fixed Window Technique:** In this technique, a given two-digit year is assumed to reside in the 100-year span [*by*,*by*+99], where integer *by*, called the *base year*, is specified separately by the caller. For example, if the two-digit year 93 is interpreted according to *by*=1995, the window is [1995,2094], and 93 is taken to mean 2093. However, if *by*=1970, then the window is [1970,2069], and 93 is taken to mean 1993.
   - **Sliding Window Technique:** In this technique, a given two-digit year is assumed to reside in the 100-year span [*cy+n*,*cy+n*+99], where integer *cy* is the current year—that is, the year at the moment of the call—and *n* is a constant added to the current year to indicate the first year of the window. For example, if the two-digit year 93 is interpreted according to *cy*=1995 and *n*=-10, the window is [1985,2084], and 93 is taken to mean 1993. However, if *cy*=1995 and *n*=0, then the window is [1995,2094], and 93 is taken to mean 2093.

     To support a sliding window, DateTimeSubtract samples the current year by performing a STORE CLOCK (STCK) instruction and then calculating the current year from the STCK result. There are some considerations to be remembered for this calculation, namely:

     - The "current year" computed is the current year of the zone of the stamp being manipulated, *not* the current year of the zone which the calculation is occurring and *not* the current year of UTC. For example, if on January 1, 1996, at 00:00:01 Eastern Standard Time (EST) a DateTimeSubtract function in Washington, DC runs the sliding window algorithm for evaluating a two-digit-year Pacific Standard Time (PST) stamp, the current year is taken to be 1995, *not* 1996, because the stamp being manipulated is a PST stamp and at that moment the current year in PST is still 1995.
     - The sliding window computation works correctly only if your system's hardware TOD clock epoch is January 1, AD 1900, 00:00:00 UTC. If your system's hardware TOD clock is set for some other epoch (for example, if your hardware TOD clock is set to local time instead of UTC), then the two-digit-year window calculations DateTimeSubtract performs might produce incorrect results under some conditions.

   In calls to DateTimeSubtract, the window for each stamp (minuend, subtrahend, and difference) is characterized separately, using the stamp's *_window_type* and *_window_position* parameters, as follows:

   - To cause DateTimeSubtract to use the fixed-window technique to interpret a stamp with a two-digit year:
     a. Set the stamp's *_window_type* parameter equal to constant *vm_tmr_window_fixed*.
     b. Set the stamp's *_window_position* parameter equal to the base year.
   - To cause DateTimeSubtract to use the sliding window technique to interpret a stamp with a two-digit year:
     a. Set the stamp's *_window_type* parameter equal to constant *vm_tmr_window_sliding*.

b. Set the stamp's *_window_position* parameter equal to the offset of the window's start from the current year.

   For example, to cause a two-digit-year subtrahend to be interpreted according to the sliding window [*cy*-50,*cy*+49], set *subtrahend_stamp_window* to `-50`.

No matter whether you use the fixed-window or sliding-window technique, be careful to ensure that the 100-year window you are characterizing resides entirely within AD years. In other words, the beginning year of your 100-year interval must be greater than zero. Also, the beginning year must be less than or equal to 5873900. DateTimeSubtract will return a conversion error if these conditions are not met.

When the caller wishes the difference to be expressed in two-digit-year notation, several choices are available:

- The caller can use *difference_stamp_window_type* and *difference_stamp_window_position* to specify a 100-year fixed or sliding window, the same way as for a two-digit-year input. Here, after the calculation is complete, DateTimeSubtract examines the difference year and decides whether the difference year lies within the 100-year window expressed by the caller. If the year is within the window, then DateTimeSubtract encodes the output string with a two-digit year. Otherwise, DateTimeSubtract returns an error indicating that the difference year does not lie within the caller's difference window.
- If the caller does not wish to specify a window for the difference, but wants instead just to have DateTimeSubtract arbitrarily leave out the century digits on the year portion of the difference stamp, then the caller can specify *vm_tmr_window_none* in parameter *difference_stamp_window_type*. For this case, there is no "difference window" per se, any input supplied in parameter *difference_stamp_window_position* is ignored, and DateTimeSubtract just leaves out the century digits when it builds the difference stamp.

When the difference is to be expressed using a fixed or sliding window, the 100-year window you characterize must reside entirely within AD years. In other words, the beginning year of the 100-year interval must be greater than zero. Also, the beginning year must be less than or equal to 5873900. DateTimeSubtract will return a conversion error if this condition is not met.

If a stamp is expressed in a non-two-digit-year format, then DateTimeSubtract ignores the stamp's *_window_type* and *_window_position* parameters.

Converting Formats and Time Zones

To convert the format or time zone of a stamp without doing any time calculation, just specify the time to be converted as the minuend and:

- specify a relative-format subtrahend of zero, or
- set *subtrahend_stamp_length* to zero.

DateTimeSubtract produces the converted stamp as the difference.

Computing a Time Sum

DateTimeSubtract can compute a time sum if you specify a negative relative-format subtrahend. This works only for subtrahend formats *vm_tmr_format_scientific_relative*, *vm_tmr_format_tod_relative*, and *vm_tmr_format_met*. To specify a negative amount in format:

- *vm_tmr_format_scientific_relative*, make the first word less than zero. DateTimeSubtract adds the two's-complement of the first word to the minuend as whole days, then adds the two's-complement of the second word as milliseconds to that interim sum.
- *vm_tmr_format_tod_relative*, just use the conventional two's-complement notation for signed doublewords.
- *vm_tmr_format_met*, just prefix the amount with a minus sign.

Precision of Arithmetic

When performing arithmetic, DateTimeSubtract converts each input to an internal form capable of representing time stamps to a resolution of one TOD clock unit. The arithmetic is performed on this

internal form and the computed difference is then converted to the desired output. If the output format is not capable of representing time information to TOD-clock-unit precision, then the difference is rounded to the nearest increment expressible in the format which the difference was requested. For example, if the output is requested in one of the scientific time formats, the difference is rounded to the nearest millisecond.

Julian and Gregorian Calendars

DateTimeSubtract assumes the switch from the Julian calendar to the Gregorian calendar occurred on September 14, 1752 (Gregorian). In other words,

- All dates labeled September 2, 1752 and earlier are reckoned using the Julian calendar.
- All dates labeled September 14, 1752 and later are reckoned using the Gregorian calendar.
- The day immediately after the day labeled September 2, 1752 is labeled September 14, 1752.

The Gregorian calendar was instituted by Pope Gregory in the year 1582 to bring the calendar back into alignment with astronomical phenomena (the equinox and solstice). It achieved the realignment by taking two actions:

- The definition of a leap year was changed. In the Julian calendar, every year divisible by 4 is a leap year; this rule had been injecting too many leap days, causing the calendar to fall behind. In the Gregorian calendar, the Julian leap year rule applies, except that a year divisible by 100 must also be divisible by 400 to be a leap year.
- To bring the calendar date back into alignment with the earth (in other words, to compensate for the excess number of leap days inserted by the Julian rule), ten days were omitted. The first date of the Gregorian calendar is October 15, 1582, and that day corresponds to October 5, 1582 on the Julian calendar. In other words, the day after October 4, 1582 was October 15, 1582 according to Pope Gregory.

The Gregorian calendar was adopted at different times in different parts of the world, some as late as 1923. Depending upon when a municipality adopted the Gregorian calendar, the number of days needing to be dropped differed. Pope Gregory had to drop only 10 days when he switched his calendar in 1582, but by 1752, due to an extra Julian leap day having been inserted in 1700, it was necessary to drop 11 days when changing from the Julian calendar to the Gregorian calendar.

On the calendar used by DateTimeSubtract, the days are labeled like this:

```
   ⋮
August 31, 1752
September 1, 1752
September 2, 1752
September 14, 1752
September 15, 1752
September 16, 1752
   ⋮
```

Notice that the day immediately after September 2, 1752 is September 14, 1752. The dates September 3, 1752 through September 13, 1752 simply do not exist, and DateTimeSubtract never produces them as output.

However, it is possible that the caller might supply one of these dates on input. DateTimeSubtract does not reject these dates as not valid input, but rather it interprets them according to Gregorian rules. For example, if you specify September 13, 1752 as input, DateTimeSubtract reckons it as the day before September 14, 1752, namely, September 2, 1752 on the DateTimeSubtract calendar.

Table 21 on page 125 summarizes DateTimeSubtract's reckoning of the dates in this period.

Table 21. DateTimeSubtract's Reckoning Dates

Input Date	DateTimeSubtract's Reckoning
September 13, 1752	September 2, 1752

Table 21. DateTimeSubtract's Reckoning Dates (continued)

Input Date	DateTimeSubtract's Reckoning
September 12, 1752	September 1, 1752
September 11, 1752	August 31, 1752
September 10, 1752	August 30, 1752
September 9, 1752	August 29, 1752
September 8, 1752	August 28, 1752
September 7, 1752	August 27, 1752
September 6, 1752	August 26, 1752
September 5, 1752	August 25, 1752
September 4, 1752	August 24, 1752
September 3, 1752	August 23, 1752

If you perform subtractions using September 3, 1752 through September 13, 1752 as one of the inputs, the output you get might not be what you expect. For example, all of these statements are true:

September 8, 1752 - 1 day = August 27, 1752
September 6, 1752 - 3 days = August 23, 1752
September 5, 1752 - August 25, 1752 = 0 days

Be careful of this property when manipulating dates that do not exist on DateTimeSubtract's calendar.

In addition, callers using format *vm_tmr_format_rexx_date_b* need to remember that the REXX function DATE('B') assigns *all* its day numbers, including those on or before September 2, 1752, using Gregorian calendar rules. Because of this, DateTimeSubtract considers 639,796—the DATE('B') value corresponding to Gregorian date September 14, 1752—to be the minimum accepted value in dates using format *vm_tmr_format_rexx_date_b*. Attempts to use values less than this threshold result in a format or conversion error being returned.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_tmr_success	vm_tmr_success	DateTimeSubtract completed successfully.
vm_tmr_warning	vm_tmr_dif_truncated	The computed difference stamp was truncated to fit into *difference_stamp_buffer*.
vm_tmr_error	vm_tmr_bad_min_format	Input parameter *minuend_stamp_format* contains an unrecognized value.
vm_tmr_error	vm_tmr_bad_sub_format	Input parameter *subtrahend_stamp_format* contains an unrecognized value.
vm_tmr_error	vm_tmr_bad_dif_format	Input parameter *difference_stamp_format* contains an unrecognized value.
vm_tmr_error	vm_tmr_bad_format_combination	Arithmetic is not possible on operands whose formats are the ones you specified.
vm_tmr_error	vm_tmr_bad_min_window_type	Parameter *minuend_stamp_window_type* contains an unrecognized value.
vm_tmr_error	vm_tmr_bad_sub_window_type	Parameter *subtrahend_stamp_window_type* contains an unrecognized value.

Return Code	Reason Code	Meaning
vm_tmr_error	vm_tmr_bad_dif_window_type	Parameter *difference_stamp_window_type* contains an unrecognized value.
vm_tmr_error	vm_tmr_min_conversion_error	The minuend contains syntax or specification error and therefore could not be converted to the form necessary for arithmetic.
vm_tmr_error	vm_tmr_sub_conversion_error	The subtrahend contains syntax or specification error and therefore could not be converted to the form necessary for arithmetic.
vm_tmr_error	vm_tmr_dif_conversion_error	The difference was computed but could not be expressed in the format requested.
vm_tmr_error	vm_tmr_bad_min_length	The value for *minuend_stamp_length* is inappropriate for the given value of *minuend_stamp_format*.
vm_tmr_error	vm_tmr_bad_sub_length	The value for *subtrahend_stamp_length* is inappropriate for the given value of *subtrahend_stamp_format*.
vm_tmr_error	vm_tmr_bad_dif_length	The value for *difference_stamp_buf_size* is inappropriate for the given value of *difference_stamp_format*.
vm_tmr_error	vm_tmr_dif_wraparound	The arithmetic was performed but resulted in either an arithmetic overflow or an arithmetic underflow. A meaningful difference could not be formed.

Programming Language Bindings

Language	Language Binding File
C	VMCTMR H
Assembler	VMASMTMR MACRO
REXX	VMREXTMR COPY

EventCreate — Create an Event Definition

EventCreate

retcode
reascode
event_name
event_name_length
event_flag
event_flag_size
loose_signal_limit
signal_timeout_period

Purpose

Use the EventCreate function to register the name of an event and specify how that event is to be managed.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

event_name
(input,CHAR,*event_name_length*) is a variable for specifying the name of the event being defined.

event_name_length
(input,INT,4) is a variable for specifying the length of *event_name*.

event_flag
(input,INT,*event_flag_size*) is an array of 4-byte variables, each element of which contains information about how the event is to be managed. Only one option from each of the following sets may be specified. If no option from a particular set is specified, the default is taken.

- Scope of the event name

  vm_evn_process_scope
  Process (the default) — only this process can monitor this event and only this process can signal this event.

  vm_evn_session_scope
  Session — all processes in the session can both monitor and signal this event.

- Manner in which an event signal is delivered to multiple event monitors in a process

  vm_evn_broadcast_signals
  Broadcast (the default) — the signal is simultaneously delivered to all qualifying monitors.

  vm_evn_fifo_signals
  FIFO sequence — the signal is delivered to one qualifying monitor at a time, in the order the monitors were created.

  vm_evn_lifo_signals
  LIFO sequence — the signal is delivered to one qualifying monitor at a time, in the inverse order of their creation.

- The treatment of the signaler

vm_evn_async_signals
The signaling thread is allowed to continue executing (the default).

vm_evn_sync_thread_signals
The signaling thread is suspended until signal processing is complete. Signal processing in a process is considered complete when all qualifying monitors have completed processing of the signal or, if there are no qualifying monitors, the signal has been discarded as a result of being the oldest loose signal when the loose signal limit was exceeded.

A monitor is considered to have completed processing a signal when that signal has become part of the current signal set of the monitor and that monitor has subsequently been reset. For FIFO and LIFO events, a bound signal may be explicitly discarded through EventDiscard or implicitly discarded if it is the oldest bound signal when the bound signal limit of the event list entry to which it is bound is exceeded. In either case, the processing of the discarded signal by that process is considered complete. If this is a session level event, all processes must complete processing before signal processing is considered complete.

vm_evn_sync_process_signals
All threads currently existing in the signaling process, with the exception of those threads running as the result of a monitor activated by this signal, are suspended to await the outcome of event processing. Threads running as the result of monitor activation may create additional threads that are not initially suspended. Upon monitor reset, however, the additional threads are treated as any other thread.

event_flag_size
(input,INT,4) is a variable for specifying the number of elements in the *event_flag* array.

loose_signal_limit
(input,INT,4) is a variable for specifying the number of event signals that may be retained if no eligible event monitor exists to which to bind the signal at the time the event is signaled. When the limit is exceeded, the oldest loose signal is discarded to make room for the newest arrival. A value of 0 indicates that no loose signals are to be retained. A value of -1 means that the loose signal list is allowed to grow without limit, subject to the availability of virtual storage. Any other negative value is considered an error.

signal_timeout_period
(input,INT,4) is a variable for specifying the maximum length of time, in microseconds, that a signaling thread should remain suspended awaiting the completion of processing of the signal. A value of 0 indicates that the signaling thread should wait indefinitely for the completion of signal processing. If the event is created such that the signaling thread is to continue processing, then this parameter is ignored.

Usage Notes

1. There is no restriction on the character composition of an event name. The length of an event name may not exceed 16MB. By convention, system event names consist of uppercase and lowercase alphabetic, numeric, and break characters, and have a maximum length of 24 bytes.
2. If the event name has session-level scope, the event may be signaled or monitored in any process in the session, and each signal is delivered to each eligible monitor in any process in the session. Such an event name must be unique among all event names known anywhere in the session.
3. If the event name has process-level scope, it may be signaled and monitored only in the process in which it was created, and each signal is delivered only to each eligible monitor in that process. Such an event name need only be unique within the creating process, however, and there may be multiple instances of an event definition with process-level scope in a session.
4. Signal propagation rules are applied on a process basis. When qualifying event handlers exist in several processes, the signal is always delivered to at least one event monitor in each process.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_evn_success	vm_evn_success	EventCreate completed successfully
vm_evn_error	vm_evn_dup_name	*Event_name* is already defined
vm_evn_error	vm_evn_bad_name_len	*Event_name_length* is less than or equal to 0
vm_evn_error	vm_evn_name_too_long	*Event_name_length* is too large
vm_evn_error	vm_evn_bad_flag	*Event_flag* array contains an unrecognized value
vm_evn_error	vm_evn_bad_flag_size	*Event_flag_size* is less than 0
vm_evn_error	vm_evn_bad_limit	*Loose_signal_limit* is invalid
vm_evn_error	vm_evn_bad_time	*Signal_timeout_period* is less than 0
vm_evn_error	vm_evn_insufficient_storage	Out of storage

Programming Language Bindings

Language	Language Binding File
C	VMCEVN H
Assembler	VMASMEVN MACRO
REXX	VMREXEVN COPY

EventDelete — Delete an Event Definition

EventDelete

retcode
reascode
event_name
event_name_length

Purpose

Use the EventDelete function to delete a previously created event.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

event_name
(input,CHAR,*event_name_length*) is a variable for specifying the name of the event whose definition is being deleted.

event_name_length
(input,INT,4) is a variable for specifying the length of *event_name*.

Usage Notes

1. Upon event deletion, all signals of that event not in the current signal set of an active monitor are discarded.
2. Upon event deletion, all inactive monitors of the event with outstanding waits or traps are activated, regardless of their selection or enablement status.
3. Monitors active at the time of event deletion are unaffected. If such a monitor has a trap routine defined for it, the trap routine is run immediately upon the resetting of the monitor. Subsequent invocations of the EventWait and EventTest functions indicate that the event has been deleted.
4. If the condition defined by any monitors of the deleted event can no longer be satisfied, subsequent invocations of the EventWait, EventTest, or EventTrap function against such monitors indicate that the monitor can never be satisfied.
5. Only the process that created the event definition may delete it. An attempt to delete an event definition created by another process is considered an error.
6. During process termination, all event definitions created by the terminating process are deleted.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_evn_success	vm_evn_success	EventDelete completed successfully
vm_evn_error	vm_evn_no_name	*Event_name* is not defined
vm_evn_error	vm_evn_bad_name_len	*Event_name_length* is less than 0, equal to 0, or greater than 16M.
vm_evn_error	vm_evn_not_authorized	Requestor is ineligible to delete *event_name*

Programming Language Bindings

Language	Language Binding File
C	VMCEVN H
Assembler	VMASMEVN MACRO
REXX	VMREXEVN COPY

EventDiscard — Inhibit Further Propagation of Signals

EventDiscard
retcode
reascode
monitor_token
index

Purpose

Use the EventDiscard function to prevent signals in the current signal set of an event monitor from being propagated to successive event monitors. It is effective only for events defined to have sequential signal propagation.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

monitor_token
(input,INT,4) is a variable for specifying the token that identifies the monitor from whose current signal set a signal is to be discarded. A value of 0 may be used to identify the active event monitor most recently activated on the current thread.

index
(input,INT,4) is a variable for specifying, as an index into the event list specified in the creation of the event monitor, the event list entry corresponding to the signal to be discarded. A value of 0 indicates that all signals in the current signal set are to be discarded.

Usage Notes

1. Discarding a signal of an event defined for broadcast delivery has no effect other than removing that signal from the current signal set because the signal has already been delivered to all qualifying event monitors.
2. Signal propagation rules are applied on a process basis. When qualifying event handlers exist in several processes, the signal is always delivered to at least one event monitor in each process.
3. If EventDiscard is issued against an event monitor which is not active, or if *monitor_token* is specified as 0 and no monitor is active on the current thread, an error return code is generated and no other action is taken.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_evn_success	vm_evn_success	EventDiscard completed successfully
vm_evn_error	vm_evn_no_monitor	*Monitor_token* is unrecognizable
vm_evn_error	vm_evn_monitor_inactive	Specified monitor is not active
vm_evn_error	vm_evn_no_active_monitor	No monitor is active on the current thread
vm_evn_error	vm_evn_bad_index	*Index* is out of range

Programming Language Bindings

Language	Language Binding File
C	VMCEVN H
Assembler	VMASMEVN MACRO
REXX	VMREXEVN COPY

EventEnable — Enable or Disable for Specific Events

EventEnable

retcode
reascode
number_of_events
event_name_address
event_name_length
event_enablement_mask

Purpose

Use the EventEnable function to enable or disable monitor activation by specific event signals. After disabling for a particular event, signals of that event do not contribute to the activation of any monitor in the invoking process. Signals of that event, however, continue to be bound to monitors subject to their respective bound signal limits. On re-enabling for an event, bound signals of that event may again contribute to the activation of monitors in the invoking process.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

number_of_events
(input,INT,4) is a variable for specifying the number of events of interest.

event_name_address
(input,INT,*number_of_events*) is an array of *number_of_events* 4-byte variables, each element of which contains the address of the name of an event whose signals are to be enabled or disabled.

event_name_length
(input,INT,*number_of_events*) is an array of *number_of_events* 4-byte variables, each element of which contains the length of the event name pointed to by the corresponding element of the *event_name_address* array.

event_enablement_mask
(input,INT,*number_of_events*) is an array of *number_of_events* 4-byte variables, each element of which contains the enablement mask of the event pointed to by the corresponding element of the *event_name_address* array. The value of the enablement mask determines the action performed:

vm_evn_disable
Disable the event pointed to by the corresponding element of the *event_name_address* array.

vm_evn_enable
Enable the event pointed to by the corresponding element of the *event_name_address* array.

Usage Notes

1. The event enablement mask is maintained on a process basis; EventEnable affects the issuing process only.
2. If any of the event names cannot be found, all other specified events are still processed.

3. While the signals of a disabled event do not contribute to the activation of a monitor, information concerning those signals is returned upon an EventTest and, should the monitor be activated, those signals are processed in exactly the same way as other signals in the current signal set.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_evn_success	vm_evn_success	EventEnable completed successfully
vm_evn_warning	vm_evn_no_name	At least one event name is not defined
vm_evn_error	vm_evn_bad_num_of_events	*Number_of_events* is less than or equal to 0
vm_evn_error	vm_evn_bad_name_len	At least one *event_name_length* is less than or equal to 0
vm_evn_error	vm_evn_bad_mask	At least one *event_enablement_mask* is invalid

Programming Language Bindings

Language	Language Binding File
C	VMCEVN H
Assembler	VMASMEVN MACRO
REXX	VMREXEVN COPY

EventModify — Modify an Event Definition

EventModify

retcode
reascode
event_name
event_name_length
event_flag
event_flag_size
loose_signal_limit
signal_timeout_period

Purpose

Use the EventModify function to modify the characteristics of an event definition previously created in the same process.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

event_name
(input,CHAR,*event_name_length*) is a variable for specifying the name of the event whose definition is to be modified.

event_name_length
(input,INT,4) is a variable for specifying the length of *event_name*.

event_flag
(input,INT,*event_flag_size*) is an array of 4-byte variables, each element of which contains information about how the event is managed. Only one option from the following set may be specified. If no option from this set is specified, the existing value of the option remains unmodified.

- The treatment of the signaler

  vm_evn_async_signals
  The signaling thread is allowed to continue executing (the default).

  vm_evn_sync_thread_signals
  The signaling thread is suspended to await the outcome of event processing.

  vm_evn_sync_process_signals
  All threads in the signaling process, with the exception of those threads running as the result of a monitor activated by this signal, are suspended to await the outcome of event processing.

event_flag_size
(input,INT,4) is a variable for specifying the number of elements in the *event_flag* array.

loose_signal_limit
(input,INT,4) is a variable for specifying the number of event signals that may be retained if no eligible event monitor exists to which to bind the signal at the time the event is signaled. When the limit is exceeded, the oldest loose signal is discarded to make room for the newest arrival. A value of 0 indicates that no loose signals are retained. A value of -1 means that the loose signal list will be

allowed to grow without limit, subject to the availability of virtual storage. A value of -2 means that the existing *loose_signal_limit* is to remain unmodified. Any other negative value is considered an error.

signal_timeout_period
(input,INT,4) is a variable for specifying the maximum length of time, in microseconds, that a signaling thread should remain suspended awaiting the completion of processing of the signal. A value of 0 indicates that the signaling thread should wait indefinitely for the completion of signal processing. A value of -1 means that the existing *signal_timeout_period* is to remain unmodified. If the modified event definition does not include the suspension of the signaler, then this parameter is ignored.

Usage Notes

1. Only the process that created the event definition may modify it. An attempt to modify an event definition created by another process is considered an error.
2. Changes to the loose signal limit take effect immediately. Any other change does not take effect until a subsequent signal is issued.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_evn_success	vm_evn_success	EventModify completed successfully
vm_evn_error	vm_evn_bad_flag_size	*Flag_size* is less than 0
vm_evn_error	vm_evn_no_name	*Event_name* is not defined
vm_evn_error	vm_evn_bad_name_len	*Event_name_length* is less than or equal to 0
vm_evn_error	vm_evn_not_authorized	Requestor is ineligible to modify *event_name*
vm_evn_error	vm_evn_bad_flag	*Event_flag* array contains an unrecognized value
vm_evn_error	vm_evn_bad_limit	*Loose_signal_limit* is invalid
vm_evn_error	vm_evn_bad_time	*Signal_timeout_period* is invalid
vm_evn_error	vm_evn_insufficient_storage	Out of storage

Programming Language Bindings

Language	Language Binding File
C	VMCEVN H
Assembler	VMASMEVN MACRO
REXX	VMREXEVN COPY

EventMonitorCreate — Define an Event Handling Environment

EventMonitorCreate
retcode
reascode
monitor_token
monitor_flag
monitor_flag_size
number_of_events
event_name_address
event_name_length
event_key_address
event_key_length
bound_signal_limit
event_count

Purpose

Use the EventMonitorCreate function to specify combinations of event names and keys identifying conditions whose occurrence you want to monitor.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

monitor_token
(output,INT,4) is a variable where the function returns a token to identify the event monitor on subsequent invocations of other event management functions.

monitor_flag
(input,INT,*monitor_flag_size*) is an array of 4-byte variables, each element of which contains information about how the event monitor is managed. Only one option from each of the following sets may be specified. If no option from a particular set is given, the default is applied.

- The longevity of the monitor

  vm_evn_no_auto_delete
  Persists until explicit EventMonitorDelete (the default)

  vm_evn_auto_delete
  Automatically deleted at first deactivation or EventMonitorReset

- The effect of monitor activation on dispatchability

  vm_evn_async_monitor
  All threads in the process containing the monitor remain dispatchable (the default).

  vm_evn_sync_process_monitor
  All threads currently existing in the process containing the monitor, except the one on which the monitor is being activated, are suspended until the monitor is deactivated.

- The binding of loose signals to this monitor

vm_evn_bind_loose_signals **(the default)**
When the monitor is created, or monitoring is restarted by the EventSelect or EventMonitorSelect function, any loose signals for which this monitor is qualified are bound to this monitor.

vm_evn_ignore_loose_signals
No loose signals are bound to this monitor.

monitor_flag_size
(input,INT,4) is a variable for specifying the number of elements in the *monitor_flag* array.

number_of_events
(input,INT,4) is a variable for specifying the number of event list entries of interest.

event_name_address
(input,INT,*number_of_events*) is an array of *number_of_events* 4-byte variables, each element of which contains the address of the name of an event whose occurrence is monitored.

event_name_length
(input,INT,*number_of_events*) is an array of *number_of_events* 4-byte variables, each element of which contains the length of the event name pointed to by the corresponding element of the *event_name_address* array.

event_key_address
(input,INT,*number_of_events*) is an array of *number_of_events* 4-byte variables, each element of which contains the address of a key that further characterizes the particular instance of the event pointed to by the corresponding entry of the *event_name_address* array. The key may be chosen to match exactly the key that is carried by the signals of interest, or a partial key, possibly including wildcard characters, may be used to match a broader range of occurrences. The wildcard characters supported, and the matching rules applied when they are used, are the same as for the IPC match keys.

event_key_length
(input,INT,*number_of_events*) is an array of *number_of_events* 4-byte variables, each element of which contains the length of the event key pointed to by the corresponding element of the *event_key_address* array. The key may be null (that is, its length may be 0) if no secondary characterization of the event is required to define the occurrence of interest; a null key in a monitor matches any key on a signal.

bound_signal_limit
(input,INT,*number_of_events*) is an array of *number_of_events* 4-byte variables, each element of which contains the number of signals of the corresponding event list entry that may be retained bound to the event monitor but unprocessed during an interval when the monitor is already active or testable or while the monitored condition remains unsatisfied. When the limit is exceeded, the oldest bound signal of a particular event list entry is discarded to make room for the newest arrival. The minimum permissible value is 1, indicating that only the most recent instance of each signal is to be retained. A value of -1 means that the bound signal list is allowed to grow without limit, subject to the availability of virtual storage. Any other negative value, or 0, is considered an error.

event_count
(input,INT,4) is a variable for specifying the number of the specified event list entries for which signals must be bound to the monitor for the monitored condition to be considered satisfied. The value must fall between 1 and *number_of_events*.

Usage Notes

1. Use the EventTrap, EventTest, or EventWait function to establish the action to be taken when the condition being monitored is satisfied.
2. The scope of an event monitor is implicitly the process in which it was created. An event monitor token is not recognized outside the process in which the corresponding monitor was created.
3. The status of an event monitor can be affected by the EventDelete and EventMonitorDelete functions. For details of these effects, see the descriptions of EventDelete and EventMonitorDelete.

4. See "Tips on Constructing Keys" on page 30 for information about including binary data in event keys.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_evn_success	vm_evn_success	EventMonitorCreate completed successfully
vm_evn_error	vm_evn_bad_flag_size	*Monitor_flag_size* is less than 0
vm_evn_error	vm_evn_bad_flag	*Monitor_flag* array contains an unrecognized value
vm_evn_error	vm_evn_bad_num_of_events	*Number_of_events* is less than or equal to 0
vm_evn_error	vm_evn_no_name	An event name is not defined
vm_evn_error	vm_evn_bad_name_len	*Event_name_length* is less than or equal to 0
vm_evn_error	vm_evn_bad_key_len	*Event_key_length* is less than 0
vm_evn_error	vm_evn_bad_limit	*Bound_signal_limit* is invalid
vm_evn_error	vm_evn_bad_event_count	*Event_count* is out of range
vm_evn_error	vm_evn_insufficient_storage	Out of storage

Programming Language Bindings

Language	Language Binding File
C	VMCEVN H
Assembler	VMASMEVN MACRO
REXX	VMREXEVN COPY

EventMonitorDelete — Delete an Event Handling Environment

EventMonitorDelete
retcode
reascode
monitor_token

Purpose

Use the EventMonitorDelete function to delete a previously created event monitor.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

monitor_token
(input,INT,4) is a variable for specifying the token that identifies the event monitor to delete. A value of 0 may be used to identify the active event monitor most recently activated on the current thread.

Usage Notes

1. When an event monitor is deleted, all bound signals of broadcast events are discarded, and any other bound signal is either propagated to the next qualifying event monitor or discarded if no other qualifying event monitor exists.
2. If EventMonitorDelete is issued against an active event monitor, the deletion does not occur until processing of the current signal set has been completed and the event monitor has been returned to the inactive state or explicitly reset.
3. If an event monitor is deleted while there is an outstanding EventWait associated with it, the EventWait function is terminated with a reason code indicating that the event monitor has been deleted.
4. If an event monitor is deleted while there is an outstanding event trap associated with it, the trap routine is driven on the thread that invokes the EventMonitorDelete function and must determine from the information returned by EventTest that the event monitor has been deleted. When the EventMonitorDelete function is issued against an active event monitor, the trap routine is invoked on the thread that resets the monitor.
5. When a process is terminated, all its event monitors are deleted and all unprocessed signals are discarded.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_evn_success	vm_evn_success	EventMonitorDelete completed successfully
vm_evn_warning	vm_evn_monitor_still_active	The specified monitor is currently active. The monitor is not deleted until it is reset.
vm_evn_error	vm_evn_no_monitor	*Monitor_token* is unrecognizable
vm_evn_error	vm_evn_no_active_monitor	No monitor is active on the current thread

Programming Language Bindings

Language	Language Binding File
C	VMCEVN H
Assembler	VMASMEVN MACRO
REXX	VMREXEVN COPY

EventMonitorEnable — Enable or Disable Specific Monitors

EventMonitorEnable
retcode
reascode
number_of_monitors
monitor_tokens
monitor_enablement_masks

Purpose

Use the EventMonitorEnable function to enable or disable specific monitors. After a monitor is disabled, no signal results in the activation of the monitor. On reenabling, the monitor may again become active.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

number_of_monitors
(input,INT,4) is a variable for specifying the number of monitors of interest.

monitor_tokens
(input,INT,*number_of_monitors*) is an array of *number_of_monitors* 4-byte variables, each element of which contains the token of a monitor that is to be enabled or disabled. A value of 0 may be used to identify the active event monitor most recently activated on the current thread.

monitor_enablement_masks
(input,INT,*number_of_monitors*) is an array of *number_of_monitors* 4-byte variables, each element of which contains the enablement mask for the monitor identified by the corresponding element of the *monitor_tokens* array. The value of the enablement mask determines the action performed:

vm_evn_disable
Disable the monitor identified by the corresponding element of the *monitor_tokens* array.

vm_evn_enable
Enable the monitor identified by the corresponding element of the *monitor_tokens* array.

Usage Notes

1. If any of the tokens are invalid, all other tokens are still processed.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_evn_success	vm_evn_success	EventMonitorEnable completed successfully
vm_evn_warning	vm_evn_no_monitor	At least one monitor is not defined
vm_evn_warning	vm_evn_no_active_monitor	No monitor is active on the current thread
vm_evn_error	vm_evn_bad_mask	At least one *monitor_enablement_mask* is invalid
vm_evn_error	vm_evn_bad_token_size	Number of tokens is less than or equal to 0

Programming Language Bindings

Language	Language Binding File
C	VMCEVN H
Assembler	VMASMEVN MACRO
REXX	VMREXEVN COPY

EventMonitorQuery — Obtain Information About an Event Monitor

EventMonitorQuery

retcode
reascode
monitor_token
monitor_flag
monitor_flag_size
monitor_flag_count
number_of_events
event_name_buffer_address
event_name_buffer_length
event_name_length
event_key_buffer_address
event_key_buffer_length
event_key_length
bound_signal_limit
bound_signal_count
monitor_selection_mask
monitor_enablement_mask
event_count
present_event_count
trap_routine_address
trap_routine_name

Purpose

Use the EventMonitorQuery function to obtain information about the definition and status of a previously created event monitor.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

monitor_token
(input,INT,4) is a variable for specifying the event monitor about which information is returned. A value of 0 may be used to identify the active event monitor most recently activated on the current thread.

monitor_flag
(output,INT,*monitor_flag_size*) is an array of 4-byte variables, in each element of which the function returns information about how the event monitor is managed. Exactly one option from each of the following sets is included. The order in which these option values are filled in is not defined and may not necessarily reflect the order in which they are listed.

- Longevity of the monitor

vm_evn_no_auto_delete
Persists until explicit EventMonitorDelete or termination of the defining process.

vm_evn_auto_delete
Automatically deleted at first deactivation or EventMonitorReset call.

- Effect of monitor activation on dispatchability

vm_evn_async_monitor
All threads in the process containing the monitor remain dispatchable (the default).

vm_evn_sync_process_monitor
All threads in the process containing the monitor, except the one on which the monitor is being activated, are suspended until the monitor is deactivated.

- Binding of loose signals to this monitor

vm_evn_bind_loose_signals
Any qualifying loose signals that exist at the time the monitor is created or selected on are bound to the monitor (the default).

vm_evn_ignore_loose_signals
No loose signals are bound to the monitor.

- Current activation state of the monitor

vm_evn_monitor_active
Active — monitored condition is satisfied and a signal processing program is executing.

vm_evn_monitor_waiting
Waiting — the monitor is not active, the monitored condition is not satisfied, and there is an outstanding EventWait function associated with the monitor.

vm_evn_monitor_trapping
Trapping — the monitor is neither active nor waiting, the monitored condition is not satisfied, and there is a trap routine associated with the monitor.

vm_evn_monitor_testable
Testable — the monitor is not active, waiting or trapping; the monitored condition may or may not be satisfied, and there is neither an outstanding EventWait function or trap routine associated with the monitor.

monitor_flag_size
(input,INT,4) is a variable for specifying the number of elements in the *monitor_flag* array.

monitor_flag_count
(output,INT,4) is a variable where the function returns the number of elements it has set in the *monitor_flag* array.

number_of_events
(input,INT,4) is a variable for specifying the number of event list entries of interest.

event_name_buffer_address
(input,INT,*number_of_event*) is an array of *number_of_events* 4-byte variables, each element of which contains the address of a character variable in which the name of an event whose occurrence is monitored is returned.

event_name_buffer_length
(input,INT,*number_of_event*) is an array of *number_of_events* 4-byte variables, each element of which contains the length of the event name buffer pointed to by the corresponding element of the *event_name_buffer_address* array.

event_name_length
(output,INT,*number_of_event*) is an array of *number_of_events* 4-byte variables, in each element of which the function returns the actual length of the event name returned in the buffer pointed to by the corresponding element of the *event_name_buffer_address* array. If the name is longer than the buffer, it is truncated; if shorter, the excess space at the end of the buffer is unchanged.

event_key_buffer_address
(input,INT,*number_of_event*) is an array of *number_of_events* 4-byte variables, each element of which contains the address of a buffer in which the function returns the key that further characterizes the particular instance of the event name pointed to by the corresponding entry of the *event_name_buffer_address* array.

event_key_buffer_length
(input,INT,*number_of_event*) is an array of *number_of_events* 4-byte variables, each element of which contains the length of the buffer pointed to by the corresponding element of the *event_key_buffer_address* array.

event_key_length
(output,INT,*number_of_event*) is an array of *number_of_events* 4-byte variables, in each element of which the function returns the actual length of the event key returned in the buffer pointed to by the corresponding element of the *event_key_buffer_address* array. If the key is longer than the buffer, it is truncated; if shorter, the excess space at the end of the buffer is unchanged.

bound_signal_limit
(output,INT,*number_of_event*) is an array of *number_of_events* 4-byte variables, in each element of which the function returns the number of signals of the corresponding event list entry that may be retained bound to the event monitor but unprocessed during an interval when the monitor is already active or testable or while the monitored condition remains unsatisfied. When the limit is exceeded, the oldest bound signal of a particular event list entry is discarded to make room for the newest arrival. The minimum permissible value is 1, indicating that only the most recent instance of each signal is to be retained. A value of -1 means that the bound signal list is allowed to grow without limit, subject to the availability of virtual storage.

bound_signal_count
(output,INT,*number_of_event*) is an array of *number_of_events* 4-byte variables, in each element of which the function returns the number of signals of the corresponding event list entry that are currently bound to the event monitor but unprocessed.

monitor_selection_mask
(output,INT,4) is a variable where the function returns the selection mask for this monitor.

monitor_enablement_mask
(output,INT,4) is a variable where the function returns the enablement mask for this monitor.

event_count
(output,INT,4) is a variable where the function returns the number of the specified event list entries for which signals must be bound to the monitor for the monitored condition to be considered satisfied.

present_event_count
(output,INT,4) is a variable where the function returns the number of the specified event list entries for which signals are currently bound to the event monitor.

trap_routine_address
(output,INT,4) is a variable where the function returns the address of a routine to be invoked when the condition defined by the monitor is satisfied, as established by the EventTrap function. If no address has been established by EventTrap, a value of 0 is returned.

trap_routine_name
(output,CHAR,8) is a variable where the function returns the name of a routine to be invoked when the condition defined by the monitor is satisfied, as established by the EventTrap function, or blank if no event trap is associated with the monitor.

Usage Notes

1. If *monitor_token* is specified as 0 and no monitor is active on the current thread, an error return code is generated and no other action is taken.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_evn_success	vm_evn_success	EventMonitorQuery completed successfully
vm_evn_warning	vm_evn_flag_truncated	*Monitor_flag_size* is less than the total number of *monitor_flag* value sets; only a subset of the *monitor_flag* values was obtained
vm_evn_warning	vm_evn_event_truncated	*Number_of_events* is less than the total number of event list entries specified when the monitor was created; only a subset of the event list entries was obtained
vm_evn_warning	vm_evn_name_truncated	At least one event name was truncated because the buffer provided was too short
vm_evn_warning	vm_evn_key_truncated	At least one event key was truncated because the buffer provided was too short
vm_evn_error	vm_evn_no_monitor	*Monitor_token* is unrecognizable
vm_evn_error	vm_evn_no_active_monitor	No monitor is active on the current thread
vm_evn_error	vm_evn_bad_flag_size	*Monitor_flag_size* is less than or equal to 0
vm_evn_error	vm_evn_bad_num_of_events	*Number_of_events* is less than or equal to 0

Programming Language Bindings

Language	Language Binding File
C	VMCEVN H
Assembler	VMASMEVN MACRO
REXX	VMREXEVN COPY

EventMonitorReset — Reset the State of an Event Monitor

EventMonitorReset
retcode
reasscode
monitor_token

Purpose

Use the EventMonitorReset function to indicate that processing of the current signal set of an event monitor is complete.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

monitor_token
(input,INT,4) is a variable for specifying the token that identifies the monitor whose state is reset. A value of 0 may be used to identify the active event monitor most recently activated on the current thread.

Usage Notes

1. Upon the invocation of the EventMonitorReset function, the current signal set is cleared. If it included any signals of events defined for sequential propagation that have not already been explicitly discarded, each of them is delivered to the next qualifying event monitor. The remaining bound signals are then examined. If they are sufficient to satisfy the monitored condition, the monitor is eligible for immediate reactivation; otherwise, the monitor becomes inactive.
2. EventMonitorReset is automatically performed whenever an active event monitor (that is, one for which a trap routine is executing, an EventWait function has been satisfied, or a successful EventTest function has been performed) makes a transition into an inactive state (that is, by returning control from the trap routine or flowing to another EventWait function) without having been explicitly reset. Thus, EventMonitorReset must be explicitly invoked only under relatively unusual circumstances, such as to indicate the completion of event processing *prematurely* (for example, to allow a trap routine to be reentered), or to allow a monitor to be polled successively with the EventTest function.
3. If EventMonitorReset is issued against an event monitor that is not active, or if *monitor_token* is specified as 0 and no monitor is active on the current thread, an error return code is generated and no other action is taken.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_evn_success	vm_evn_success	EventMonitorReset completed successfully
vm_evn_error	vm_evn_no_monitor	*Monitor_token* is unrecognizable
vm_evn_error	vm_evn_monitor_inactive	Specified monitor is not active
vm_evn_error	vm_evn_no_active_monitor	No monitor is active on the current thread

Programming Language Bindings

Language	Language Binding File
C	VMCEVN H
Assembler	VMASMEVN MACRO
REXX	VMREXEVN COPY

EventMonitorSelect — Start or Stop Monitoring by Specific Monitors

EventMonitorSelect
retcode
reascode
number_of_monitors
monitor_tokens
monitor_selection_masks

Purpose

Use the EventMonitorSelect function to start or stop monitoring by specific monitors. After monitoring is stopped, no signals are bound to the specified monitor. On restarting monitoring, loose signals are bound to the specified monitor in accordance with the monitor definition.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

number_of_monitors
(input,INT,4) is a variable for specifying the number of monitors of interest.

monitor_tokens
(input,INT,*number_of_monitors*) is an array of *number_of_monitors* 4-byte variables, each element of which contains the token of the monitor that is selected on or off. A value of 0 may be used to identify the active event monitor most recently activated on the current thread.

monitor_selection_masks
(input,INT,*number_of_monitors*) is an array of *number_of_monitors* 4-byte variables, each element of which contains the selection mask for the monitor identified by the corresponding element of the *monitor_tokens* array. The value of the selection mask determines the action to be performed:

vm_evn_select_off
Stop monitoring of the monitor identified by the corresponding element of the *monitor_tokens* array.

vm_evn_select_on
Start monitoring of the monitor identified by the corresponding element of the *monitor_tokens* array.

Usage Notes

1. If any of the tokens are invalid, all other tokens are still processed.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_evn_success	vm_evn_success	EventMonitorSelect completed successfully
vm_evn_warning	vm_evn_no_monitor	At least one monitor is not defined
vm_evn_warning	vm_evn_no_active_monitor	No monitor is active on the current thread

Return Code	Reason Code	Meaning
vm_evn_error	vm_evn_bad_token_size	Number of tokens is less than or equal to 0
vm_evn_error	vm_evn_bad_mask	At least one *monitor_selection_mask* is invalid

Programming Language Bindings

Language	Language Binding File
C	VMCEVN H
Assembler	VMASMEVN MACRO
REXX	VMREXEVN COPY

EventQuery — Obtain Information about an Event Definition

EventQuery

retcode
reascode
event_name
event_name_length
event_flag
event_flag_size
event_flag_count
loose_signal_limit
signal_timeout_period
loose_signal_count
event_selection_mask
event_enablement_mask
monitor_token
monitor_token_size
monitor_token_count

Purpose

Use the EventQuery function to obtain information about an existing event definition, including a list of all event monitors defined in the current process which are sensitive to occurrences of the event. This function is primarily for use in obtaining diagnostic information. Use the EventMonitorQuery function to obtain further information about a particular event monitor.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

event_name
(input,CHAR,*event_name_length*) is a variable for specifying the name of an event to be queried.

event_name_length
(input,INT,4) is a variable for specifying the length of *event_name*.

event_flag
(output,INT,*event_flag_size*) is an array of 4-byte variables, in each element of which the function returns information about how the event is managed. Exactly one option from each of the following sets is included. The order in which these options values are filled in is not defined and may not necessarily reflect the order in which they are listed here.

- Scope of the event name

  vm_evn_process_scope
  Process

  vm_evn_session_scope
  Session

- Manner in which an event signal is propagated to multiple event monitors

vm_evn_broadcast_signals
Broadcast

vm_evn_fifo_signals
FIFO sequence

vm_evn_lifo_signals
LIFO sequence

- Treatment of the signaler

vm_evn_async_signals
The signaling thread is allowed to continue executing (the default).

vm_evn_sync_thread_signals
The signaling thread is suspended to await the outcome of event processing.

vm_evn_sync_process_signals
All threads in the signaling process, with the exception of those threads running as the result of a monitor activated by this signal, are suspended to await the outcome of event processing.

event_flag_size
(input,INT,4) is a variable for specifying the number of elements in the *event_flag* array.

event_flag_count
(output,INT,4) is a variable where the function returns the number of elements it has set in the *event_flag* array.

loose_signal_limit
(output,INT,4) is a variable where the function returns the number of event signals that may be retained if no eligible event monitor exists to which to bind the signal or if a process is not monitoring for the event at the time the event is signaled. When the limit is exceeded, the oldest loose signal is discarded to make room for the newest arrival. A value of 0 indicates that no loose signals are retained. A value of -1 indicates that the loose signal list is allowed to grow without limit, subject to the availability of virtual storage.

signal_timeout_period
(output,INT,4) is a variable where the function returns the maximum length of time, in microseconds, that a signaling thread is allowed to remain suspended awaiting the completion of processing of the signal. A value of 0 indicates that the signaling thread waits indefinitely for the completion of signal processing. If the option specifying that the signalling thread is to continue processing is included in the *event_flag* array, then this parameter is meaningless.

loose_signal_count
(output,INT,4) is a variable where the function returns the number of loose signals currently being retained for the specified event.

event_selection_mask
(output,INT,4) is a variable where the function returns the state of the event selection mask, as follows:

vm_evn_select_off
Not monitoring for *event_name*

vm_evn_select_on
Monitoring for *event_name*

event_enablement_mask
(output,INT,4) is a variable where the function returns the state of the event enablement mask, as follows:

vm_evn_disable
Disabled for *event_name*

vm_evn_enable
Enabled for *event_name*

monitor_token
(output,INT,4) is an array of 4-byte variables, in which the function returns the list of tokens identifying the event monitors defined in the current process that are sensitive to the specified event. For events defined as sequential, the tokens are returned in the order in which they are processed. For events defined as broadcast, the tokens are returned in the order in which the monitors were created.

monitor_token_size
(input,INT,4) is a variable for specifying the number of elements in the *monitor_token* array that are available to be filled in.

monitor_token_count
(output,INT,4) is a variable where the function returns the total number of event monitors defined in the current process that are sensitive to the specified event. If *monitor_token_count* is not greater than *monitor_token_size*, then the first *monitor_token_count* elements of the *monitor_token* array contain the tokens that identify that entire set of monitors and the remainder, if any, are unchanged; otherwise, only the first *monitor_token_size* monitor tokens are returned.

Usage Notes

1. The event selection and enablement masks are maintained on a process basis and are reported for the issuing process only.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_evn_success	vm_evn_success	EventQuery completed successfully
vm_evn_warning	vm_evn_flag_truncated	*Event_flag_size* is less than the total number of *event_flag* value sets; only a subset of the *event_flag* values was obtained
vm_evn_warning	vm_evn_token_truncated	*Monitor_token_size* is less than the total number of monitors defined; only a subset of the monitor tokens has been returned
vm_evn_error	vm_evn_no_name	*Event_name* is not defined
vm_evn_error	vm_evn_bad_name_len	*Event_name_length* is less than or equal to 0
vm_evn_error	vm_evn_bad_flag_size	Flag size is less than 0
vm_evn_error	vm_evn_bad_token_size	*Monitor_token_size* is less than 0

Programming Language Bindings

Language	Language Binding File
C	VMCEVN H
Assembler	VMASMEVN MACRO
REXX	VMREXEVN COPY

EventQueryAll — Obtain All Event Names and Monitor Tokens

EventQueryAll

retcode
reascode
number_of_events
event_name_address
event_name_length
actual_name_length
event_name_count
monitor_token
monitor_token_size
monitor_token_count

Purpose

Use the EventQueryAll function to obtain the names of all events and the tokens for all event monitors visible to this process. This function is primarily for use in obtaining diagnostic information. Use the EventQuery and EventMonitorQuery functions to get further information about events and event monitors.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

number_of_events
(input,INT,4) is a variable for specifying the number of elements in the following three arrays.

event_name_address
(input,INT,*number_of_events*) is an array of *number_of_events* 4-byte variables, each element of which contains the address of a character variable in which the function returns the name of an event visible to this process.

event_name_length
(input,INT,*number_of_events*) is an array of *number_of_events* 4-byte variables, each element of which contains the length of the buffer of the corresponding element of the *event_name_address* array. If the name is longer than the buffer, it is truncated; if shorter, the excess space at the end of the buffer is unchanged.

actual_name_length
(output,INT,*number_of_events*) is an array of *number_of_events* 4-byte variables, in each element of which the function returns the actual length of the event name pointed to by the corresponding element of the *event_name_address* array.

event_name_count
(output,INT,4) is a variable where the function returns the total number of events visible to the current process. If *event_name_count* is not greater than *number_of_events*, then the first *event_name_count* elements of the arrays associated with the event names are output and the remaining array elements, if any, are unchanged. Otherwise, only the first *number_of_events* event names are returned.

monitor_token
(output,INT,*monitor_token_size*) is an array of 4-byte variables in which the function returns the list of tokens identifying all the event monitors defined in the current process.

monitor_token_size
(input,INT,4) is a variable for specifying the number of elements in the *monitor_token* array that are available to be filled in.

monitor_token_count
(output,INT,4) is a variable where the function returns the total number of event monitors defined in the current process. If *monitor_token_count* is not greater than *monitor_token_size*, then the first *monitor_token_count* elements of the *monitor_token* array contain the tokens that identify that entire set of monitors and the remainder, if any, are unchanged. Otherwise, only the first *monitor_token_size* monitor tokens are returned.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_evn_success	vm_evn_success	EventQueryAll completed successfully
vm_evn_warning	vm_evn_name_truncated	At least one event name was truncated because the buffer provided was too short
vm_evn_warning	vm_evn_event_truncated	*Number_of_events* is less than the total number of events visible to this process; only a subset of the event names has been returned
vm_evn_warning	vm_evn_token_truncated	*Monitor_token_size* is less than the total number of monitors defined; only a subset of the monitor tokens has been returned
vm_evn_error	vm_evn_bad_num_of_events	Number of events is less than 0
vm_evn_error	vm_evn_bad_token_size	*Monitor_token_size* is less than 0

Programming Language Bindings

Language	Language Binding File
C	VMCEVN H
Assembler	VMASMEVN MACRO
REXX	VMREXEVN COPY

EventRetrieve — Retrieve Data From an Event

EventRetrieve

retcode
reascode
monitor_token
index
data_buffer
data_buffer_length
event_data_length

Purpose

Use the EventRetrieve function to retrieve data from an event signal in the current signal set of an active event monitor.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

monitor_token
(input,INT,4) is a variable for specifying the token that identifies the monitor from whose current signal set the retrieval is performed. A value of 0 may be used to identify the active event monitor most recently activated on the current thread.

index
(input,INT,4) is a variable for specifying, as an index into the event list specified in the creation of the event monitor, the event list entry corresponding to the signal from which the data is retrieved.

data_buffer
(output,CHAR,*data_buffer_length*) is a variable where the function returns the signaled data for the event list entry identified by *index*.

data_buffer_length
(input,INT,4) is a variable for specifying the length of *data_buffer*.

event_data_length
(output,INT,4) is a variable where the function returns the length of the signaled data for the event list entry identified by *index*. If *event_data_length* is greater than *data_buffer_length*, the signaled data is truncated on the right and a warning return code is generated.

Usage Notes

1. If EventRetrieve is issued repeatedly for the same *index* during a single activation of an event monitor, it will retrieve the same event-related data on each call.
2. If EventRetrieve is issued against an event monitor that is not active, or if *monitor_token* is specified as 0 and no monitor is active on the current thread, an error return code is generated and no other action is taken.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_evn_success	vm_evn_success	EventRetrieve completed successfully
vm_evn_warning	vm_evn_data_truncated	*Event_data_length* exceeds *data_buffer_length*; the event data copied to *data_buffer* has been truncated
vm_evn_error	vm_evn_no_monitor	*Monitor_token* is unrecognizable
vm_evn_error	vm_evn_monitor_inactive	Specified monitor is not active
vm_evn_error	vm_evn_no_active_monitor	No monitor is active on the current thread
vm_evn_error	vm_evn_bad_index	*Index* is out of range
vm_evn_error	vm_evn_bad_data_len	*Data_buffer_length* is less than 0

Programming Language Bindings

Language	Language Binding File
C	VMCEVN H
Assembler	VMASMEVN MACRO
REXX	VMREXEVN COPY

EventSelect — Start or Stop Monitoring for Specific Events

```
EventSelect
        retcode
        reascode
        number_of_events
        event_name_address
        event_name_length
        event_selection_mask
```

Purpose

Use the EventSelect function to start or stop monitoring of specific event signals. This means that no signals of this event will be bound to any monitor in the invoking process. After monitoring of a particular event is stopped, signals of that event are retained in accordance with the *loose_signal_limit* specified when the event definition was created. On restarting monitoring for an event, any retained signals are delivered to qualifying event monitors in accordance with the event and monitor definitions.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

number_of_events
(input,INT,4) is a variable for specifying the number of events of interest.

event_name_address
(input,INT,*number_of_events*) is an array of *number_of_events* 4-byte variables, each element of which contains the address of the name of an event whose signals are to be selected on or off.

event_name_length
(input,INT,*number_of_events*) is an array of *number_of_events* 4-byte variables, each element of which contains the length of the event name pointed to by the corresponding element of the *event_name_address* array.

event_selection_mask
(input,INT,*number_of_events*) is an array of *number_of_events* 4-byte variables, each element of which contains the selection mask of the event name pointed to by the corresponding element of the *event_name_address* array. The value of the selection mask determines the action to be performed:

vm_evn_select_off
Stop monitoring of the event whose name is pointed to by the corresponding element of the *event_name_address* array.

vm_evn_select_on
Start monitoring of the event whose name is pointed to by the corresponding element of the *event_name_address* array.

Usage Notes

1. The event selection mask is maintained on a process basis; EventSelect affects the issuing process only.
2. If any of the event names cannot be found, all other specified events are still processed.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_evn_success	vm_evn_success	EventSelect completed successfully
vm_evn_warning	vm_evn_no_name	At least one event name is not defined
vm_evn_error	vm_evn_bad_name_len	At least one element of the *event_name_length* array is less than or equal to 0
vm_evn_error	vm_evn_bad_num_of_events	*number_of_events* is less than or equal to 0
vm_evn_error	vm_evn_bad_mask	At least one element of the *event_selection_mask* array is invalid
vm_evn_error	vm_evn_insufficient_storage	Out of storage

Programming Language Bindings

Language	Language Binding File
C	VMCEVN H
Assembler	VMASMEVN MACRO
REXX	VMREXEVN COPY

EventSignal — Signal the Occurrence of an Event

EventSignal
retcode
reascode
event_name
event_name_length
event_data
event_data_length
event_key_offset
event_key_length

Purpose

Use the EventSignal function to indicate the occurrence of the specified event, and, optionally, to pass data associated with the occurrence to any event monitors that have registered an interest in the event.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

event_name
(input,CHAR,*event_name_length*) is a variable for specifying the name of the event whose occurrence is to be signaled.

event_name_length
(input,INT,4) is a variable for specifying the length of *event_name*.

event_data
(input,CHAR,*event_data_length*) is a variable for specifying data to be associated with this signal.

event_data_length
(input,INT,4) is a variable for specifying the length of *event_data*.

event_key_offset
(input,INT,4) is a variable for specifying the offset in *event_data* of the first byte of a key that characterizes the particular instance of the event to be signaled.

event_key_length
(input,INT,4) is a variable for specifying the length of the key from *event_data*. The key may be null (that is, its length may be 0) if no secondary characterization of the event is necessary for this type of event or for this occurrence of the event.

Usage Notes

1. See "Tips on Constructing Keys" on page 30 for information about including binary data in event keys.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_evn_success	vm_evn_success	EventSignal completed successfully

Return Code	Reason Code	Meaning
vm_evn_warning	vm_evn_timeout	*Signal_timeout_period* expired before signal processing was completed
vm_evn_error	vm_evn_no_name	*Event_name* is not defined
vm_evn_error	vm_evn_bad_name_len	*Event_name_length* is less than or equal to 0
vm_evn_error	vm_evn_bad_key	*Event_key* is invalid
vm_evn_error	vm_evn_bad_key_offset	*Event_key_offset* is out of range
vm_evn_error	vm_evn_bad_key_len	*Event_key_length* is out of range
vm_evn_error	vm_evn_bad_data_len	*Event_data_length* is less than zero
vm_evn_error	vm_evn_insufficient_storage	Out of storage

Programming Language Bindings

Language	Language Binding File
C	VMCEVN H
Assembler	VMASMEVN MACRO
REXX	VMREXEVN COPY

EventTest — Test for the Occurrence of Events

EventTest

retcode
reascode
monitor_token
number_of_events
event_flag

Purpose

Use the EventTest function to check the condition of an existing event monitor.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

monitor_token
(input,INT,4) is a variable for specifying the token that identifies the monitor whose condition is to be tested. A value of 0 may be used to identify the active event monitor most recently activated on the current thread.

number_of_events
(input,INT,4) is a variable for specifying the number of events whose occurrence is to be tested. In general, this should be the same as the number of event_name and event_key combinations specified in the definition of the event monitor.

event_flag
(output,INT,*number_of_events*) is an array of *number_of_events* 4-byte variables, in each element of which the function returns an indication of the occurrence or nonoccurrence of the event identified by the corresponding event list entries specified in defining the event monitor:

0,1,2,...
The event has been signaled; the number is the length of data provided on the signal that may be obtained with the EventRetrieve function.

-1
The event has not been signaled.

-2
The event definition has been deleted.

-3
No corresponding event list entry was defined in the event monitor.

Usage Notes

1. A successful EventTest issued against an inactive event monitor causes the monitor to be activated and establishes its current signal set.
2. EventTest issued against an active event monitor simply reports on the current signal set.
3. If *monitor_token* is specified as 0 and no monitor is active on the current thread, an error return code is generated and no other action is taken.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_evn_success	vm_evn_success	EventTest completed successfully
vm_evn_warning	vm_evn_event_truncated	*Number_of_events* is less than the number of event list entries specified in the creation of the event monitor; only a subset of event occurrences has been indicated in event flags
vm_evn_warning	vm_evn_cannot_satisfy	Monitor cannot be satisfied
vm_evn_warning	vm_evn_event_deleted	One or more event names have been deleted
vm_evn_warning	vm_evn_signal_lost	One or more signals may have been lost due to lack of storage
vm_evn_warning	vm_evn_monitor_inactive	Condition is not satisfied
vm_evn_error	vm_evn_no_monitor	*Monitor_token* is unrecognizable
vm_evn_error	vm_evn_no_active_monitor	No monitor is active on the current thread
vm_evn_error	vm_evn_bad_num_of_events	*Number_of_events* is less than 0

Programming Language Bindings

Language	Language Binding File
C	VMCEVN H
Assembler	VMASMEVN MACRO
REXX	VMREXEVN COPY

EventTrap — Define an Asynchronous Event Handler

EventTrap

retcode

reascode

monitor_token

trap_routine_address

trap_routine_name

Purpose

Use the EventTrap function to nominate a routine to receive control asynchronously when the condition defined by an existing event monitor is satisfied.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

monitor_token
(input,INT,4) is a variable for specifying the token that identifies the monitor whose condition is to be trapped. A value of 0 may be used to identify the active event monitor most recently activated on the current thread.

trap_routine_address
(input,INT,4) is a variable for specifying the address of the routine to be invoked when the condition defined by the monitor is satisfied. If an address was previously associated with the event monitor, it is automatically replaced by the new *trap_routine_address*. If the address is a nonzero value, the *trap_routine_name* parameter is meaningless.

trap_routine_name
(input,CHAR,8) is a variable for specifying the name of the routine to be invoked when the condition defined by the monitor is satisfied. This parameter has meaning only if the *trap_routine_address* parameter is 0. If a name was previously associated with the event monitor, it is automatically replaced by the new *trap_routine_name*. If the *trap_routine_address* parameter is zero and the *trap_routine_name* is blank, the trap associated with the specified event monitor is canceled.

Usage Notes

1. A trap routine specified by the trap routine address parameter must be part of the same load module as the tasking application and may be a C or assembler program.
2. A trap routine is run on a thread in a class by itself. Trap routines invoked as a result of a session level signal run at the priority of the root process. Trap routines invoked as a result of a process level signal run at the priority of the signaler.
3. A trap routine specified by the trap routine name parameter must be a program that can be invoked with CMSCALL. Its entry conditions are those of standard CMSCALL linkage.
4. The trap routine must use the EventTest function to determine which combination of events caused the event monitor to be activated and the EventRetrieve function to obtain whatever data was provided when those events were signaled.
5. For events in which signals are propagated in serial order, the trap routine may inhibit the propagation of signal to handlers of lower precedence by calling the EventDiscard function.

6. If *monitor_token* is specified as 0 and no monitor is active on the current thread, an error return code is generated and no other action is taken.
7. REXX applications can specify the trap routine by name but not by address. They must always pass zero in the *trap_routine_address* parameter.
8. The entry point where execution of the new thread is to begin cannot be VMSTART, the multitasking initialization entry point.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_evn_success	vm_evn_success	EventTrap completed successfully
vm_evn_warning	vm_evn_cannot_satisfy	Monitor can never be satisfied
vm_evn_error	vm_evn_no_monitor	*Monitor_token* is unrecognizable
vm_evn_error	vm_evn_no_active_monitor	No monitor is active on the current thread
vm_evn_error	vm_evn_not_mt	The application is not part of a process initiated by the multitasking initialization routine VMSTART

Programming Language Bindings

Language	Language Binding File
C	VMCEVN H
Assembler	VMASMEVN MACRO
REXX	VMREXEVN COPY

EventWait — Wait for the Occurrence of Events

EventWait
retcode
reascode
monitor_token
number_of_events
event_flag

Purpose

Use the EventWait function to await the satisfaction of the condition defined by an existing event monitor.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

monitor_token
(input,INT,4) is a variable for specifying the token that identifies the monitor whose condition is to be awaited. The monitor must not already be in the waiting state, and there may be no more than one wait outstanding for a monitor at a time. A value of 0 may be used to identify the active event monitor most recently activated on the current thread.

number_of_events
(input,INT,4) is a variable for specifying the number of events whose occurrence is to be indicated when the EventWait function completes. In general, this should be the same as the number of event_name and event_key combinations specified in the definition of the event monitor.

event_flag
(output,INT,*number_of_events*) is an array of *number_of_events* 4-byte variables, in each element of which the function returns an indication of the occurrence or nonoccurrence of the event identified by the corresponding event list entries specified in defining the event monitor:

0,1,2,...
The event has been signaled; the number is the length of data provided on the signal that may be obtained with the EventRetrieve function.

-1
The event has not been signaled.

-2
The event definition has been deleted.

-3
No corresponding event list entry was defined in the event monitor.

Usage Notes

1. If EventWait is issued for an event monitor that already has a trap routine associated with it, the wait takes precedence over the trap; that is, the subsequent satisfaction of the condition defined for the monitor will result in the wait being satisfied but will not cause the trap routine to be driven. The trap will not become viable again until the monitor is reset.
2. If EventWait is issued against an active monitor, that monitor is reset.

3. If *monitor_token* is specified as 0 and no monitor is active on the current thread, an error return code is generated and no other action is taken.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_evn_success	vm_evn_success	EventWait completed successfully
vm_evn_warning	vm_evn_event_truncated	*Number_of_events* is less than the number of event list entries specified in the creation of the event monitor; only a subset of event occurrences has been indicated
vm_evn_warning	vm_evn_event_deleted	EventWait was terminated because one or more events were deleted
vm_evn_warning	vm_evn_monitor_deleted	EventWait was terminated because the event monitor was deleted
vm_evn_warning	vm_evn_signal_lost	One or more signals may have been lost due to lack of storage
vm_evn_warning	vm_evn_cannot_satisfy	EventWait was terminated because the event monitor cannot be satisfied
vm_evn_error	vm_evn_already_waiting	Event monitor is already waiting
vm_evn_error	vm_evn_no_monitor	*Monitor_token* is unrecognizable
vm_evn_error	vm_evn_no_active_monitor	No monitor is active on the current thread
vm_evn_error	vm_evn_bad_num_of_events	*Number_of_events* is less than 0

Programming Language Bindings

Language	Language Binding File
C	VMCEVN H
Assembler	VMASMEVN MACRO
REXX	VMREXEVN COPY

MonitorBufferGet — Obtain the Address of the CMS Monitor Data Area

MonitorBufferGet
retcode
reascode
monitor_buffer_address

Purpose

Use the MonitorBufferGet function to obtain the address of the CMS monitor data area.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

monitor_buffer_address
(output,INT,4) is a variable where the function stores the address of the monitor data area.

Usage Notes

1. The monitor data area contains information about threading operations and POSIX processes. In particular, the following information is contained in the monitor data area:

   ThreadCreate count
   The number of threads created

   ThreadCreate time
   The amount of time spent creating threads (TOD clock units)

   ThreadDelete count
   The number of threads deleted

   ThreadDelete time
   The amount of time spent deleting threads (TOD clock units)

   Slow switch count
   The number of times the "slow path" through the CMS dispatcher was taken

   Fast switch count
   The number of times the "fast path" through the CMS dispatcher was taken

   Blocked threads
   The current number of blocked threads

   Process watermark
   The greatest number of processes that have existed concurrently

   Thread watermark
   The greatest number of threads that have existed concurrently

   Process limit failures
   The number of times an attempt to create a POSIX process has failed because the process limit was reached

2. Mapping macros for the monitor data area are available in the language binding files for MonitorBufferGet. See those files for specific details.

3. There is one monitor data area for the duration of the CMS session. Once the application has obtained the address of this buffer, it can inspect the buffer at will.
4. Data collection begins when the first multitasking or ö application is executed.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_mon_success	vm_mon_success	MonitorBufferGet completed successfully

Programming Language Bindings

Language	Language Binding File
C	VMCMON H
Assembler	VMASMMON MACRO
REXX	VMREXMON COPY

MutexAcquire — Acquire a Mutex

MutexAcquire
retcode
reascode
mutex_handle
wait_on_mutex

Purpose

Use the MutexAcquire function to gain possession of a mutex. If another thread holds the mutex, the issuing thread may be blocked until the mutex is available.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

mutex_handle
(input,INT,4) is a variable for specifying the handle of the mutex to be acquired. This value is returned by the MutexCreate or MutexGetHandle function.

wait_on_mutex
(input,INT,4) is a variable for specifying whether the invoking thread wishes to wait for the mutex to become available if it is already held. The valid values are:

vm_syn_dont_wait_on_mutex
Do not wait if the mutex is held by another thread.

vm_syn_wait_on_mutex
Wait. When the wait is satisfied, the thread gains possession of the mutex.

Usage Notes

1. If a thread holding a mutex terminates, the mutex is released and the next thread to acquire the mutex receives a warning return code indicating that the shared resource protected by the mutex may be in an indeterminate state. This thread must determine if the resource is in a valid state and what to do if it is not. This situation may be caused by such items as a programming error (forgetting to release the mutex) or a program check occurring when a thread is in a critical section.

   If a thread holding a mutex abnormally terminates, an abnormal event termination event handler may try to clean up and release the mutex (only if it executes on the failing thread) and recover.
2. If a mutex is deleted, any thread waiting to acquire the mutex is unblocked and given an error return code.
3. If a thread attempts to acquire a mutex that it already holds, an error is returned.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_syn_success	vm_syn_success	MutexAcquire completed successfully

Return Code	Reason Code	Meaning
vm_syn_warning	vm_syn_indeterminate_state	Resource protected by mutex may be in an indeterminate state
vm_syn_error	vm_syn_handle_not_found	Mutex indicated by *mutex_handle* does not exist
vm_syn_error	vm_syn_mutex_already_held	Mutex is already held by another thread
vm_syn_error	vm_syn_mutex_held_by_caller	Mutex is already held by the caller
vm_syn_error	vm_syn_bad_wait_on_mutex	Invalid *wait_on_mutex* parameter
vm_syn_error	vm_syn_mutex_deleted	Mutex has been deleted

Programming Language Bindings

Language	Language Binding File
C	VMCSYN H
Assembler	VMASMSYN MACRO
REXX	VMREXSYN COPY

MutexCreate — Create a Mutex

MutexCreate
retcode
reascode
mutex_handle
mutex_name
mutex_name_length
scope_of_mutex

Purpose

Use the MutexCreate function to create a mutex.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

mutex_handle
(output,INT,4) is a variable where the function returns the handle of the mutex that is created.

mutex_name
(input,CHAR,*mutex_name_length*) is a variable for specifying the name of the mutex.

mutex_name_length
(input,INT,4) is a variable for specifying the length of *mutex_name*. It must be greater than 0 and less than 16MB in length.

scope_of_mutex
(input,INT,4) is a variable for specifying the scope of the mutex. The valid values are:

vm_syn_process_scope
The current process

vm_syn_session_scope
The current session

Usage Notes

1. A mutex can have either process scope or session scope, as follows:
   - A mutex that has process scope is known only in the process where it is created and can be manipulated by only the threads in that process. Such a mutex must have a name unique among all process-level mutexes created by the calling process.
   - A mutex that has session scope is known in all the processes in the session and can be manipulated by any of the threads in these processes. Such a mutex must have a name unique among all session-level mutexes.
2. The mutex scope is fixed for the life of the mutex.
3. All mutexes created by a process are deleted when a process terminates. Any threads in other processes waiting on such mutexes are unblocked and given a nonzero return code.

4. CMS supports up to 32,768 session-scope semaphores, mutexes, and condition variables, altogether. Also, for each process, CMS supports up to 32,768 process-scope semaphores, mutexes, and condition variables, altogether.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_syn_success	vm_syn_success	MutexCreate completed successfully
vm_syn_error	vm_syn_mutex_already_exists	Mutex already exists
vm_syn_error	vm_syn_bad_scope_of_mutex	Invalid *scope_of_mutex* parameter
vm_syn_error	vm_syn_insufficient_storage	Mutex not created because storage is not available
vm_syn_error	vm_syn_bad_mutex_name_len	*mutex_name_length* is out of range.
vm_syn_error	vm_syn_limit_reached	Selected scope's limit on total number of synchronization objects has been reached.

Programming Language Bindings

Language	Language Binding File
C	VMCSYN H
Assembler	VMASMSYN MACRO
REXX	VMREXSYN COPY

MutexDelete — Delete a Mutex

```
MutexDelete
        retcode
        reascode
        mutex_handle
```

Purpose

Use the MutexDelete function to delete a mutex.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

mutex_handle
(input,INT,4) is a variable for specifying the handle of the mutex to be deleted. This value is returned by the MutexCreate or MutexGetHandle function.

Usage Notes

1. A mutex can be deleted only by the process that created it. An error is returned if the calling process is not the creating process.
2. Two conditions must exist for a thread to delete a mutex. They are:
   - The thread must be in the process in which the mutex was created.
   - The thread must be holding the mutex.

   If either of these conditions are not met, an error is returned.
3. If a mutex is deleted, any thread waiting to acquire the mutex is unblocked and given an error return code.
4. If a mutex is deleted, all the condition variables associated with this mutex are deleted. Any thread waiting on such condition variables is unblocked and given a return code indicating that the condition variable and the mutex have been deleted.
5. All mutexes created by a process are deleted when a process terminates. Any threads in other processes waiting on such mutexes are unblocked and given an error return code.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_syn_success	vm_syn_success	MutexDelete completed successfully
vm_syn_error	vm_syn_handle_not_found	Mutex indicated by *mutex_handle* does not exist
vm_syn_error	vm_syn_not_mutex_creator	Mutex is not deleted because process is not the mutex creator
vm_syn_error	vm_syn_mutex_not_held	Mutex is not deleted because thread does not hold the mutex

Programming Language Bindings

Language	Language Binding File
C	VMCSYN H
Assembler	VMASMSYN MACRO
REXX	VMREXSYN COPY

MutexGetHandle — Get the Handle of a Mutex

MutexGetHandle

retcode

reascode

mutex_handle

mutex_name

mutex_name_length

scope_of_mutex

Purpose

Use the MutexGetHandle function to retrieve the handle of an existing mutex.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

mutex_handle
(output,INT,4) is a variable where the function returns the mutex handle.

mutex_name
(input,CHAR,*mutex_name_length*) is a variable for specifying the name of the mutex.

mutex_name_length
(input,INT,4) is a variable for specifying the length of *mutex_name*. It must be greater than 0 and less than 16MB in length.

scope_of_mutex
(output,INT,4) is a variable where the function returns the scope of the mutex. The valid values are:

vm_syn_process_scope
The current process

vm_syn_session_scope
The current session

Usage Notes

1. A mutex must be created by the MutexCreate function before this function can get its handle. If the mutex is not created, an error is returned.
2. If the threads using a mutex in an application share memory, the handle of a mutex may be stored in the shared memory by the thread creating the mutex. When the other threads in an application require the handle to manipulate the mutex, it may be retrieved from the shared memory. However, if the threads using a mutex in an application do not share memory, the MutexGetHandle function should be invoked to get the handle of the mutex.
3. Mutex handles are kept either per process or session. This depends on the level at which the mutex was created.
4. The search sequence used to find the specified mutex name begins with the process of the function caller. All the mutexes in this process are first searched and then all the session level mutexes are searched. If no match is found, an error is returned.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_syn_success	vm_syn_success	MutexGetHandle completed successfully
vm_syn_error	vm_syn_name_not_found	*Mutex_name* does not exist
vm_syn_error	vm_syn_bad_mutex_name_len	*Mutex_name_length* is out of range.

Programming Language Bindings

Language	Language Binding File
C	VMCSYN H
Assembler	VMASMSYN MACRO
REXX	VMREXSYN COPY

MutexRelease — Release a Mutex

MutexRelease
retcode
reascode
mutex_handle

Purpose

Use the MutexRelease function to release control of a mutex, unblocking one thread waiting to acquire the mutex.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

mutex_handle
(input,INT,4) is a variable for specifying the handle of the mutex to be released. This value is returned by the MutexCreate or MutexGetHandle function.

Usage Notes

1. If a thread holding a mutex terminates without releasing it, the mutex is released and the next thread to acquire the mutex receives a warning return code.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_syn_success	vm_syn_success	MutexRelease completed successfully
vm_syn_error	vm_syn_handle_not_found	Mutex indicated by *mutex_handle* does not exist
vm_syn_error	vm_syn_mutex_not_held	Mutex is not released because thread does not hold the mutex

Programming Language Bindings

Language	Language Binding File
C	VMCSYN H
Assembler	VMASMSYN MACRO
REXX	VMREXSYN COPY

ProcessCheckPoint — Take a Snapshot of the Process State

ProcessCheckPoint

retcode

reascode

suspended_thread_count

blocked_thread_count

Purpose

Use the ProcessCheckPoint function to take a snapshot of the process state in terms of suspended and blocked threads. This function is primarily for use in obtaining diagnostic information.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

suspended_thread_count
(output,INT,4) is a variable where the function returns the number of suspended threads in this process.

blocked_thread_count
(output,INT,4) is a variable where the function returns the number of blocked threads in this process.

Usage Notes

1. Once ProcessCheckPoint is invoked, it cannot be invoked successfully again by this process until both the ProcessQuerySuspended and ProcessQueryBlocked functions have been invoked successfully.
2. Neither ProcessQuerySuspended nor ProcessQueryBlocked may be invoked successfully until ProcessCheckPoint is invoked successfully.
3. The suspended thread count returned by ProcessCheckPoint indicates how much data will be returned by ProcessQuerySuspended.
4. The blocked thread count returned by ProcessCheckPoint indicates how much data will be returned by ProcessQueryBlocked.
5. If there are no blocked and no suspended threads at the time ProcessCheckPoint is issued, ProcessCheckPoint must be reissued before ProcessQueryBlocked or ProcessQuerySuspended can be invoked successfully.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_pro_success	vm_pro_success	ProcessCheckPoint completed successfully
vm_pro_error	vm_pro_ckpt_already_taken	ProcessCheckpoint has already been issued, but one or both of ProcessQuerySuspended and ProcessQueryBlocked have not been issued
vm_pro_error	vm_pro_out_of_storage	There is not enough storage to hold the checkpoint data

Programming Language Bindings

Language	Language Binding File
C	VMCPRO H
Assembler	VMASMPRO MACRO
REXX	VMREXPRO COPY

ProcessGetID — Obtain the ID of a Process

```
ProcessGetID
        retcode
        reascode
        process_ID
        process_name
        process_name_length
```

Purpose

Use the ProcessGetID function to obtain the process ID of an existing process identified by name.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

process_ID
(output,INT,4) is a variable where the function returns the system-supplied identifier of the specified process.

process_name
(input,CHAR,*process_name_length*) is a variable for specifying the name of the process whose process ID is to be obtained.

process_name_length
(input,INT,4) is a variable for specifying the length of *process_name*. The length must be greater than 0 and less than 16MB characters.

Usage Notes

1. To obtain the process ID of the process owning the calling thread, use the ThreadGetID function.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_pro_success	vm_pro_success	ProcessGetID completed successfully
vm_pro_error	vm_pro_bad_name_len	*Process_name_length* is out of range
vm_pro_error	vm_pro_no_such_process	No process has name *process_name*

Programming Language Bindings

Language	Language Binding File
C	VMCPRO H
Assembler	VMASMPRO MACRO
REXX	VMREXPRO COPY

ProcessQueryBlocked — Find Blocked Threads

ProcessQueryBlocked

retcode
reascode
blocked_threads
block_types
object_names
object_name_lengths
actual_name_lengths

Purpose

Use the ProcessQueryBlocked function to find out which threads in the process are blocked and what is blocking them. This function is primarily for use in obtaining diagnostic information.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

blocked_threads
(output,INT,*blocked_thread_count*) is an array of 4-byte variables, in each element of which the function returns the thread ID of a blocked thread in this process. The number of elements in this array is determined by invoking the ProcessCheckPoint function.

block_types
(output,INT,*blocked_thread_count*) is an array of 4-byte variables, in each element of which the function returns the type of block for each blocked thread. The number of elements in this array is determined by invoking the ProcessCheckPoint function. The possible values in this array are as follows:

vm_pro_qreceive_block
Blocking queue receive

vm_pro_qsend_block
Blocking queue send

vm_pro_signal_block
Synchronous event signal

vm_pro_event_wait_block
Event wait

vm_pro_cnv_wait_block
Condition variable wait

vm_pro_mut_acquire_block
Mutex acquire

vm_pro_sem_wait_block
Semaphore wait

object_names
(input,INT,*blocked_thread_count*) is an array of 4-byte variables, each element of which contains the address of a character variable in which the function returns the name of an object. The number of

elements in this array is determined by invoking the ProcessCheckPoint function. The kind of object name returned in each character variable depends on the block type, as follows:

vm_pro_qreceive_block
Queue name

vm_pro_qsend_block
Queue name

vm_pro_signal_block
Event name

vm_pro_event_wait_block
Monitor token

vm_pro_cnv_wait_block
Condition variable name

vm_pro_mut_acquire_block
Mutex name

vm_pro_sem_wait_block
Semaphore name

object_name_lengths
(input,INT,*blocked_thread_count*) is an array of 4-byte variables, each element of which contains the length of the character variable pointed to by the corresponding element in the *object_names* array. The number of elements in this array is determined by invoking the ProcessCheckPoint function.

actual_name_lengths
(output,INT,*blocked_thread_count*) is an array of 4-byte variables, in each element of which the function returns the actual length of the object name returned in the character variable pointed to by the corresponding element in the *object_names* array. The number of elements in this array is determined by invoking the ProcessCheckPoint function.

Usage Notes

1. ProcessQueryBlocked may not be invoked successfully until the ProcessCheckPoint function is invoked successfully.
2. The *blocked_thread_count* returned by ProcessCheckPoint determines the size of the arrays returned by ProcessQueryBlocked. The application must make sure it allocates sufficiently large arrays.
3. If a thread is blocked on a synchronous process level event signal and the event associated with that event signal has been deleted, the actual name length returned is 0.
4. If any of the object names must be truncated, ProcessQueryBlocked is **not** considered to have completed successfully. Thus, after such a completion of ProcessQueryBlocked, ProcessCheckPoint may not be reinvoked successfully.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_pro_success	vm_pro_success	ProcessQueryBlocked completed successfully
vm_pro_warning	vm_pro_name_truncated	At least one object name has been truncated
vm_pro_error	vm_pro_data_not_available	ProcessCheckPoint has not yet been issued for this process

Programming Language Bindings

Language	Language Binding File
C	VMCPRO H

Language	Language Binding File
Assembler	VMASMPRO MACRO
REXX	VMREXPRO COPY

ProcessQuerySuspended — Find Suspended Threads

ProcessQuerySuspended

retcode

retcode

suspended_threads

suspend_counts

Purpose

Use the ProcessQuerySuspended function to find out which threads in the process are suspended and what their suspend counts are. This function is primarily for use in obtaining diagnostic information.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

suspended_threads
(output,INT,*suspended_thread_count*) is an array of 4-byte variables, in each element of which the function returns the thread ID of a suspended thread in this process. The number of elements in this array is determined by invoking the ProcessCheckPoint function.

suspend_counts
(output,INT,*suspended_thread_count*) is an array of 4-byte variables, in each element of which the function returns the suspend count associated with the thread identified in the corresponding element of the *suspended_threads* array. The number of elements in this array is determined by invoking the ProcessCheckPoint function.

Usage Notes

1. ProcessQuerySuspended may not be invoked successfully until the ProcessCheckPoint function is invoked.
2. The *suspended_thread_count* returned by ProcessCheckPoint determines the size of the arrays returned by ProcessQuerySuspended.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_pro_success	vm_pro_success	ProcessQuerySuspended completed successfully
vm_pro_error	vm_pro_data_not_available	ProcessCheckpoint has not yet been issued for this process

Programming Language Bindings

Language	Language Binding File
C	VMCPRO H
Assembler	VMASMPRO MACRO
REXX	VMREXPRO COPY

QueueClose — Close a Queue

QueueClose
retcode
reascode
queue_handle

Purpose

Use the QueueClose function to close an open queue.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

queue_handle
(input,INT,4) is variable for specifying the handle of the queue to be closed.

Usage Notes

1. QueueClose does not disturb the messages in a queue.
2. When closing the queue, the creator receives a warning if messages are left in the queue.
3. The proper way to respond to a *queue was deleted* condition signaled by some other queue function is to close the queue. Thus, the *queue was deleted* condition, while an error if incurred by the other functions, is only a warning if incurred by QueueClose.
4. The primary queue may not be closed.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_ipc_success	vm_ipc_success	QueueClose completed successfully
vm_ipc_warning	vm_ipc_queue_not_empty	Messages still in queue
vm_ipc_error	vm_ipc_bad_handle	Invalid *queue_handle*
vm_ipc_warning	vm_ipc_queue_deleted	Queue was deleted
vm_ipc_error	vm_ipc_primary_queue	Cannot close primary queue
vm_ipc_error	vm_ipc_comm_retry	Communication error — recommend retry
vm_ipc_error	vm_ipc_comm_lost	Communication error — connection lost

Programming Language Bindings

Language	Language Binding File
C	VMCIPC H
Assembler	VMASMIPC MACRO
REXX	VMREXIPC COPY

QueueCreate — Create a Queue

QueueCreate
retcode
reascode
queue_name
queue_name_length
export_level
queue_handle

Purpose

Use the QueueCreate function to create a queue.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

queue_name
(input,CHAR,*queue_name_length*) is a variable for specifying the name of the queue to be created.

queue_name_length
(input,INT,4) is a variable for specifying the length of *queue_name*.

export_level
(input,INT,4) is a variable for specifying the export level, as follows:

vm_ipc_plevel
Process level

vm_ipc_slevel
Session level

vm_ipc_nlevel
Network level

queue_handle
(output,INT,4) is a variable where the function returns the handle for the queue.

Usage Notes

1. QueueCreate opens the queue as well. It returns the handle for the queue in *queue_handle*.
2. Queue handles are per-process.
3. Queue handles are guaranteed never to contain bytes having values corresponding to the EBCDIC code point values for fuzzy match key wildcard characters. This is so that queue handles may be used easily as components of event keys.
4. If the queue already exists and is already open, then the handle on which the process has the queue open is returned in *queue_handle*.
5. If the queue already exists but is not open, then *queue_handle* is set to zero and the queue is not opened.
6. If you plan to send this queue from a second-level interrupt handler, reliable results are obtained only if you:

a. Create the queue at session level.
b. Identify it as a service queue (use QueueIdentifyService).
c. Use the service ID in the *queue_handle* parameter in your interrupt handler's QueueSend call.

For more information, see "Interrupt Handling" on page 287.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_ipc_success	vm_ipc_success	QueueCreate completed successfully
vm_ipc_error	vm_ipc_out_of_storage	Virtual storage unavailable
vm_ipc_error	vm_ipc_already_exists	Queue already exists
vm_ipc_error	vm_ipc_bad_export_level	Unrecognized *export_level*
vm_ipc_error	vm_ipc_bad_name_len	Invalid *queue_name_length*

Programming Language Bindings

Language	Language Binding File
C	VMCIPC H
Assembler	VMASMIPC MACRO
REXX	VMREXIPC COPY

QueueDelete — Delete a Queue

QueueDelete
retcode
reascode
queue_name
queue_name_length
export_level

Purpose

Use the QueueDelete function to delete a queue.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

queue_name
(input,CHAR,*queue_name_length*) is a variable for specifying the name of the queue to be deleted.

queue_name_length
(input,INT,4) is a variable for specifying the length of *queue_name*.

export_level
(input,INT,4) is a variable for specifying the export level, as follows:

vm_ipc_plevel
Process level

vm_ipc_slevel
Session level

vm_ipc_nlevel
Network level

Usage Notes

1. Messages residing in a deleted queue are discarded. The invoker of QueueDelete is given a warning if this occurs.
2. Any thread waiting on the receipt of a discarded message (that is, a thread that used the QueueSendBlock function) is unblocked and given a return and reason code indicating that its message was discarded.
3. Any thread waiting on a message to arrive on the deleted queue (that is, a thread that used the QueueReceiveBlock function) is unblocked and given a return and reason code indicating that the queue was deleted.
4. When a process terminates, queues created by the process are deleted.
5. A process cannot delete its primary queue.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_ipc_success	vm_ipc_success	QueueDelete completed successfully
vm_ipc_error	vm_ipc_bad_export_level	Unrecognized *export_level*
vm_ipc_error	vm_ipc_bad_name_len	Invalid *queue_name_length*
vm_ipc_warning	vm_ipc_msgs_discarded	Messages were discarded
vm_ipc_error	vm_ipc_no_such_queue	Queue does not exist
vm_ipc_error	vm_ipc_not_authorized	Not authorized for operation
vm_ipc_error	vm_ipc_primary_queue	Cannot delete primary queue

Programming Language Bindings

Language	Language Binding File
C	VMCIPC H
Assembler	VMASMIPC MACRO
REXX	VMREXIPC COPY

QueueIdentifyCarrier — Identify a Communication Carrier

QueueIdentifyCarrier

retcode
reascode
carrier_name
carrier_name_length
service_id
old_service_id

Purpose

Use the QueueIdentifyCarrier function to identify a communication carrier for CMS to use for remote IPC operations.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reasons code.

carrier_name
(input,CHAR,*carrier_name_length*) is a variable for specifying the name of the communication carrier.

carrier_name_length
(input,INT,4) is a variable for specifying the length of *carrier_name*.

service_id
(input,INT,4) is a variable for specifying the new service ID.

old_service_id
(output,INT,4) is a variable where the function returns the previous service ID.

Usage Notes

1. Use this function to associate an IPC service queue ID with a communication carrier to be used for remote IPC activity.
2. To map a carrier name to a service ID, the caller's process must own the queue associated with the service ID. QueueIdentifyCarrier fails if a queue is not yet associated with the service ID or if the caller's process does not own said queue.
3. To break the mapping between a communication carrier and a service ID, the caller may specify a service ID of zero. If a queue is still associated with the carrier's service ID, then the caller's process must own said queue to break the mapping. If a queue is no longer associated with the carrier's service ID, then any process may break the mapping.
4. The function returns in *old_service_id* the service ID previously associated with the named communication carrier. It returns zero if the carrier was not previously registered.
5. The service ID must be in the range of -32 to -1.
6. The carrier name may be up to 16MB-1 bytes long.
7. IBM reserves service IDs in the range of -16 to -1 for its own use. Applications attempting to register service queues in that range of service IDs may encounter unpredictable results.
8. A given service ID may be used by only one carrier at a time.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_ipc_success	vm_ipc_success	QueueIdentifyCarrier completed successfully
vm_ipc_error	vm_ipc_bad_name_len	*Carrier_name_length* is invalid
vm_ipc_error	vm_ipc_bad_service_id	Service ID is out of range
vm_ipc_error	vm_ipc_service_undefined	Service is currently not defined
vm_ipc_error	vm_ipc_not_authorized	Service queue is not owned by caller's process
vm_ipc_error	vm_ipc_sid_in_use	Service ID already in use by some other carrier

Programming Language Bindings

Language	Language Binding File
C	VMCIPC H
Assembler	VMASMIPC MACRO
REXX	VMREXIPC COPY

QueueIdentifyService — Identify a Service Queue

```
QueueIdentifyService
        retcode
        reascode
        service_id
        service_queue_name
        service_queue_name_length
        old_queue_name_buffer
        old_queue_name_buffer_size
        old_queue_name_length
```

Purpose

Use the QueueIdentifyService function to identify a service queue.

Parameters

retcode
: (output,INT,4) is a variable where the function stores the return code.

reascode
: (output,INT,4) is a variable where the function stores the reason code.

service_id
: (input,INT,4) is a variable for specifying the service ID.

service_queue_name
: (input,CHAR,*service_queue_name_length*) is a variable for specifying the name of the service queue.

service_queue_name_length
: (input,INT,4) is a variable for specifying the length of *service_queue_name*.

old_queue_name_buffer
: (output,CHAR,*old_queue_name_buffer_size*) is a variable where the function returns the name of the previous service queue.

old_queue_name_buffer_size
: (input,INT,4) is a variable for specifying the size of *old_queue_name_buffer*.

old_queue_name_length
: (output,INT,4) is a variable where the function returns the actual length of the name of the previous service queue.

Usage Notes

1. The service ID must be in the range of -256 to -1. Service IDs are negative to differentiate them from queue handles.
2. IBM reserves service IDs in the range of -16 to -1 and -33 to -128 for its own use. Applications attempting to register service queues in that range of service IDs may encounter unpredictable results.
3. If there previously was no service queue associated with the passed service ID, the function completes successfully and returns 0 in *old_queue_name_length*.
4. If the old service queue name would not fit in the caller's buffer, then the function completes with a warning, as much of the old name as will fit is placed in the caller's buffer, and the old name length is set to the true length (before truncation) of the old name.

5. To break the mapping between a service ID and a service queue name, the caller may specify a service queue name length of 0.
6. A queue can be the service queue for only one service ID at a time.
7. Applications using QueueIdentifyService will cause CMS to use slightly extra storage if they use service IDs outside the range [-32..-1]. If storage is at a premium, service IDs should be kept within the range [-32..-1].

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_ipc_success	vm_ipc_success	QueueIdentifyService completed successfully
vm_ipc_error	vm_ipc_bad_service_id	Service ID is out of range
vm_ipc_error	vm_ipc_bad_name_len	*Service_queue_name_length* is invalid
vm_ipc_error	vm_ipc_no_such_queue	Service queue does not exist
vm_ipc_error	vm_ipc_not_authorized	Service queue is not owned by caller's process
vm_ipc_warning	vm_ipc_old_name_truncated	Old service queue name was truncated
vm_ipc_error	vm_ipc_queue_in_use	Queue is already a service queue
vm_ipc_error	vm_ipc_out_of_storage	Virtual storage unavailable

Programming Language Bindings

Language	Language Binding File
C	VMCIPC H
Assembler	VMASMIPC MACRO
REXX	VMREXIPC COPY

QueueOpen — Open a Queue

QueueOpen
retcode
reascode
queue_name
queue_name_length
search_sequence
search_sequence_length
queue_handle
export_level

Purpose

Use the QueueOpen function to open an existing queue.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

queue_name
(input,CHAR,*queue_name_length*) is a variable for specifying the name of the queue to be opened.

queue_name_length
(input,INT,4) is a variable for specifying the length of *queue_name*.

search_sequence
(input,INT,*search_sequence_length*) is an array of 4-byte variables for specifying the export level search sequence to be used. Each element of this array is one of the following values:

vm_ipc_plevel
Process level

vm_ipc_slevel
Session level

vm_ipc_nlevel
Network level

search_sequence_length
(input,INT,4) is a variable for specifying the number of elements in the *search_sequence* array.

queue_handle
(output,INT,4) is a variable where the function returns the handle of the opened queue.

export_level
(output,INT,4) is a variable where the function returns the export level of the queue. Values returned are:

vm_ipc_plevel
Process level

vm_ipc_slevel
Session level

vm_ipc_nlevel
Network level

Usage Notes

1. Queue handles are per-process.
2. Queue handles are guaranteed never to contain bytes having values corresponding to the EBCDIC code point values for fuzzy match key wildcard characters. This is so that queue handles may be used easily as components of event keys.
3. If the process owning the invoking thread has already opened the queue, then the function completes with a warning and the previously-assigned handle is again returned.
4. If the passed search sequence is null (that is, *search_sequence_length* is zero), then the export levels are searched as described in "Export Level Search Order" on page 29.
5. If *search_sequence_length* is greater than the number of export levels available or is less than zero an error is returned.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_ipc_success	vm_ipc_success	QueueOpen completed successfully
vm_ipc_error	vm_ipc_bad_name_len	Invalid *queue_name_length*
vm_ipc_error	vm_ipc_bad_search_seq_len	Invalid length for *search_sequence* array
vm_ipc_error	vm_ipc_bad_search_seq	Unrecognized *export_level* in search sequence
vm_ipc_warning	vm_ipc_already_open	Queue is already open
vm_ipc_error	vm_ipc_out_of_storage	Virtual storage unavailable
vm_ipc_error	vm_ipc_no_such_queue	Queue not found

Programming Language Bindings

Language	Language Binding File
C	VMCIPC H
Assembler	VMASMIPC MACRO
REXX	VMREXIPC COPY

QueueQuery — Query Waiting Message Count

QueueQuery
retcode
reascode
queue_handle
match_key
match_key_length
message_count

Purpose

Use the QueueQuery function to count the messages waiting in a queue.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

queue_handle
(input,INT,4) is a variable for specifying the handle of the queue to be queried.

match_key
(input,CHAR,*match_key_length*) is a variable for specifying the key to be matched against the keys of messages in the queue.

match_key_length
(input,INT,4) is a variable for specifying the length of *match_key*.

message_count
(output,INT,4) is a variable where the function returns the number of messages that reside in the specified queue and whose key matches the specified match key.

Usage Notes

1. Specify the match-all match key to determine the total number of messages in the queue.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_ipc_success	vm_ipc_success	QueueQuery completed successfully
vm_ipc_error	vm_ipc_bad_handle	Invalid *queue_handle*
vm_ipc_error	vm_ipc_bad_key_len	Invalid *match_key_length*
vm_ipc_error	vm_ipc_not_authorized	Not authorized for operation
vm_ipc_error	vm_ipc_queue_deleted	Queue was deleted

Programming Language Bindings

Language	Language Binding File
C	VMCIPC H
Assembler	VMASMIPC MACRO
REXX	VMREXIPC COPY

QueueReceiveBlock — Receive a Message (Blocking)

QueueReceiveBlock

retcode
reascode
queue_handle
match_key
match_key_length
timeout
message
maximum_length
returned_length
key_offset
key_length
sender_UID
sender_PID
reply_token

Purpose

Use the QueueReceiveBlock function to receive a message from a queue, blocking if necessary until a message becomes available.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the return code.

queue_handle
(input,INT,4) is a variable for specifying the handle of the queue.

match_key
(input,CHAR,*match_key_length*) is a variable for specifying the key of interest to the caller.

match_key_length
(input,INT,4) is a variable for specifying the length of *match_key*.

timeout
(input,INT,4) is a variable for specifying the timeout value.

message
(output,CHAR,*maximum_length*) is a variable for a buffer where the function returns the received message.

maximum_length
(input,INT,4) is a variable for specifying the length of the message buffer.

returned_length
(output,INT,4) is variable where the function returns the actual length of the received message.

key_offset
(output,INT,4) is a variable where the function returns the offset of the key within the message.

key_length
(output,INT,4) is a variable where the function returns the length of the key within the message.

sender_UID
(output,CHAR,8) is a variable for a buffer where the function returns the user ID of the sending process.

sender_PID
(output,INT,4) is a variable where the function returns the process ID of the sending process.

reply_token
(output,INT,4) is a variable where the function returns the reply token.

Usage Notes

1. If the message being received does not fit in the message buffer, then an error is returned and *returned_length* holds the waiting message's true length. As much of the message as will fit is placed in the caller's buffer, and the rest of the returned parameters, except the reply token, are filled in as usual. The message is still available for receipt. The caller should retry the operation with a sufficiently large buffer. It is not guaranteed, though, that a retry will pick up the message that was described; some other thread may have received the message in the interim.
2. The reply token may be used by any thread in the process that received the message.
3. The reply token is valid for only one call to QueueReply.
4. If the sending process is located in the same session as the process executing QueueReceiveBlock, then an asterisk (*) is returned in the *sender_UID* buffer.
5. If the sender did not send the message with the QueueSendReply function, the returned reply token is 0.
6. The timeout period is specified in seconds. The timeout period is always extended upward by CMS to the next 10-second boundary.
7. Depending on the relative timing of the call to QueueReceiveBlock and the ticking of the IPC timer, the call may time out one period later than might otherwise be expected.
8. To wait indefinitely, specify a *timeout* of 0.
9. To match any message key, specify the match-all match key.
10. See "Tips on Constructing Keys" on page 30 for information about composing match keys from binary data.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_ipc_success	vm_ipc_success	QueueReceiveBlock completed successfully
vm_ipc_error	vm_ipc_bad_key_len	Invalid *match_key_length*
vm_ipc_error	vm_ipc_bad_handle	Invalid *queue_handle*
vm_ipc_error	vm_ipc_queue_deleted	Queue was deleted
vm_ipc_error	vm_ipc_not_authorized	Not authorized for operation
vm_ipc_error	vm_ipc_buf_too_small	Message did not fit
vm_ipc_error	vm_ipc_queue_closed	Queue was closed
vm_ipc_error	vm_ipc_bad_timeout	*Timeout* parameter is negative
vm_ipc_warning	vm_ipc_timeout	Function timed out

Programming Language Bindings

Language	Language Binding File
C	VMCIPC H
Assembler	VMASMIPC MACRO
REXX	VMREXIPC COPY

QueueReceiveImmed — Receive a Message (Nonblocking)

QueueReceiveImmed
retcode
reascode
queue_handle
match_key
match_key_length
message
maximum_length
returned_length
key_offset
key_length
sender_UID
sender_PID
reply_token

Purpose

Use the QueueReceiveImmed function to receive a message from a queue, returning immediately if a message is not available.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

queue_handle
(input,INT,4) is a variable for specifying the handle of the queue.

match_key
(input,CHAR,*match_key_length*) is a variable for specifying the key of interest.

match_key_length
(input,INT,4) is a variable for specifying the length of *match_key*.

message
(output,CHAR,*maximum_length*) is a variable for a buffer where the function returns the received message.

maximum_length
(input,INT,4) is a variable for specifying the length of the message buffer.

returned_length
(output,INT,4) is a variable where the function returns the actual length of the received message.

key_offset
(output,INT,4) is a variable where the function returns the offset of the key within the message.

key_length
(output,INT,4) is a variable where the function returns the length of the key within the message.

sender_UID
(output,CHAR,8) is a variable for a buffer where the function returns the user ID of the sending process.

sender_PID
(output,INT,4) is a variable where the function returns the process ID of the sending process.

reply_token
(output,INT,4) is a variable where the function returns the reply token.

Usage Notes

1. If the message being received does not fit in the message buffer, an error is returned and *returned_length* holds the waiting message's length. As much of the message as will fit is placed in the caller's buffer, and the rest of the returned parameters, except the reply token, are filled in as usual. The message is still available for receipt. The caller should retry the operation with a sufficiently large buffer. It is not guaranteed, though, that a retry will pick up the message that was described — some other thread may have received the message in the interim.
2. The reply token may be used by any thread in the process that received the message.
3. The reply token is valid for only one call to QueueReply.
4. If the sending process is located in the same session as the process executing QueueReceiveImmed, then an asterisk (*) is returned in the *sender_UID* buffer.
5. If the sender did not send the message with the QueueSendReply function, the returned reply token is 0.
6. If no message is available, a warning is returned.
7. To match any message key, specify the match-all match key.
8. See "Tips on Constructing Keys" on page 30 for information about composing match keys from binary data.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_ipc_success	vm_ipc_success	QueueReceiveImmed completed successfully
vm_ipc_error	vm_ipc_bad_key_len	Invalid *match_key_length*
vm_ipc_error	vm_ipc_bad_handle	Invalid *queue_handle*
vm_ipc_error	vm_ipc_queue_deleted	Queue was deleted
vm_ipc_error	vm_ipc_not_authorized	Not authorized for operation
vm_ipc_warning	vm_ipc_no_msg_available	No message available
vm_ipc_error	vm_ipc_buf_too_small	Message did not fit

Programming Language Bindings

Language	Language Binding File
C	VMCIPC H
Assembler	VMASMIPC MACRO
REXX	VMREXIPC COPY

QueueReply — Reply to a Message

```
QueueReply
        retcode
        reascode
        reply_token
        message
        message_length
        key_offset
        key_length
```

Purpose

Use the QueueReply function to reply to a message sent with the QueueSendReply function.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

reply_token
(input,INT,4) is a variable for specifying the reply token provided by the QueueReceiveBlock or QueueReceiveImmed function.

message
(input,CHAR,*message_length*) is a variable for specifying the message text.

message_length
(input,INT,4) is a variable for specifying the length of the message text.

key_offset
(input,INT,4) is a variable for specifying the offset into the message text of the first byte of the key.

key_length
(input,INT,4) is a variable for specifying the length of the key within the message text.

Usage Notes

1. CMS guarantees that when this function completes the caller may reuse the message buffer.
2. The sender should be aware that placing pointers, structures, or other address-oriented information in the message may not have the desired effect, especially if the message is destined for a queue located at the network level.
3. Any thread in the process that received the original message may use the reply token to reply to the message.
4. The reply token is valid for only one call to QueueReply.
5. If the reply is sent to a remotely-located network level queue, the reply will fail if the owner of the reply queue has closed the queue since the time it issued QueueSendReply.
6. See "Tips on Constructing Keys" on page 30 for information about including binary data in message keys.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_ipc_success	vm_ipc_success	QueueReply completed successfully
vm_ipc_error	vm_ipc_out_of_storage	Virtual storage unavailable
vm_ipc_error	vm_ipc_bad_msg_len	Invalid *message_length*
vm_ipc_error	vm_ipc_bad_reply_token	Invalid *reply_token*
vm_ipc_error	vm_ipc_bad_kokl	Invalid *key_offset*/*key_length* combination (with respect to *message_length*)
vm_ipc_error	vm_ipc_reply_queue_deleted	Reply queue has been deleted
vm_ipc_error	vm_ipc_comm_retry	Communication error — recommend retry
vm_ipc_error	vm_ipc_comm_lost	Communication error — connection lost

Programming Language Bindings

Language	Language Binding File
C	VMCIPC H
Assembler	VMASMIPC MACRO
REXX	VMREXIPC COPY

QueueSend — Send a Message

QueueSend

retcode
reascode
queue_handle
message
message_length
key_offset
key_length

Purpose

Use the QueueSend function to send a message to an opened queue.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

queue_handle
(input,INT,4) is a variable for specifying handle of the queue.

message
(input,CHAR,*message_length*) is a variable for specifying the message text.

message_length
(input,INT,4) is a variable for specifying the length of the message text.

key_offset
(input,INT,4) is a variable for specifying the offset into the message text of the first byte of the key.

key_length
(input,INT,4) is a variable for specifying the length of the key within the message text.

Usage Notes

1. CMS guarantees that when this function completes the caller may reuse the message buffer.
2. The sender should be aware that placing pointers, structures, or other address-oriented information in the message may not have the desired effect, especially if the message is destined for a queue located at the network level.
3. To send a message to a service queue, use the service ID in place of the queue handle.
4. If you use QueueSend from a second-level interrupt handler (for example, an IUCV exit routine), reliable results are obtained only if:
   a. Your main line identifies the target queue as a service queue.
   b. Your interrupt handler uses the queue's service ID (*not* its handle) in its QueueSend call.

   Messages sent from interrupt handlers then appear to come from the interrupted process. For more information, see “Interrupt Handling” on page 287.
5. See “Tips on Constructing Keys” on page 30 for information about including binary data in message keys.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_ipc_success	vm_ipc_success	QueueSend completed successfully
vm_ipc_error	vm_ipc_out_of_storage	Virtual storage unavailable
vm_ipc_error	vm_ipc_bad_msg_len	Invalid *message_length*
vm_ipc_error	vm_ipc_bad_handle	Invalid *queue_handle*
vm_ipc_error	vm_ipc_bad_kokl	Invalid *key_offset*/*key_length* combination (with respect to *message_length*)
vm_ipc_error	vm_ipc_queue_deleted	Queue was deleted
vm_ipc_error	vm_ipc_comm_retry	Communication error — recommend retry
vm_ipc_error	vm_ipc_comm_lost	Communication error — connection lost
vm_ipc_error	vm_ipc_bad_service_id	Service ID is invalid
vm_ipc_error	vm_ipc_service_undefined	Service ID is undefined

Programming Language Bindings

Language	Language Binding File
C	VMCIPC H
Assembler	VMASMIPC MACRO
REXX	VMREXIPC COPY

QueueSendBlock — Send a Message and Block

QueueSendBlock
retcode
reascode
queue_handle
message
message_length
key_offset
key_length
timeout_period

Purpose

Use the QueueSendBlock function to send a message to an opened queue, blocking the caller until the message is received.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

queue_handle
(input,INT,4) is a variable for specifying the handle of the queue.

message
(input,CHAR,*message_length*) is a variable for specifying the message text.

message_length
(input,INT,4) is a variable for specifying the length of the message text.

key_offset
(input,INT,4) is a variable for specifying the offset into the message text of the first byte of the key.

key_length
(input,INT,4) is a variable for specifying the length of the key within the message text.

timeout_period
(input,INT,4) is a variable for specifying the length of time the thread should be blocked waiting for the message to be received.

Usage Notes

1. CMS guarantees that when this function completes the caller may reuse the message buffer.
2. The sender should be aware that placing pointers, structures, or other address-oriented information in the message may not have the desired effect, especially if the message is destined for a queue located at the network level.
3. If the message is discarded (for example, because of queue deletion), the thread will be unblocked and given a return code indicating that the message was discarded.
4. This function gives two threads a means to rendezvous. One thread should use QueueSendBlock, and the other should use QueueReceiveBlock.

5. The timeout period is specified in seconds. Timeout periods are always extended upward by CMS to the next ten-second boundary.
6. Depending on the relative timing of the call to QueueSendBlock and the ticking of the IPC timer, the call may time out one period later than might otherwise be expected.
7. If the send is to a remote queue, the timeout period is evaluated at both the local kernel and the remote kernel. Communication delays *are* counted as part of the timeout period.
8. A timeout period of 0 indicates that the thread should wait indefinitely.
9. See "Tips on Constructing Keys" on page 30 for information about including binary data in message keys.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_ipc_success	vm_ipc_success	QueueSendBlock completed successfully
vm_ipc_error	vm_ipc_bad_msg_len	Invalid *message_length*
vm_ipc_error	vm_ipc_msg_discarded	Message was discarded
vm_ipc_error	vm_ipc_out_of_storage	Virtual storage unavailable
vm_ipc_error	vm_ipc_bad_handle	Invalid *queue_handle*
vm_ipc_error	vm_ipc_bad_kokl	Invalid *key_offset*/*key_length* combination (with respect to *message_length*)
vm_ipc_error	vm_ipc_queue_deleted	Queue was deleted
vm_ipc_error	vm_ipc_comm_retry	Communication error — recommend retry
vm_ipc_error	vm_ipc_comm_lost	Communication error — connection lost
vm_ipc_error	vm_ipc_bad_timeout	*Timeout_period* is negative
vm_ipc_warning	vm_ipc_timeout	Function timed out

Programming Language Bindings

Language	Language Binding File
C	VMCIPC H
Assembler	VMASMIPC MACRO
REXX	VMREXIPC COPY

QueueSendReply — Send a Message and Request Reply

QueueSendReply
retcode
reascode
queue_handle
message
message_length
key_offset
key_length
reply_queue_handle

Purpose

Use the QueueSendReply function to send a message to an opened queue and specify where a reply should be placed.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

queue_handle
(input,INT,4) is a variable for specifying the handle of the queue to which the message should be sent.

message
(input,CHAR,*message_length*) is a variable for specifying the message text.

message_length
(input,INT,4) is a variable for specifying the length of the message text.

key_offset
(input,INT,4) is a variable for specifying the offset into the message text of the first byte of the key.

key_length
(input,INT,4) is a variable for specifying the length of the key within the message text.

reply_queue_handle
(input,INT,4) is a variable for specifying the handle of the queue into which the reply should be placed.

Usage Notes

1. CMS guarantees that when this function completes the caller may reuse the message buffer.
2. The sender should be aware that placing pointers, structures, or other address-oriented information in the message may not have the desired effect, especially if the message is destined for a queue located at the network level.
3. The receiver of the message need not be able to see or have opened the reply queue.
4. The invoker of this function must have authority to send messages to and receive messages from the reply queue.
5. To send a message to a service queue, use the service ID in place of the queue handle.

6. If the message is being sent to a remotely-located network level queue, the remote program will be able to issue QueueReply successfully only if the owner of the reply queue keeps the queue open until the reply arrives.
7. See "Tips on Constructing Keys" on page 30 for information about including binary data in message keys.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_ipc_success	vm_ipc_success	QueueSendReply completed successfully
vm_ipc_error	vm_ipc_bad_msg_len	Invalid *message_length*
vm_ipc_error	vm_ipc_bad_kokl	Invalid *key_offset*/*key_length* combination (with respect to *message_length*)
vm_ipc_error	vm_ipc_out_of_storage	Virtual storage unavailable
vm_ipc_error	vm_ipc_bad_handle	Invalid *queue_handle*
vm_ipc_error	vm_ipc_bad_reply_handle	Invalid *reply_queue_handle*
vm_ipc_error	vm_ipc_not_authorized	Not authorized for operation
vm_ipc_error	vm_ipc_queue_deleted	Queue was deleted
vm_ipc_error	vm_ipc_reply_queue_deleted	Reply queue was deleted
vm_ipc_error	vm_ipc_bad_service_id	Service ID is out of range
vm_ipc_error	vm_ipc_service_undefined	Service ID is not currently defined
vm_ipc_error	vm_ipc_comm_retry	Communication error — recommend retry
vm_ipc_error	vm_ipc_comm_lost	Communication error — connection lost

Programming Language Bindings

Language	Language Binding File
C	VMCIPC H
Assembler	VMASMIPC MACRO
REXX	VMREXIPC COPY

QueueSignalEvents — Signal Queue Events

QueueSignalEvents
retcode
reascode
queue_handle
signal_flag

Purpose

Use the QueueSignalEvents function to specify whether queue events should be signaled.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

queue_handle
(input,INT,4) is a variable for specifying the handle of the queue.

signal_flag
(input,INT,4) is a variable for specifying whether queue events should be signaled. Valid values are:

vm_ipc_signal_off
Do not signal

vm_ipc_signal_on
Do signal

Usage Notes

1. The name of the IPC event is *VMIPC*. VMIPC is signaled only if the queue owner has requested it by invoking QueueSignalEvents.
2. VMIPC is signaled only in the queue owner's process and only if the queue is open by the owner.
3. The VMIPC event is created with the following attributes:
   - Process-level event
   - Broadcast delivery (flag *vm_evn_broadcast_signals*)
   - Asynchronous signalling (flag *vm_evn_async_signals*)
   - Loose signal limit is zero
   - Timeout period is zero.
4. When signaling is enabled, CMS signals VMIPC each time it delivers a message to an open queue, as follows:
   - Event data = queue handle || message key
   - Event data length = message key length + 4
   - Key offset = 0
   - Key length = event data length.
5. So as to facilitate the use of queue handles in event keys, CMS guarantees that the four bytes of a queue handle will never contain a byte that could be interpreted as a wildcard character in a fuzzy match key.

6. There is some overhead associated with signalling VMIPC. Performance-critical applications should carefully consider whether its use is appropriate.
7. Because the VMIPC event data key contains the message key, the event data key might contain binary data. See “Tips on Constructing Keys” on page 30 for information about the use of binary data in keys.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_ipc_success	vm_ipc_success	QueueSignalEvents completed successfully
vm_ipc_error	vm_ipc_bad_signal_flag	Invalid *signal_flag* value
vm_ipc_error	vm_ipc_bad_handle	Invalid *queue_handle*
vm_ipc_error	vm_ipc_queue_deleted	Queue has been deleted
vm_ipc_error	vm_ipc_not_authorized	Queue is not owned by caller's process

Programming Language Bindings

Language	Language Binding File
C	VMCIPC H
Assembler	VMASMIPC MACRO
REXX	VMREXIPC COPY

SemCreate — Create a Semaphore

SemCreate

retcode

reascode

semaphore_handle

semaphore_name

semaphore_name_length

scope_of_semaphore

initial_value_of_semaphore

Purpose

Use the SemCreate function to create a semaphore. A semaphore is a less-structured synchronization mechanism than a mutex and is primarily used to wait for a condition to occur and resume execution when the condition occurs.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

semaphore_handle
(output,INT,4) is a variable where the function returns the handle of the semaphore.

semaphore_name
(input,CHAR,*semaphore_name_length*) is a variable for specifying the name of the semaphore to be created.

semaphore_name_length
(input,INT,4) is a variable for specifying the length of *semaphore_name*. It must be greater than 0 and less than 16MB in length.

scope_of_semaphore
(input,INT,4) is a variable for specifying the scope of the semaphore. The valid values are:

vm_syn_process_scope
The current process

vm_syn_session_scope
The current session

initial_value_of_semaphore
(input,INT,4) is a variable for specifying the initial value of the semaphore.

Examples of the values that may be used and their behavior are as follows:

0

This should be used when a wait/post mechanism is required.

1

This should be used when a mechanism to protect a critical section is required.

This use of a semaphore places the responsibility of protecting a critical section solely upon the application because of the preconditions placed on the operations defined by a semaphore. CMS does not prevent a thread from signaling a semaphore without first having waited on it. Mutexes do not allow such behavior.

Note: See "Basic Semaphore Processing" on page 49 for examples of these two approaches.

Usage Notes

1. A semaphore can have either process scope or session scope, as follows:
   - A semaphore that has process scope is known only in the process where it is created and can be manipulated only by the threads in that process. The name of such a semaphore must be unique among only the creating process' process-level semaphores.
   - A semaphore that has session scope is known in all the processes in the session and can be manipulated by any of the threads in these processes. The name of such a semaphore must be unique among all session-scope semaphores.
2. A semaphore's scope is fixed for the life of the semaphore.
3. All semaphores created by a process are deleted when a process terminates. Any threads in other processes waiting on such a semaphore are unblocked and given a nonzero return code.
4. CMS supports up to 32,768 session-scope semaphores, mutexes, and condition variables, altogether. Also, for each process, CMS supports up to 32,768 process-scope semaphores, mutexes, and condition variables, altogether.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_syn_success	vm_syn_success	SemCreate completed successfully
vm_syn_error	vm_syn_bad_scope_of_sem	Invalid *scope_of_semaphore* parameter
vm_syn_error	vm_syn_sem_already_exists	Semaphore already exists
vm_syn_error	vm_syn_insufficient_storage	Semaphore not created because storage is not available
vm_syn_error	vm_syn_bad_sem_name_len	*Semaphore_name_length* is out of range
vm_syn_error	vm_syn_limit_reached	Selected scope's limit on total number of synchronization objects has been reached.

Programming Language Bindings

Language	Language Binding File
C	VMCSYN H
Assembler	VMASMSYN MACRO
REXX	VMREXSYN COPY

SemDelete — Delete a Semaphore

SemDelete

retcode

reascode

semaphore_handle

Purpose

Use the SemDelete function to delete a semaphore.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

semaphore_handle
(input,INT,4) is a variable for specifying the handle of the semaphore to be deleted. This value is returned by the SemCreate or SemGetHandle function.

Usage Notes

1. A semaphore can be deleted only by the process that created it. An error is returned if any process in the session other than the creator of the semaphore tries to delete it.
2. If a semaphore is deleted and threads are waiting on it, the blocked threads are unblocked and a return code is given to each thread indicating that the semaphore has been deleted.
3. All semaphores created by a process are deleted when a process terminates. Any threads in other processes waiting on such semaphores are unblocked and given a nonzero return code.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_syn_success	vm_syn_success	SemDelete completed successfully
vm_syn_error	vm_syn_handle_not_found	*Semaphore_handle* does not exist
vm_syn_error	vm_syn_not_sem_creator	Semaphore is not deleted because process is not semaphore creator

Programming Language Bindings

Language	Language Binding File
C	VMCSYN H
Assembler	VMASMSYN MACRO
REXX	VMREXSYN COPY

SemGetHandle — Get the Handle of a Semaphore

SemGetHandle
retcode
reascode
semaphore_handle
semaphore_name
semaphore_name_length
scope_of_semaphore

Purpose

Use the SemGetHandle function to get the handle for an existing semaphore identified by name.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

semaphore_handle
(output,INT,4) is a variable where the function returns the handle of the semaphore.

semaphore_name
(input,CHAR,*semaphore_name_length*) is a variable for specifying the name of the existing semaphore.

semaphore_name_length
(input,INT,4) is a variable for specifying the length of *semaphore_name*. It must be greater than 0 and less than 16MB in length.

scope_of_semaphore
(output,INT,4) is a variable where the function returns the scope of the semaphore. Its values are as follows:

vm_syn_process_scope
The current process

vm_syn_session_scope
The current session

Usage Notes

1. A semaphore must be created by the SemCreate function before this function can get its handle. If the semaphore is not created, an error is returned.
2. This function only returns the handle of the semaphore; it does not test or change the value of the semaphore.
3. If the threads using a semaphore in an application share memory, the handle of a semaphore may be stored in the shared memory by the thread creating the semaphore. When the other threads in an application require the handle to manipulate the semaphore, it may be retrieved from the shared memory. However, if threads using a semaphore in an application do not share memory, the SemGetHandle function should be used to get the handle of the semaphore.
4. Semaphore handles are kept either per-process or per-session. This is dependent upon the level at which the semaphore was created.

5. The search sequence used to find the specified semaphore name begins with the process of the function caller. All the semaphores in this process are searched first and then all the session level semaphores are searched. If no match is found, an error is returned.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_syn_success	vm_syn_success	SemGetHandle completed successfully
vm_syn_error	vm_syn_name_not_found	*Semaphore_name* does not exist
vm_syn_error	vm_syn_bad_sem_name_len	*Semaphore_name_length* is out of range

Programming Language Bindings

Language	Language Binding File
C	VMCSYN H
Assembler	VMASMSYN MACRO
REXX	VMREXSYN COPY

SemQueryValue — Query the Value of a Semaphore

SemQueryValue
retcode
reascode
semaphore_handle
semaphore_value

Purpose

Use the SemQueryValue function to get the value of a semaphore.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

semaphore_handle
(input,INT,4) is a variable for specifying the handle of the semaphore. This value is returned by the SemCreate or SemGetHandle function.

semaphore_value
(output,INT,4) is a variable where the function returns the value of the semaphore.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_syn_success	vm_syn_success	SemQueryValue completed successfully
vm_syn_error	vm_syn_handle_not_found	*Semaphore_handle* does not exist

Programming Language Bindings

Language	Language Binding File
C	VMCSYN H
Assembler	VMASMSYN MACRO
REXX	VMREXSYN COPY

SemReInit — Reinitialize a Semaphore's Value

```
SemReInit
    retcode
    reascode
    semaphore_handle
```

Purpose

Use the SemReInit function to reinitialize the value of a semaphore. All the threads waiting on the semaphore are unblocked.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

semaphore_handle
(input,INT,4) is a variable for specifying the handle of the semaphore to be reinitialized. This value is returned by the SemCreate or SemGetHandle function.

Usage Notes

1. This function may be used by an application in which multiple threads wait for a condition to occur by issuing the SemWait function. When this condition occurs, this function is invoked, reinitializing the semaphore's value and unblocking all the threads waiting on the semaphore.
2. If no threads are waiting on a semaphore and this function is issued, the value of the semaphore is reinitialized. The next thread that issues the SemWait function on this semaphore does not have to wait if the semaphore's new value is greater than 0.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_syn_success	vm_syn_success	SemReInit completed successfully
vm_syn_error	vm_syn_handle_not_found	*Semaphore_handle* does not exist

Programming Language Bindings

Language	Language Binding File
C	VMCSYN H
Assembler	VMASMSYN MACRO
REXX	VMREXSYN COPY

SemSignal — Signal a Semaphore

SemSignal
retcode
reascode
semaphore_handle

Purpose

Use the SemSignal function to signal a semaphore.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

semaphore_handle
(input,INT,4) is a variable for specifying the handle of the semaphore to be signaled. This value is returned by the SemCreate or SemGetHandle function.

Usage Notes

1. This function increments the semaphore's value by 1. If the result is greater than zero, the wait queue is already empty, and the effect of the SemSignal function is that the next issuer of the SemWait function finds the semaphore available.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_syn_success	vm_syn_success	SemSignal completed successfully
vm_syn_error	vm_syn_handle_not_found	*Semaphore_handle* does not exist

Programming Language Bindings

Language	Language Binding File
C	VMCSYN H
Assembler	VMASMSYN MACRO
REXX	VMREXSYN COPY

SemWait — Wait on a Semaphore

SemWait
retcode
reascode
semaphore_handle

Purpose

Use the SemWait function to wait on a semaphore.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

semaphore_handle
(input,INT,4) is a variable for specifying the handle of the semaphore to be waited on. This value is returned by the SemCreate or SemGetHandle function.

Usage Notes

1. This function decrements the semaphore's value by 1. If the result is greater than or equal to 0, the thread issuing this function remains running. If the value is less than 0, the thread is placed on the wait queue associated with the semaphore in a FIFO manner.
2. If a thread waiting on a semaphore is terminated (this is accomplished through the use of ThreadDelete), the semaphore's value is incremented and the thread is removed from the wait list.
3. If a semaphore is deleted, any thread waiting on the semaphore is unblocked and given an error return code.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_syn_success	vm_syn_success	SemWait completed successfully
vm_syn_error	vm_syn_handle_not_found	*Semaphore_handle* does not exist
vm_syn_error	vm_syn_sem_deleted	Semaphore was deleted

Programming Language Bindings

Language	Language Binding File
C	VMCSYN H
Assembler	VMASMSYN MACRO
REXX	VMREXSYN COPY

ThreadCreate — Create a Thread

ThreadCreate

retcode
reascode
thread_ID
thread_flag
thread_flag_size
priority_offset
entry_point_address
parameter_list
parameter_list_size

Purpose

Use the ThreadCreate function to create a new thread in the caller's process and specify the address of the entry point at which it is to begin execution.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

thread_ID
(output,INT,4) is a variable where the function returns the system-supplied identifier of the new thread.

thread_flag
(input,INT,*thread_flag_size*) is an array of 4-byte variables, each element of which contains information about how the new thread is to be created and managed. Only one option from each of the following sets may be specified. If no option from a particular set is specified, the default is taken.

- The dispatching class of the new thread.

  vm_pro_new_class
  Distinct class by itself (the default)

  vm_pro_my_class
  Same class as caller

- Whether the parameter list should be copied.

  vm_pro_copy_plist
  Do copy the parameter list (the default)

  vm_pro_no_copy_plist
  Do not copy the parameter list

thread_flag_size
(input,INT,4) is a variable for specifying the number of elements in the *thread_flag* array.

priority_offset
(input,INT,4) is a variable for specifying the offset that should be added to the priority of the calling thread to determine the priority of the new thread.

entry_point_address
(input,INT,4) is a variable for specifying the address of the first instruction to be executed by the new thread.

parameter_list
(input,INT,*parameter_list_size*) is an array of 4-byte variables that contains the parameter list passed to *entry_point_address*.

parameter_list_size
(input,INT,4) is a variable for specifying the number of elements in the *parameter_list* array.

Usage Notes

1. Variable *priority_offset* sets the new thread's priority relative to the caller's priority. To create a lower-priority thread, use a negative value; to create a higher-priority thread, use a positive value. The passed value must result in a thread priority in the range of 0 to 32767.
2. For assembler programs, the following entry conditions are provided for each thread:

Entry	Description
R1	Address of *parameter_list* array, or a copy of that array if option *vm_pro_copy_plist* was specified
R13	Address of a 72-byte save area in which the thread may save the registers (that is, STM 14,12,12(13))
R14	Return address
R15	Thread's entry point
Addressing mode	Addressing mode of the creator
PSW key	If this is the first thread in the process, it is X'E', otherwise, it is the PSW key of the creator.
Interrupts	Enabled

   In addition, if the virtual machine is XC mode, then the address space mode is primary and the access registers are those of the creating thread.
3. See "Writing Multitasking Applications in Assembler" on page 85 for a discussion of the entry conditions provided to the APPLMAIN thread.
4. For assembler programs, each thread that deletes itself implicitly by returning to the kernel (including APPLMAIN) is expected to provide register values as follows:

   R15
   Thread's return code
5. The ThreadCreate function allows a parameter list to be passed to the new thread. It is the caller's responsibility to construct the *parameter_list* array in a format appropriate to *entry_point_address*. The parameter list must be built according to the parameter passing conventions of the programming language being used.

   For assembler environment programs, the format of the parameter list is left to the application programmer. The programmer must ensure that the parameter list is in the format expected by the thread entry point.

   For C programs, the programmer must build the parameter list as an array of addresses of parameter values. An example illustrates this. Suppose there exists a C function `f` as follows:

```
void f ( a, b, c )
   int a;
   int b;
   int * c;
{
   *c = a + b;
}
```

In the usual case, a C caller would invoke function `f` as follows:

```
extern void f (int, int, int *);
int a, b, c;
a = 2;
b = 3
f (a, b, &c;);
```

But suppose that instead of invoking `f` by call, the programmer wants to invoke it using ThreadCreate. In that case, parameter passing is a bit different, in that parameters are passed by a pointer to a structure. Changes are needed in both the calling code and in the function code.

First, function `f` must be changed to:

```
void func(struct { int a; int b; int * c; } *p)
 {
  *(p->c) = p->a + p->b;
 }
```

Then, to invoke `f` using ThreadCreate, the code would be as follows:

```
extern void f (int, int, int *);
int rc, re, thread_id;
int parameter_list[3], parameter_list_length;
int a, b, c;
/*  initialize parameters for f  */
a = 2;
b = 3;
/*  build the parameter list - R1 will point to a copy of  */
/*  this array when the kernel starts f                    */
parameter_list[0] = (int) &a;
parameter_list[1] = (int) &b;
parameter_list[2] = (int) &c;
parameter_list_length = 3;
/*  call ThreadCreate  */
ThreadCreate
(
   &rc,                      /*  return code           */
   &re,                      /*  reason code           */
   &tid,                     /*  thread ID             */
   &rc,                      /*  placeholder for flags */
   0,                        /*  number of flag words  */
   0,                        /*  same priority as me   */
   f,                        /*  thread's entry point  */
   parameter_list,           /*  parameter list array  */
   parameter_list_length     /*  plist array length    */
);
```

6. When using a parameter list to pass data to a thread, the programmer must keep in mind that if the *vm_pro_no_copy_plist* option is used, correct results will be obtained only if the elements of the *parameter_list* array continue to be valid until the new thread is done using them. If the caller reuses the *parameter_list* array before the new thread finishes with the parameters, then incorrect results will be obtained. Further, if the storage in which *parameter_list* resides is deallocated too early, incorrect results may also occur.

   Note that if *parameter_list* contains addresses of other data items, correct results will be achieved only if said other data items continue to be valid until the new thread is done using them, regardless of whether CMS copied the parameter list. Because CMS cannot discern the meanings of individual entries in the *parameter_list* array, it cannot assist in keeping those data items valid.

   Finally, note that if parameter list copying is not used, then CMS passes the address of *parameter_list* directly to the new thread. This reduces the parameter array to nothing more than storage shared between the two threads. Though there are more direct means for establishing shared storage between threads, this mechanism may prove helpful in some situations.

7. When parameter list copying is requested, the *parameter_list* array must contain no more than 128 entries.

8. If the creating thread is executing in access-register mode, the new thread's access registers will contain the same values as those of its creator. The new thread will start execution in primary mode. To use the access registers, the new thread must switch to access-register mode.

9. The *thread_ID* is guaranteed not to contain bytes whose values correspond to the code points for key wildcard characters.
10. ThreadCreate cannot be called from a REXX exec.
11. The entry point where execution of the new thread is to begin cannot be VMSTART, the multitasking initialization entry point.
12. **Attention:** If using the LOAD macro to obtain the thread entry point address, you must not specify COMPSWT ON. Also, you must not use the COMPSWT ON macro function before loading a thread's object. Using COMPSWT ON alters the entry point address of a separately loadable thread object. This will lead to failures running the thread.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_pro_success	vm_pro_success	ThreadCreate completed successfully
vm_pro_error	vm_pro_bad_flags	*Thread_flag* array contains an unrecognized value
vm_pro_error	vm_pro_bad_priority	*Priority_offset* results in an out-of-range priority for the new thread
vm_pro_error	vm_pro_bad_plist_len	*Parameter_list_size* is less than 0
vm_pro_error	vm_pro_bad_flags_len	*Thread_flag_size* is less than 0
vm_pro_error	vm_pro_out_of_storage	Out of storage
vm_pro_error	vm_pro_plist_too_big	Parameter list is too large to be copied
vm_pro_error	vm_pro_not_mt	The application is not part of a process initiated by the multitasking initialization routine VMSTART

Programming Language Bindings

Language	Language Binding File
C	VMCPRO H
Assembler	VMASMPRO MACRO
REXX	VMREXPRO COPY

ThreadDelay — Delay This thread

ThreadDelay
retcode
reascode
interval

Purpose

Use the ThreadDelay function to delay the invoking thread for the interval specified. The time interval is specified in 4 bytes of milliseconds.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

interval
(input,INT,4) is a variable for specifying the length of the interval, in milliseconds, that the thread is to be delayed. The interval must be greater than zero.

Usage Notes

1. All delayed threads in a process are awakened by the invocation of TimerStopAll.
2. A timer that is set for an extremely small interval (small enough to possibly expire before the start processing is complete) could produce unpredictable results.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_pro_success	vm_pro_success	ThreadDelay completed successfully
vm_pro_error	vm_pro_interval_invalid	Invalid *interval* specification
vm_pro_error	vm_pro_out_of_storage	Out of storage

Programming Language Bindings

Language	Language Binding File
C	VMCPRO H
Assembler	VMASMPRO MACRO
REXX	VMREXPRO COPY

ThreadDelete — Delete Threads

ThreadDelete
retcode
reascode
thread_ID

Purpose

Use the ThreadDelete function to terminate the execution of either an individual thread or all threads other than the calling thread in the calling thread's process.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

thread_ID
(input,INT,4) is a variable for specifying the thread to be deleted, as follows:

-1
All threads in the caller's process except the calling thread

0
The calling thread

***n*>0**
Thread *n* in the caller's process

Usage Notes

1. If the calling thread deletes itself, it never regains control.
2. A thread may also delete itself by returning control to the system.
3. If the last thread in the application process terminates itself by calling ThreadDelete, CMS terminates and passes a return code of 0 to CMS.
4. If the last thread in the application process terminates itself by returning to the system, the CMS application terminates and the return code passed to CMS is the return code (that is, R15) of the last thread.
5. Termination by ThreadDelete is considered to be normal termination; no error event is signaled for the deleted threads, and no recovery from the termination request is possible.
6. Control does not return to the caller until the specified threads are deleted.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_pro_success	vm_pro_success	ThreadDelete completed successfully
vm_pro_error	vm_pro_no_such_thread	No thread in the caller's process has ID *thread_ID*

Programming Language Bindings

Language	Language Binding File
C	VMCPRO H
Assembler	VMASMPRO MACRO
REXX	VMREXPRO COPY

ThreadGetID — Obtain the ID of the Calling Thread

```
ThreadGetID
        retcode
        reascode
        thread_ID
        process_ID
```

Purpose

Use the ThreadGetID function to obtain the thread ID of the calling thread and the process ID of the process that owns the calling thread.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

thread_ID
(output,INT,4) is a variable where the function returns the system-supplied identifier of the calling thread.

process_ID
(output,INT,4) is a variable where the function returns the system-supplied identifier of the process that owns the calling thread.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_pro_success	vm_pro_success	ThreadGetID completed successfully

Programming Language Bindings

Language	Language Binding File
C	VMCPRO H
Assembler	VMASMPRO MACRO
REXX	VMREXPRO COPY

ThreadQueryDispatchClass — Query a Thread's Dispatch Class

ThreadQueryDispatchClass
retcode
reascode
thread_ID
classmate_ID_list
classmate_ID_list_size
classmate_ID_list_count

Purpose

Use the ThreadQueryDispatchClass function to obtain a list of the threads in the caller's process residing in the same dispatch class as the specified thread. This function is primarily for use in obtaining diagnostic information.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

thread_ID
(input,INT,4) is a variable for specifying the system-supplied identifier of the thread for which the dispatch class list is to be obtained. A value of 0 may be used to represent the calling thread.

classmate_ID_list
(output,INT,*classmate_ID_list_size*) is an array of 4-byte variables, in each element of which the function returns the thread ID of a thread residing both in the caller's process and in the same dispatch class as the specified thread.

classmate_ID_list_size
(input,INT,4) is a variable for specifying the number of elements in the *classmate_ID_list* array.

classmate_ID_list_count
(output,INT,4) is a variable where the function returns the number of threads in the specified thread's dispatch class.

Usage Notes

1. If the value returned in *classmate_ID_list_count* is less than or equal to the value provided in *classmate_ID_list_size*, then the first *classmate_ID_list_count* entries in the *classmate_ID_list* array contain the complete set of classmate IDs and the remaining array elements are unchanged; otherwise, only the first *classmate_ID_list_size* classmate IDs are returned.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_pro_success	vm_pro_success	ThreadQueryDispatchClass completed successfully
vm_pro_error	vm_pro_no_such_thread	No thread in the calling thread's process has ID *thread_ID*

Return Code	Reason Code	Meaning
vm_pro_error	vm_pro_bad_dspclass_len	*Classmate_ID_list_size* is less than zero

Programming Language Bindings

Language	Language Binding File
C	VMCPRO H
Assembler	VMASMPRO MACRO
REXX	VMREXPRO COPY

ThreadQueryEntryPoint — Query a Thread's Entry Point

ThreadQueryEntryPoint
retcode
reascode
thread_ID
entry_point_address

Purpose

Use the ThreadQueryEntryPoint function to obtain the entry point of the specified thread in the caller's process. This function is primarily for use in obtaining diagnostic information.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

thread_ID
(input,INT,4) is a variable for specifying the system-supplied identifier of the thread for which the entry point address is to be obtained. A value of 0 may be used to represent the calling thread.

entry_point_address
(output,INT,4) is a variable where the function returns the address of the instruction at which the specified thread began execution.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_pro_success	vm_pro_success	ThreadQueryEntryPoint completed successfully
vm_pro_error	vm_pro_no_such_thread	No thread in the calling thread's process has ID *thread_ID*

Programming Language Bindings

Language	Language Binding File
C	VMCPRO H
Assembler	VMASMPRO MACRO
REXX	VMREXPRO COPY

ThreadQueryParameterList — Query a Thread's Parameter List

ThreadQueryParameterList

retcode
reascode
thread_ID
parameter_list
parameter_list_size
parameter_list_count

Purpose

Use the ThreadQueryParameterList function to obtain the current contents of the parameter list of the specified thread in the caller's process. This function is primarily for use in obtaining diagnostic information.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

thread_ID
(input,INT,4) is a variable for specifying the system-supplied identifier of the thread for which the parameter list is obtained. A value of 0 may be used to represent the calling thread.

parameter_list
(output,INT,4) is an array of 4-byte variables, in each element of which the function returns the current value of a parameter from the parameter list of the specified thread.

parameter_list_size
(input,INT,4) is a variable for specifying the number of elements in the *parameter_list* array.

parameter_list_count
(output,INT,4) is a variable where the function returns the number of parameters in the specified thread's parameter list.

Usage Notes

1. If the value returned in *parameter_list_count* is less than or equal to the value provided in *parameter_list_size*, then the first *parameter_list_count* entries in the *parameter_list* array contain the complete current contents of the specified thread's parameter list and the remaining array elements are unchanged; otherwise only the first *parameter_list_size* entries are returned.
2. Callers of ThreadQueryParameterList always see the current contents of the specified thread's parameter list. If a thread's parameter list is changed after the thread is created, subsequent calls to ThreadQueryParameterList reflect the change.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_pro_success	vm_pro_success	ThreadQueryParameterList completed successfully

Return Code	Reason Code	Meaning
vm_pro_error	vm_pro_no_such_thread	No thread in the calling thread's process has ID *thread_ID*
vm_pro_error	vm_pro_bad_plist_len	*Parameter_list_size* is less than 0

Programming Language Bindings

Language	Language Binding File
C	VMCPRO H
Assembler	VMASMPRO MACRO
REXX	VMREXPRO COPY

ThreadQueryPriority — Query a Thread's Priority

ThreadQueryPriority
retcode
reascode
thread_ID
thread_priority

Purpose

Use the ThreadQueryPriority function to obtain the priority of the specified thread relative to other threads in the calling thread's process.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

thread_ID
(input,INT,4) is a variable for specifying the system-supplied identifier of the thread for which the priority is obtained. A value of 0 may be used to represent the calling thread.

thread_priority
(output,INT,4) is a variable where the function returns the priority of the specified thread.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_pro_success	vm_pro_success	ThreadQueryPriority completed successfully
vm_pro_error	vm_pro_no_such_thread	No thread in the calling thread's process has ID *thread_ID*

Programming Language Bindings

Language	Language Binding File
C	VMCPRO H
Assembler	VMASMPRO MACRO
REXX	VMREXPRO COPY

ThreadQuerySuspendCount — Query a Thread's Suspend Count

ThreadQuerySuspendCount
retcode
reascode
thread_ID
suspend_count

Purpose

Use the ThreadQuerySuspendCount function to obtain the suspend count of the specified thread in the caller's process. This function is primarily for use in obtaining diagnostic information.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

thread_ID
(input,INT,4) is a variable for specifying the system-supplied identifier of the thread for which the suspend count is to be obtained. A value of 0 may be used to represent the calling thread.

suspend_count
(output,INT,4) is a variable where the function returns the suspend count of the specified thread.

Usage Notes

1. A thread's suspend count is never negative.
2. A thread is not considered dispatchable if its suspend count is positive.
3. To increment a thread's suspend count, use the ThreadSuspend function.
4. To decrement a thread's suspend count, use the ThreadResume function.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_pro_success	vm_pro_success	ThreadQuerySuspendCount completed successfully
vm_pro_error	vm_pro_no_such_thread	No thread in the calling thread's process has ID *thread_ID*

Programming Language Bindings

Language	Language Binding File
C	VMCPRO H
Assembler	VMASMPRO MACRO
REXX	VMREXPRO COPY

ThreadQueryUserData — Query User Data Word

```
ThreadQueryUserData
        retcode
        reascode
        user_data_word
```

Purpose

Use the ThreadQueryUserData function to query a thread's user data word.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

user_data_word
(output,INT,4) is a variable where the function returns the value of the user data word.

Usage Notes

1. This function can query the user word of only the calling thread.
2. One example of the use of a user word is the anchoring of control blocks maintained on a thread basis. By placing the address of the control block in the thread user word, any code running on the thread can easily retrieve the control block's address.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_pro_success	vm_pro_success	ThreadQueryUserData completed successfully

Programming Language Bindings

Language	Language Binding File
C	VMCPRO H
Assembler	VMASMPRO MACRO
REXX	VMREXPRO COPY

ThreadResume — Decrement a Thread's Suspend Count

ThreadResume
retcode
reascode
thread_ID

Purpose

Use the ThreadResume function to decrement the suspend count of one or more threads in the caller's process.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

thread_ID
(input,INT,4) is a variable for specifying the thread whose suspend count should be decremented, as follows:

-1
All threads in the caller's process except the calling thread

***n*>0**
Thread *n* in the caller's process

Usage Notes

1. Though it results in a null operation, a *thread_ID* of 0 may be used to represent the calling thread.
2. If the suspend count of an affected thread was 0, it remains 0 and no error is returned.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_pro_success	vm_pro_success	ThreadResume completed successfully
vm_pro_error	vm_pro_no_such_thread	No thread in the caller's process has ID *thread_ID*

Programming Language Bindings

Language	Language Binding File
C	VMCPRO H
Assembler	VMASMPRO MACRO
REXX	VMREXPRO COPY

ThreadSetDispatchClass — Set the Dispatching Class of Threads

ThreadSetDispatchClass

retcode
reascode
thread_ID

Purpose

Use the ThreadSetDispatchClass function to set the dispatching class affiliation of a specified thread or all threads other than the calling thread in the caller's process.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

thread_ID
(input,INT,4) is a variable for specifying the threads whose dispatch class should be set, as follows:

-1
All threads in the caller's process should be placed into the dispatch class of the calling thread.

0
The calling thread should be placed into a dispatch class of its own.

***n*>0**
Thread *n* in the caller's process should be placed into the dispatch class of the calling thread.

Usage Notes

1. A *dispatch class* is a set of threads for which it is guaranteed that only one thread executes at a time. In other words, threads in the same dispatch class are never dispatched in parallel on multiple virtual processors.
2. It is not guaranteed that each thread of a given dispatch class is dispatched on the same virtual processor.
3. Specifying your own nonzero *thread_ID* is semantically equivalent to specifying a *thread_ID* of 0.
4. The circumstances under which a running thread loses control to another in the same dispatch class are the running thread:
   - Enters a voluntary wait state (for example, for a message on a queue or for an event to be signaled)
   - Uses ThreadYield to give control to some other thread in its dispatch class
   - Is deleted
   - Is suspended.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_pro_success	vm_pro_success	ThreadSetDispatchClass completed successfully

Return Code	Reason Code	Meaning
vm_pro_error	vm_pro_no_such_thread	No thread in the calling thread's process has ID *thread_ID*
vm_pro_error	vm_pro_out_of_storage	Out of storage

Programming Language Bindings

Language	Language Binding File
C	VMCPRO H
Assembler	VMASMPRO MACRO
REXX	VMREXPRO COPY

ThreadSetPriority — Set the Dispatching Priority of Threads

ThreadSetPriority

retcode

reascode

thread_ID

priority

flags

Purpose

Use the ThreadSetPriority function to set the dispatching priority of a specified thread or all threads other than the calling thread in the current process.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

thread_ID
(input,INT,4) is a variable for specifying the thread whose priority should be adjusted, as follows:

-1
All threads in the caller's process except the calling thread

0
The calling thread

***n*>0**
Thread *n* in the caller's process

priority
(input,INT,4) is a variable for specifying the absolute priority or relative priority of the specified thread. The relative priority is the offset that should be added to the current priority to determine the new priority.

flags
(input,INT,4) is a variable for specifying whether *priority* is to be treated as an absolute or relative value, as follows:

vm_pro_absolute_priority
priority is absolute

vm_pro_relative_priority
priority is relative

Usage Notes

1. To decrease a thread's priority, use a negative priority. To increase it, use a positive value.
2. The value passed in *priority* must result in a thread priority in the range of 0 to 32767. If *thread_ID* is -1 and this condition is encountered, the priorities of some threads may not be adjusted.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_pro_success	vm_pro_success	ThreadSetPriority completed successfully
vm_pro_error	vm_pro_no_such_thread	No thread in the caller's process has ID *thread_ID*
vm_pro_error	vm_pro_bad_priority	*Priority* resulted in an out-of-range thread priority
vm_pro_error	vm_pro_bad_flags	*Flags* contains an invalid value

Programming Language Bindings

Language	Language Binding File
C	VMCPRO H
Assembler	VMASMPRO MACRO
REXX	VMREXPRO COPY

ThreadSetUserData — Set User Data Word

ThreadSetUserData
retcode
reascode
user_data_word

Purpose

Use the ThreadSetUserData function to set a thread's user data word.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

user_data_word
(input,INT,4) is a variable for specifying the new value for the user word.

Usage Notes

1. This function can set the user word of only the calling thread.
2. One example of the use of a user word is the anchoring of control blocks maintained on a thread basis. By placing the address of the control block in the thread user word, any code running on the thread can easily retrieve the control block's address.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_pro_success	vm_pro_success	ThreadSetUserData completed successfully

Programming Language Bindings

Language	Language Binding File
C	VMCPRO H
Assembler	VMASMPRO MACRO
REXX	VMREXPRO COPY

ThreadSuspend — Increment a Thread's Suspend Count

ThreadSuspend
retcode
reascode
thread_ID

Purpose

Use the ThreadSuspend function to increment the suspend count of one or more threads in the caller's process.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

thread_ID
(input,INT,4) is a variable for specifying the thread whose suspend count should be incremented, as follows:

-1
All threads in the caller's process except the calling thread

0
The calling thread

***n*>0**
Thread *n* in the caller's process

Usage Notes

1. A thread is considered undispatchable if its suspend count is positive.
2. A suspend count is never negative.
3. ThreadSuspend does not cause a suspend count to overflow (wrap from largest positive to smallest negative number).
4. ThreadSuspend does not return to its caller until all affected threads are suspended.
5. If the calling thread suspends itself, it does not regain control until it is resumed by some other thread in its process.
6. Suspending the current thread in a dispatch class causes that thread to lose control to some other thread in its dispatch class. See the usage notes in “ThreadSetDispatchClass — Set the Dispatching Class of Threads” on page 244 for more information.
7. When a thread is suspended, the suspension is not applied to the thread until the thread attempts to leave the CMS multitasking kernel.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_pro_success	vm_pro_success	ThreadSuspend completed successfully

Return Code	Reason Code	Meaning
vm_pro_error	vm_pro_no_such_thread	No thread in the caller's process has ID *thread_ID*

Programming Language Bindings

Language	Language Binding File
C	VMCPRO H
Assembler	VMASMPRO MACRO
REXX	VMREXPRO COPY

ThreadYield — Yield Control to Another Thread

ThreadYield
retcode
reascode
classmate_ID

Purpose

Use the ThreadYield function to yield control to another thread without becoming undispatchable.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

classmate_ID
(input,INT,4) is a variable for specifying the thread ID of the thread in the caller's dispatch class to which this thread is yielding.

Usage Notes

1. A *classmate_ID* value of 0 may be used to indicate that the highest priority dispatchable thread in the caller's dispatch class should be the next thread dispatched from the caller's dispatch class.
2. A *classmate_ID* value of -1 may be used to indicate that the caller wishes to retain control of its own dispatch class while letting threads from other dispatch classes run.
3. See the usage notes in ThreadSetDispatchClass for more information about the conditions under which the running thread in a dispatch class may lose control to some other thread in its class.
4. Regardless of the value of *classmate_ID*, if ThreadYield has completed successfully, threads from other dispatch classes might have run as a result of the call.
5. If ThreadYield fails, no thread switch happens.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_pro_success	vm_pro_success	ThreadYield completed successfully
vm_pro_error	vm_pro_no_such_thread	No thread residing in both the caller's process and the caller's dispatch class has ID *thread_ID*
vm_pro_error	vm_pro_not_dispatchable	The thread having ID *thread_ID* is not currently dispatchable

Programming Language Bindings

Language	Language Binding File
C	VMCPRO H
Assembler	VMASMPRO MACRO

Language	Language Binding File
REXX	VMREXPRO COPY

TimerStartInt — Start an Interval Timer

TimerStartInt

retcode
reascode
token
timertype
cycle
intervalunits
interval
userword

Purpose

Use the TimerStartInt function to start a timer. The timer can be a real timer or a CPU timer. The time interval is specified in 4 bytes of milliseconds or microseconds.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

token
(output,INT,4) is a variable where the function returns a timer identifier token. This token is uniquely associated with the timer set by this TimerStartInt call and is used by many of the other timer functions.

timertype
(input,INT,4) is a variable for specifying whether the timer is a real timer or a CPU timer. Valid values are:

vm_tmr_timertype_real
real

vm_tmr_timertype_cpu
CPU

cycle
(input,INT,4) is a variable for specifying whether the timer is a single timer, meaning that it expires after the specified time interval, or a cyclical timer, meaning that it continues to expire at regular intervals until specifically stopped. Valid values are:

vm_tmr_cycle_single
single

vm_tmr_cycle_cyclical
cyclical

intervalunits
(input,INT,4) is a variable for specifying the units in which the time interval is specified. Valid values are:

vm_tmr_intunit_micro
microseconds

vm_tmr_intunit_milli
milliseconds

interval
(input,INT,4) is a variable for specifying the time interval. The maximum interval allowed is 2^{31}-1.

userword
(input,CHAR,8) is a variable for specifying the user data to be associated with this timer.

Usage Notes

1. The application must use Event Management facilities to be notified of timer expiration. For further information, see Chapter 7, "Timer Services," on page 59, and Chapter 3, "Event Management," on page 17.
2. The maximum 4-byte interval of 2^{31}-1 = X'7FFFFFFF' allows for a specification of up to 2,147,483,647 milliseconds or microseconds. If milliseconds are used, the maximum time interval is approximately 24.85 days. If microseconds are used, the maximum time interval is approximately 35.8 minutes.
3. Timers set for extremely small intervals (that is, small enough that they could expire before the start processing is complete) could produce unpredictable results.
4. TimerTest and TimerStop should be used to test or stop a particular timer started with TimerStartInt. TimerTestMicros and TimerStopMicros cannot be used on timers started with TimerStartInt.
5. If the requested interval would cause the clock comparator to wrap past (or to) zero, (if programming support uses the standard epoch of January 1, 1900, 0:00 A.M. GMT/UTC, this will occur sometime soon after the year 2041), the timer is set for the maximum clock comparator interval and, therefore, would expire earlier than originally requested. A warning return code and *interval exceeds limit* reason code is issued when this occurs.
6. The timer token is guaranteed not to contain bytes corresponding to the code points for key wildcard characters.
7. *Userword* is part of the event data key for the VMTIMER event. Applications using *userword* as part of the key passed to EventMonitorCreate may incur unexpected results if *userword* is binary data. See "Tips on Constructing Keys" on page 30 for more information.
8. If a userword of all binary zeros is specified, **no** userword will be included in the signal key when Timer Services signals this VMTIMER event.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_tmr_success	vm_tmr_success	TimerStartInt completed successfully
vm_tmr_warning	vm_tmr_interval_exceeds_limit	Interval requested would cause the clock comparator to exceed its maximum value. The timer was instead set to the maximum clock comparator value.
vm_tmr_error	vm_tmr_type_invalid	Invalid *timertype* specification
vm_tmr_error	vm_tmr_cycle_invalid	Invalid *cycle* specification
vm_tmr_error	vm_tmr_intervalunit_invalid	Invalid *intervalunits* specification
vm_tmr_error	vm_tmr_interval_invalid	Invalid *interval* specification; interval must be greater than 0 and less than 2^{31}.
vm_tmr_error	vm_tmr_insufficient_storage	Out of storage

Programming Language Bindings

Language	Language Binding File
C	VMCTMR H
Assembler	VMASMTMR MACRO
REXX	VMREXTMR COPY

TimerStartMicros — Start an Interval Timer

TimerStartMicros

retcode
reascode
token
timertype
cycle
interval
userword

Purpose

Use the TimerStartMicros function to start a timer. The timer can be a real timer or a CPU timer. The time interval is specified in 8 bytes of microseconds.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

token
(output,INT,4) is a variable where the function returns a timer identifier token. This token is uniquely associated with the timer set by this TimerStartMicros call and is used by many of the other timer functions.

timertype
(input,INT,4) is a variable for specifying whether the timer is a real timer or a CPU timer. Valid values are:

vm_tmr_timertype_real
real

vm_tmr_timertype_cpu
CPU

cycle
(input,INT,4) is a variable for specifying whether the timer is a single timer, meaning that it expires after the specified time interval, or a cyclical timer, meaning that it continues to expire at regular intervals until specifically stopped. Valid values are:

vm_tmr_cycle_single
single

vm_tmr_cycle_cyclical
cyclical

interval
(input,INT,8) is a variable for specifying the time interval. The maximum interval allowed is 2^{51}-1.

userword
(input,CHAR,8) is a variable for specifying the user data to be associated with this timer.

Usage Notes

1. The application must use Event Management facilities to be notified of timer expiration. For further information, see Chapter 7, “Timer Services,” on page 59, and Chapter 3, “Event Management,” on page 17.
2. Because the interval for TimerStartMicros is 8 bytes, this function should only be called from assembler or other languages that can manipulate 8-byte binary numbers. For example, in some languages the interval can be represented as an 8-byte character string.
3. Timers set for extremely small intervals (that is, small enough that they could expire before the start processing is complete) could produce unpredictable results.
4. TimerTestMicros and TimerStopMicros should be used to test or stop a particular timer started with TimerStartMicros. TimerTest and TimerStop cannot be used on timers started with TimerStartMicros.
5. If the requested interval would cause the clock comparator to wrap past (or to) zero, (if programming support uses the standard epoch of January 1, 1900, 0:00 A.M. GMT/UTC, this will occur sometime soon after the year 2041), the timer is set for the maximum clock comparator interval and therefore would expire earlier than originally requested. A warning return code and "interval exceeds limit" reason code is issued when this occurs.
6. The maximum 8-byte interval of 2^{51}-1 = X'0007FFFFFFFFFFFF' allows for specification of up to 2,251,799,813,685,247 microseconds. This is approximately 26,062.5 days or 71 years (the exact amount in years depends on the number of leap years included).
7. The timer token is guaranteed not to contain bytes corresponding to the code points for key wildcard characters.
8. *Userword* is part of the event data key for the VMTIMER event. Applications using *userword* as part of the key passed to EventMonitorCreate may incur unexpected results if *userword* is binary data. See “Tips on Constructing Keys” on page 30 for more information.
9. If a userword of all binary zeros is specified, **no** userword will be included in the signal key when Timer Services signals this VMTIMER event.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_tmr_success	vm_tmr_success	TimerStartMicros completed successfully
vm_tmr_warning	vm_tmr_interval_exceeds_limit	Interval requested would cause the clock comparator to exceed its maximum value. The timer was instead set to the maximum clock comparator value.
vm_tmr_error	vm_tmr_type_invalid	Invalid *timertype* specification
vm_tmr_error	vm_tmr_cycle_invalid	Invalid *cycle* specification
vm_tmr_error	vm_tmr_interval_invalid	Invalid *interval* specification; interval must be greater than zero and less than 2^{51}.
vm_tmr_error	vm_tmr_insufficient_storage	Out of storage

Programming Language Bindings

Language	Language Binding File
C	VMCTMR H
Assembler	VMASMTMR MACRO
REXX	VMREXTMR COPY

TimerStartTOD — Start a TOD Timer

TimerStartTOD
    retcode
    reascode
    token
    tod
    zone
    userword

Purpose

Use the TimerStartTOD function to start a real timer. The timer expires at the specified time of day.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

token
(output,INT,4) is a variable where the function returns a timer identifier token. This token is uniquely associated with the timer set by this TimerStartTOD call and is used by many of the other timer functions.

tod
(input,CHAR,6) is a variable for specifying the time, in the form HHMMSS, 24-hour clock notation, when the timer is to expire. Valid times are 000000 to 235959 (000000 being midnight).

zone
(input,INT,4) is a variable for specifying the time zone to use. Valid values are:

vm_tmr_zone_local
local time

vm_tmr_zone_gmt
Greenwich Mean Time

userword
(input,CHAR,8) is a variable for specifying data that the user wishes to associate with this timer.

Usage Notes

1. The application must use Event Management facilities to be notified of timer expiration. For further information, see Chapter 7, “Timer Services,” on page 59, and Chapter 3, “Event Management,” on page 17.
2. If the requested time of day is earlier than the current time of day, the timer will be set for the requested time on the *following* day.
3. When the system operator enters the CP SET TIMEZONE command, running timers set by using TimerStartTOD are adjusted so that the caller-requested expiration time is preserved even though the time zone has changed. If the time zone change results in the caller-specified time being skipped, the affected timers expire immediately. This can happen only on time zone changes that advance the system clock, for example, EST to EDT.

4. Timers started with TimerStartTOD are treated as real single timers, whose intervals are in milliseconds, by other Timer Services functions.
5. TimerTest and TimerStop should be used to test or stop a particular timer started with TimerStartTOD. TimerTestMicros and TimerStopMicros cannot be used on timers started with TimerStartTOD.
6. The timer token is guaranteed not to contain bytes whose values correspond to the code points for key wildcard characters.
7. The *userword* is part of the event data key for the VMTIMER event. Applications using *userword* as part of the key passed to EventMonitorCreate may incur unexpected results if *userword* is binary data. See "Tips on Constructing Keys" on page 30 for more information
8. If a userword of all binary zeros is specified, **no** userword will be included in the signal key when Timer Services signals this VMTIMER event.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_tmr_success	vm_tmr_success	TimerStartTOD completed successfully
vm_tmr_warning	vm_tmr_interval_exceeds_limit	Interval requested would cause the clock comparator to exceed its maximum value. The timer was instead set to the maximum clock comparator value.
vm_tmr_error	vm_tmr_tod_invalid	Invalid *tod* specification
vm_tmr_error	vm_tmr_zone_invalid	Invalid *zone* specification
vm_tmr_error	vm_tmr_insufficient_storage	Out of storage

Programming Language Bindings

Language	Language Binding File
C	VMCTMR H
Assembler	VMASMTMR MACRO
REXX	VMREXTMR COPY

TimerStop — Cancel a Timer

```
TimerStop
        retcode
        reascode
        token
        intervalunits
        interval
        userword
```

Purpose

Use the TimerStop function to cancel a timer previously started by TimerStartInt or TimerStartTOD and return the time remaining for that timer.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

token
(input,INT,4) is a variable for specifying the token that identifies the timer that is to be stopped. This token was returned to the application when it issued the TimerStartTOD or TimerStartInt function to start the timer.

intervalunits
(output,INT,4) is a variable where the function returns a value indicating the units in which the timer interval was specified. The possible values are:

vm_tmr_intunit_micro
microseconds

vm_tmr_intunit_milli
milliseconds

interval
(output,INT,4) is a variable where the function returns the number of units remaining in the timer interval.

userword
(output,CHAR,8) is a variable where the function returns the user data that was established for this timer when it was started.

Usage Notes

1. TimerStop issues an EventSignal for the stopped timer. This lets the application receive notification of the cancelation of this timer through the Event Management facilities. TimerStop signals the VMTIMER event with a key consisting of the timer token concatenated with an S (indicating it was stopped) concatenated with the userword. If a userword of all binary zeros was specified when the timer was started, **no** userword is included in the signal key. See "Timer Services Examples" on page 59 for an example.
2. Timers started with TimerStartTOD are treated as real single timers whose intervals are in milliseconds.

3. If a TimerStop is issued on a timer that has already been stopped or a single timer that has expired, an error return code and *unrecognized token* reason code are returned.
4. TimerStop can be used to stop only timers started with TimerStartInt or TimerStartTOD.
5. At process termination, CMS issues TimerStopAll to stop all outstanding timers.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_tmr_success	vm_tmr_success	TimerStop completed successfully
vm_tmr_error	vm_tmr_invalid_call	Timer has been started by TimerStartMicros and must be stopped by TimerStopMicros
vm_tmr_error	vm_tmr_unrecognized_token	Unrecognized token; timer does not exist or has already expired or been stopped

Programming Language Bindings

Language	Language Binding File
C	VMCTMR H
Assembler	VMASMTMR MACRO
REXX	VMREXTMR COPY

TimerStopAll — Cancel All Timers

TimerStopAll
retcode
reascode

Purpose

Use the TimerStopAll function to cancel all previously started timers.

Parameters

retcode
(output,INT,4) is a variable where the function

reascode
(output,INT,4) is a variable where the function stores the reason code.

Usage Notes

1. TimerStopAll issues an EventSignal for each stopped timer. This lets the application receive notification of the cancelation of each of these timers through the Event Management facilities. TimerStopAll signals the VMTIMER event with a key consisting of the timer token concatenated with an S (indicating it was stopped) concatenated with the userword. If a userword of all binary zeros was specified when the timer was started, **no** userword will be included in the signal key. See "Timer Services Examples" on page 59 for an example.
2. At process termination, CMS issues TimerStopAll to stop all of the process's outstanding timers.
3. In addition to cancelling all outstanding timers set by the CMS application multitasking TimerStart*xxx* calls, TimerStopAll cancels all outstanding timers set by the POSIX services `sleep()` and `alarm()`. The cancellation of a timer set using the POSIX `sleep()` or `alarm()` service causes both a VMTIMER event to be signalled and a POSIX SIGALRM signal to be generated. See *XL C/C++ for z/VM: Runtime Library Reference*, SC09-7624, for details on `sleep()`, `alarm()`, and SIGALRM.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_tmr_success	vm_tmr_success	TimerStopAll completed successfully

Programming Language Bindings

Language	Language Binding File
C	VMCTMR H
Assembler	VMASMTMR MACRO
REXX	VMREXTMR COPY

TimerStopMicros — Cancel a Timer

TimerStopMicros
retcode
reascode
token
interval
userword

Purpose

Use the TimerStopMicros function to cancel a timer previously started with the TimerStartMicros function and return the time remaining for that timer.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

token
(input,INT,4) is a variable for specifying the token that identifies the timer that is to be stopped. This token was returned to the application when it issued the TimerStartMicros function to start the timer.

interval
(output,INT,8) is a variable where the function returns the number of microseconds remaining in the timer interval.

userword
(output,CHAR,8) is a variable where the function returns the user data that was established for this timer when it was started.

Usage Notes

1. TimerStopMicros issues an EventSignal for the stopped timer. This lets the application receive notification of the cancelation of this timer through the Event Management facilities. TimerStopMicros signals the VMTIMER event with a key consisting of the timer token concatenated with an S (indicating it was stopped) concatenated with the userword. If a userword of all binary zeros was specified when the timer was started, **no** userword will be included in the signal key. See “Timer Services Examples” on page 59 for an example.
2. If a TimerStopMicros is issued on a timer that has already been stopped or a single timer that has expired, an error return code and *vm_tmr_unrecognized_token* reason code are returned.
3. Because the interval for TimerStopMicros is 8 bytes, this function should only be called from assembler or other languages that can manipulate 8-byte binary numbers. For example, in some languages the interval can be represented as an 8-byte character string.
4. TimerStopMicros can be used to stop only timers started with TimerStartMicros.
5. At process termination, CMS issues TimerStopAll to stop all outstanding timers.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_tmr_success	vm_tmr_success	TimerStopMicros completed successfully
vm_tmr_error	vm_tmr_invalid_call	Timer has been started by TimerStartInt or TimerStartTOD and must be stopped by TimerStop
vm_tmr_error	vm_tmr_unrecognized_token	Unrecognized token; timer does not exist or has already expired or been stopped

Programming Language Bindings

Language	Language Binding File
C	VMCTMR H
Assembler	VMASMTMR MACRO
REXX	VMREXTMR COPY

TimerTest — Query a Timer

TimerTest
retcode
reascode
token
timertype
cycle
intervalunits
interval
userword

Purpose

Use the TimerTest function to get information about a timer: its type (real or CPU), its cycle (single or cyclical), and the amount of time remaining in the interval (in 4 bytes of milliseconds or microseconds). The timer must have been previously started with the TimerStartInt or TimerStartTOD function.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

token
(input,INT,4) is a variable for specifying the token that identifies the timer that is to be tested. This token was returned to the application when it issued the TimerStartTOD or TimerStartInt function to start the timer.

timertype
(output,INT,4) is a variable where the function returns a value indicating whether the token represents a real timer or a CPU timer. The possible values are:

vm_tmr_timertype_real
real time

vm_tmr_timertype_cpu
CPU time

cycle
(output,INT,4) is a variable where the function returns a value indicating whether the timer is a single timer or a cyclical timer. Their possible values are:

vm_tmr_cycle_single
single

vm_tmr_cycle_cyclical
cyclical

intervalunits
(output,INT,4) is a variable where the function returns a value indicating the units in which the timer interval was specified. The possible values are:

vm_tmr_intunit_micro
microseconds

vm_tmr_intunit_milli
milliseconds

interval
(output,INT,4) is a variable where the function returns the number of units remaining in the timer interval.

userword
(output,CHAR,8) is a variable where the function returns the user data that was established when the timer was started.

Usage Notes

1. Timers started with TimerStartTOD are treated as real single timers whose intervals are in milliseconds.
2. If a TimerTest is issued on a timer that has already been stopped or a single timer that has expired, an error return code and *vm_tmr_unrecognized_token* reason code are returned.
3. TimerTest can be used to test only timers started with TimerStartInt or TimerStartTOD.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_tmr_success	vm_tmr_success	TimerTest completed successfully
vm_tmr_error	vm_tmr_invalid_call	Timer was started by TimerStartMicros and must be tested by TimerTestMicros
vm_tmr_error	vm_tmr_unrecognized_token	Unrecognized token; timer does not exist or has already expired or been stopped

Programming Language Bindings

Language	Language Binding File
C	VMCTMR H
Assembler	VMASMTMR MACRO
REXX	VMREXTMR COPY

TimerTestMicros — Query a Timer

TimerTestMicros
retcode
reascode
token
timertype
cycle
interval
userword

Purpose

Use the TimerTestMicros function to return information about a timer: its type (real or CPU), its cycle (single or cyclical), and the time remaining in the interval (in 8 bytes of microseconds). The timer must have been previously started with the TimerStartMicros function.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

token
(input,INT,4) is a variable for specifying the token that identifies the timer that is to be tested. This token was returned to the application when it issued the TimerStartMicros function to start the timer.

timertype
(output,INT,4) is a variable where the function returns a value indicating whether the token represents a real time timer or a CPU timer. The possible values are:

vm_tmr_timertype_real
real time

vm_tmr_timertype_cpu
CPU time

cycle
(output,INT,4) is a variable where the function returns a value indicating whether the timer is a single timer or a cyclical timer. The possible values are:

vm_tmr_cycle_single
single

vm_tmr_cycle_cyclical
cyclical

interval
(output,INT,8) is a variable where the function returns the number of microseconds remaining in the timer interval.

userword
(output,CHAR,8) is a variable where the function returns the user data that was established when the timer was started.

Usage Notes

1. If a TimerTestMicros is issued on a timer that has already been stopped or a single timer that has expired, an error return code and *vm_tmr_unrecognized_token* reason code are returned.
2. Because the interval for TimerTestMicros is 8 bytes, this function should only be called from assembler or other languages that can manipulate 8-byte binary numbers. For example, in some languages the interval can be represented as an 8-byte character string.
3. TimerTestMicros can be used to test only timers started with TimerStartMicros.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_tmr_success	vm_tmr_success	TimerTestMicros completed successfully
vm_tmr_error	vm_tmr_invalid_call	Timer was started by either TimerStartInt or TimerStartTOD and must be tested by TimerTest
vm_tmr_error	vm_tmr_unrecognized_token	Unrecognized token; timer does not exist or has already expired or been stopped

Programming Language Bindings

Language	Language Binding File
C	VMCTMR H
Assembler	VMASMTMR MACRO
REXX	VMREXTMR COPY

TraceControl — Define and Queries Trace Attributes

TraceControl
retcode
reascode
function
wrapsize
num_types
tracetypes
tracetype_settings

Purpose

Use the TraceControl function to initiate tracing, select specific trace types, or determine the current trace selectivity. This function controls tracing for the entire session.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

function
(input,INT,4) is a variable for specifying the use of the *tracetypes* array and the *tracetype_settings* array. Valid values are:

vm_trc_qitrace
The internal trace types are queried and their setting is returned in the *tracetype_settings* array.

vm_trc_itrace
The *tracetypes* and *tracetype_settings* set the internal traces on and off.

wrapsize
(input/output,INT,4) is a variable that indicates how many trace events are retained if no eligible trace event monitor exists at the time the event is signaled. For a *wrapsize* > 0, when the *wrapsize* is exceeded, the oldest trace event is discarded to make room for the newest arrival. The *wrapsize* variable is either input or output, depending on the function:

- An input variable for the *vm_trc_itrace* function
- An output variable for the *vm_trc_qitrace* function.

In addition to being able to set an integer value greater than 0, other valid values for *wrapsize* are as follows:

vm_trc_wrapnone
No trace events are to be retained

vm_trc_wrapnostor
Trace events continue to be retained until virtual storage is exhausted

vm_trc_wrapnochg
wrapsize is to remain unchanged

For the *vm_trc_qitrace* query function, the current *wrapsize* value is returned.

num_types

(input,INT,4) is a variable for specifying the number of trace types to be set or queried. This value is the number of elements used in the *tracetypes* and *tracetype_settings* arrays. The value must be greater than or equal to 0. If 0 is specified, the *tracetypes* and *tracetype_settings* arrays are ignored, and only *wrapsize* is set or queried.

tracetypes

(input,INT,*num_types*) is an array of 4-byte variables that specifies the trace types to be set or queried. The size of the array is specified by the *num_types* parameter. Valid values for the elements of the array are:

vm_trc_all
All trace types

vm_trc_comm
Communication events

vm_trc_disp
Dispatch events

vm_trc_proc
Process management events

vm_trc_lang
Language Adapter events

vm_trc_sync
Synchronization events

vm_trc_misc
Miscellaneous events

When tracing is set on, this starts the internal CMS tracing for the trace type(s) set.

tracetype_settings

(input/output,INT,*num_types*) is an array of 4-byte variables that indicate the settings of the trace types in the corresponding elements of the *tracetypes* array. The array is either input or output, depending on the function:

- An input array for the *vm_trc_itrace* function
- An output array for the *vm_trc_qitrace* function.

Valid values for each element are:

vm_trc_off
Corresponding trace type is set OFF

vm_trc_on
Corresponding trace type is set ON

vm_trc_unchg
Corresponding trace type is UNCHANGED

Usage Notes

1. TraceControl always results in an event definition of session scope and broadcast signals.
2. The *tracetypes* and *tracetype_settings* arrays are processed in array order. For example, if values corresponding to *vm_trc_all* and *vm_trc_off* are the first elements in the respective arrays, all trace categories are set off before processing subsequent elements in the arrays. As a result, in general, an array element may be nullified by a subsequent array element.
3. The trace type *vm_trc_all* is invalid for the *vm_trc_qitrace* function, because individual elements may have been set after the *vm_trc_all* setting. The *vm_trc_all* setting is returned in the *tracetype_settings* array, but a warning return code indicates that individual trace types should be queried.

4. If an invalid trace type is specified for the *vm_trc_qitrace* function, the corresponding *tracetype_settings* array will return a value of *vm_trc_off* for that trace type, and a warning return code will be given. The rest of the array will be processed.
5. The AccountControl function uses Event Services to collect trace data and process it to produce accounting information.
6. During initialization, the following trace attributes are in effect:

   wrapsize = *vm_trc_wrapnostor*
   num_types = 1
   tracetypes = *vm_trc_all*
   tracetype_settings = *vm_trc_off*

   These values may be overridden when an application invokes TraceControl with its own settings.
7. End-user commands, TRACECTL and QUERY TRACECTL, are provided to set and query trace parameters. See *z/VM: CMS Commands and Utilities Reference*.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_trc_success	vm_trc_success	TraceControl completed successfully
vm_trc_error	vm_trc_bad_func	*Function* is invalid
vm_trc_error	vm_trc_bad_wrap_size	*Wrapsize* value is invalid
vm_trc_error	vm_trc_bad_numtype	*Num_types* is invalid
vm_trc_warning	vm_trc_array_bad_value	*Tracetypes* or *tracetype_settings* array value is invalid

Programming Language Bindings

Language	Language Binding File
C	VMCTRC H
Assembler	VMASMTRC MACRO
REXX	VMREXTRC COPY

TraceSignal — Signal a Trace Event

TraceSignal
retcode
reascode
trace_type_id
subtype_id
data_buffer
data_len
fmtrtn_name

Purpose

Use the TraceSignal function to build a trace header record and signal a VMTRACE event.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

trace_type_id
(input,INT,4) is a variable for specifying the trace type that is to be signaled. This value is stored in the trace header as part of the event key when the VMTRACE event is signaled. System-defined trace types supported by TraceControl may be used, as well as any user-defined trace type. Trace types in the range of 0–31 are reserved for CMS. Only trace types defined by CMS may be set or queried using TraceControl services.

subtype_id
(input,INT,4) is a variable for specifying the trace subtype that is to be signaled. This value is stored in the trace header as part of the event key when the VMTRACE event is signaled.

data_buffer
(input,CHAR,*data_len*) is a variable for a buffer that holds the trace data to be signaled. This data is appended to the trace header.

data_len
(input,INT,4) is a variable for specifying the length of the trace data contained in *data_buffer*. This value must be greater than or equal to 0. If 0 is specified, the event data being signaled contains only a trace header.

fmtrtn_name
(input,CHAR,8) is a variable for specifying the name of the format routine you wish to use when processing trace records. If the format routine name is left blank, the default system routine, DVF, is filled in.

Usage Notes

1. The TraceSignal interface builds a header record in a format that matches the format used by CMS. The mapping of the header is defined in the *vm_trc_recfmt* structure in the language binding files.
2. When a VMTRACE event is signaled using the TraceSignal interface, the entire trace record is defined as the event key. Event monitors may be created specifying the header and data as the key, or any portion of it using wildcarding.

3. If tracing is on for a CMS trace type and that trace type is signalled by the application, qualifying event monitors may have CMS trace data as well as the data from this TraceSignal for retrieval. In general, the application should avoid using trace types reserved for CMS.
4. Trace subtypes associated with CMS trace types are reserved for definition by CMS and are defined in Appendix B, "CMS Trace Record Formats," on page 303. Users should not add their own subtypes for any of the CMS-defined types. If a user wants to trace events not listed as a CMS type or subtype, then a new user type category can be defined, (with a type ID not in the range of 0 - 31), and monitored using event services. TraceSignal may then be used to issue an event signal for that event.
5. A TraceSignal results in an event signal being issued for the specified *trace_type_id* regardless of whether tracing was enabled for a CMS trace type by TraceControl.
6. If *data_len* plus the length of the trace header is greater than X'7FFFFFFF', then an overflow condition occurs and an error return code is given with a reason code of vm_trc_bad_datalen.
7. If a TraceSignal is issued for a CMS trace type for *vm_trc_comm* or *vm_trc_disp* events, accounting records will be affected if accounting is on for *vm_act_comm* or *vm_act_cpu*.
8. See "Tips on Constructing Keys" on page 30 for a discussion of the use of binary data in event data keys.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_trc_success	vm_trc_success	TraceSignal completed successfully
vm_trc_error	vm_trc_bad_datalen	Data length specified is invalid
vm_trc_error	vm_trc_insufficient_storage	No more storage available

Programming Language Bindings

Language	Language Binding File
C	VMCTRC H
Assembler	VMASMTRC MACRO
REXX	VMREXTRC COPY

VCPUCreate — Create a Virtual Processor (Virtual CPU)

VCPUCreate
retcode
reascode
number_to_create
number_available

Purpose

Use the VCPUCreate function to add a virtual processor to the virtual machine and make it available to CMS for executing threads.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

number_to_create
(input,INT,4) is a variable for specifying the number of processors to create. The number must be between 0 and 63, inclusive. If zero is specified, no processors are created, but the *number_available* parameter is updated.

number_available
(output,INT,4) is a variable where the function returns the number of processors, at completion of this operation, that exist in the virtual machine and are usable. This number includes the base CPU but excludes any processors that were in stopped-state when the tasking application started.

Usage Notes

1. CMS dispatches threads on the new virtual processor. It cannot be manipulated by the application by means other than the process and thread functions.
2. The use of multiple virtual processors is supported only in XA- or XC-mode virtual machines.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_cpu_success	vm_cpu_success	VCPUCreate completed successfully
vm_cpu_error	vm_cpu_not_xa	Virtual machine was not XA or XC; CPUs were not created
vm_cpu_error	vm_cpu_no_more_vcpus	Virtual CPU limit exceeded; CPUs were not created
vm_cpu_error	vm_cpu_bad_number_to_create	The *number_to_create* parameter was less than zero or greater than 63

Programming Language Bindings

Language	Language Binding File
C	VMCCPU H
Assembler	VMASMCPU MACRO
REXX	VMREXCPU COPY

VCPUSelect — Request Special Virtual CPU Dispatching

VCPUSelect
retcode
reascode
selection

Purpose

Use the VCPUSelect function to request special virtual CPU dispatching consideration for the calling thread.

Parameters

retcode
(output,INT,4) is a variable where the function stores the return code.

reascode
(output,INT,4) is a variable where the function stores the reason code.

selection
(input,INT,4) is a variable for specifying the virtual CPU dispatching requested for the calling thread. Valid values are:

vm_cpu_base_only
Specifies that this thread must be dispatched only on the base virtual CPU. Upon return from this function the thread will be executing on the base virtual CPU.

vm_cpu_any
Specifies that this thread can be dispatched on any virtual CPU.

Return Codes and Reason Codes

Return Code	Reason Code	Meaning
vm_cpu_success	vm_cpu_success	VCPUSelect completed successfully
vm_cpu_error	vm_cpu_bad_selection	Invalid *selection*

Programming Language Bindings

Language	Language Binding File
C	VMCCPU H
Assembler	VMASMCPU MACRO
REXX	VMREXCPU COPY

Chapter 14. System Exits

To let installations tailor or extend the multitasking environment a set of system exits (or installation-wide exits) is provided that lets installation-provided routines execute at key points in CMS processing. Exits are implemented through installation-replaceable modules.

The set of exits is divided into two classes. The first class, *general-purpose exits*, allows customer-written code to get control during key points, such as session start-up and thread creation. The second class, *programming language environment exits*, is specifically designed to allow the installation to support compiled language run-time environments other than C/C++ for z/VM, C for VM/ESA, and Basic Assembler Language. These language exits are called to transfer control from CMS kernel code to the tasking application. Both classes of exit use OS Type 1 linkage and must be written not to rely on the existence of any extended context, such as a language run-time environment. Certain additional linkage constraints may apply in some cases.

System Exit Linkage Conventions

The linkage from CMS to system exits is roughly the same for all exit entry points, the difference being in the parameter lists passed to the various entry points. Fundamentally, the linkage is this:

R1
Address of routine-specific parameter list

R13
Address of a 72-byte save area

R14
Return address

R15
Entry point address.

In addition, interrupts are disabled, the PSW key is set to SYSTEM key, and if the virtual machine is XC mode, then the address space mode is set to primary. The exit should be AMODE 31 capable.

The exit is expected to preserve the general purpose registers and the additional conditions cited.

The following sections describe each entry point's parameter list and also describe deviations from this standard, if any.

General-Purpose Exits

These exits are a set of entry points contained in an installation provided module called DMSSXM MODULE that resides on the S-disk. The individual system exits are entry points within this module. Typically, these entry points are in separate text files. They are combined with a CMS-provided interface routine, through a procedure described below, to form the system exits module.

Session Initialization Exit

Entry point USRSINIT is called early in session initialization. The multitasking API is not yet available for use, and tasking has not yet been turned on. This exit lets the installation perform session-wide tailoring. This exit should not be used for application-specific processing.

Parameters for session initialization exits are as follows:

Word 0
is the address of a signed 4-byte return code that USRSINIT must set to indicate whether the exit processing was successful. A value of 0 indicates success. Any other value indicates failure, which causes CMS to stop initialization with an abnormal termination code equal to the return code generated by USRSINIT.

© Copyright IBM Corp. 1992, 2023

Thread Initialization Exit

Entry point USRTINIT is called during thread creation after the thread has been completely built. The exit runs on the new thread. Interrupts are disabled and the address space mode is primary.

Parameters for thread initialization exits are as follows:

Word 0
is the address of a signed 4-byte return code that USRTINIT must set to indicate whether the exit processing was successful. A value of 0 indicates success. Any other value indicates failure, which causes CMS to end with an abnormal termination code equal to the return code generated by USRTINIT.

Word 1
is a signed 4-byte process ID of the process owning the new thread.

Word 2
is a signed 4-byte thread ID of the new thread.

The exit can determine whether this is the first thread in the process by examining the thread ID of the new thread (the ID will be 1 if the thread is the first one).

Thread Termination Exit

Entry point USRTTERM is called at least once during thread termination processing. The first call is made just before CMS begins to do cleanup for the terminating thread. If the terminating thread is the last thread in the process, CMS will call USRTTERM again just before the beginning of process cleanup. When this second call is made, the exit may assume that thread cleanup has already been done for all threads in the process, including the thread on which the process cleanup call is executing. In both cases, the exit runs on the terminating thread. Interrupts are disabled and the address space mode is primary.

Parameters for thread termination exits are as follows:

Word 0
is address of a signed 4-byte return code that USRTTERM must set to indicate whether the exit processing was successful. A value of 0 indicates success. Any other value indicates failure, which causes CMS to end with an abnormal termination code of F07. by USRTTERM.

Word 1
is the signed 4-byte process ID of the process owning the terminating thread.

Word 2
is the signed 4-byte thread ID of the terminating thread.

Word 3
is a signed 4-byte flag, which is 1 if this is the process cleanup call, or 0 otherwise.

Root Process Exit

Some CMS services are provided by threads running in a system process known as the *root process*. If the installation needs to add function to CMS, it can use this exit to run one or more threads in the root process.

Entry point USRRTHD is the entry point given control on the new thread of the root process. It has the following attributes:

- Is created in a dispatching class of its own
- Is passed no parameters at thread creation
- Has priority equal to other threads of the root process.

This thread is created after all other CMS initialization is complete but before the commands process is created. Threads running in the root process are not provided with a high-level language context — this includes the USRRTHD thread.

Building a System Exits Module

Customer-supplied system exits are packaged separately from the CMS nucleus in a module file named DMSSXM.

IBM does not ship a DMSSXM MODULE containing null exits. CMS simply does not attempt to drive any exits if DMSSXM is not present. This corresponds to the approach used for SFS exits (for example, DMSJNE). The installation-supplied DMSSXM must be placed on the CMS system disk.

The build procedure for the system exits module is:

1. Produce, through compilation or assembly steps, the text decks for the USR*xxx* entry points.
2. Use LOAD and INCLUDE commands with the RLDSAVE option to load or include the USR*xxx* text files into storage.
3. Issue `INCLUDE DMSSXM ( RESET DMSSXMIN RLDSAVE` to bring the IBM-supplied system exits initialization code into storage.
4. Issue `GENMOD DMSSXM ( MAP` to write the system exits module to disk.

The system programmer may choose to put the system exits module in the installation saved segment, CMSINST. This segment is the only one loaded early enough to contain the system exits module. This allows the system programmer's site to share a single copy of his module. If the module is placed into a segment, the system programmer should specify that the module should be processed as a nucleus extension at segment load time.

Programming Language Environment Exits

CMS provides language run-time environment support for C and assembler. In addition, CMS allows customers to replace the supplied language environment support routines with their own. This is useful for supporting tasking applications written in a language other than the ones supported by IBM. To support some other language run-time environment, the customer should provide replacements for all of the language support exits. Those replacements should then be bound into a language environment manager module, the procedure for which will be described after first presenting descriptions of the exits.

If you plan to write your own run-time support, you will need to provide object code for several entry points. Together these routines manage the creation, deletion, and context switching operations associated with your run-time environment. In this section, this collection of routines is called the *environment manager*.

CMS does not invoke the environment manager for threads that run in the root process. All root process threads are written to run without a high-level language context. This includes any event traps that might be sprung in the root or any abnormal termination retry routines that might be executed in the root. This point becomes relevant for the customer only if the USRRTHD exit is exploited.

Table 22 on page 279 tells the names and functions of the run-time support routines. The following sections describe the functions and linkage for each routine. Each of these routines must be capable of executing in AMODE 31.

Table 22. Run-Time Support Entry Points and Functions

Entry Point Name	Function
RTEPCR	Process creation initialization
RTEPDE	Process deletion cleanup
RTETCR	Thread creation initialization
RTETDE	Thread deletion cleanup
RTERR	Run application code in proper context
does not apply	Context switches

Process Creation

When CMS creates a new process, it calls entry point RTEPCR to give the environment manager an opportunity to respond to the new process. RTEPCR does not run in the context of the new process. CMS calls RTEPCR before any threads are created in the new process. CMS calls RTEPCR with this parameter list:

Word 0
Address of a 4-word array containing the R0, R1, R2, and R13 values that will be passed to APPLMAIN (see "Writing Multitasking Applications in Assembler" on page 85).

Word 1
Address of a process-specific anchor word reserved for use by the environment manager in managing this process's environment

Word 2
Context switch handler address (to be filled in by the environment manager)

Word 3
Address of a list of additional entry points for functions provided to help environment managers implement higher-level thread functions. The list has the following contents:

Word 0
The number of entries following in the list (not counting this word)

Word 1
Entry point address of the Thread Block function

Word 2
Entry point address of the Thread Unblock function.

These functions use register-only linkage (BALR R14,R15) and parameter passing and do not require an in-storage parameter list. These language environment manager functions have the following specifications:

Thread Block
Increment the block count of the calling thread. This value is equal to the number of times Thread Block was called for this thread minus the number of times Thread Unblock was called for this thread. If the count is greater than zero, the thread is blocked (made non-dispatchable). Control returns after the thread is unblocked by the Thread Unblock function.

Register Usage

R14
Return address

R15
Input is the address of Thread Block. Output is the return code.

Return Code

0
Successful completion

Thread Unblock
Decrement the block count of the specified thread. If the count is less than or equal to zero, the thread is unblocked (made dispatchable).

Register Usage

R1
Thread ID of the thread that is to be unblocked

R2
Process ID of the process that owns the thread to be unblocked

R3
Call Flag - value of 1 indicates the call is from an interrupt handler routine; value of 0 indicates the call is from noninterrupt handler code.

R14
Return address

R15
Input is the address of Thread Unblock. Output is the return code.

Return Codes

0
Successful completion

8
The specified thread does not exist

The language environment manager is trusted not to unblock threads in processes it does not manage. The process ID parameter is provided because the function may need to be called from an interrupt handler. The flag value allows CMS to be sure if the caller is in an interrupt handler, since some servers interfere with interrupt processing. RTEPCR is expected to perform any process-level environment setup operations deemed appropriate and return to CMS, filling in the address of the context switch handler for this process. If there is no context switch handler for this process, RTEPCR should put a zero into this field. RTEPCR must be MP-capable.

Process Deletion

When CMS deletes a process, it calls entry point RTEPDE to give the environment manager an opportunity to respond to the process deletion. By the time RTEPDE is called, all threads in the process have been deleted. RTEPDE does not run in the context of the deleted process. CMS calls RTEPDE with this parameter list:

Word 0
Reserved for IBM

Word 1
Address of the environment manager anchor word for this process.

RTEPDE is expected to perform process-level cleanup operations and return to CMS. RTEPDE must be MP-capable.

Thread Creation

When CMS creates a thread, it calls entry point RTETCR to give the environment manager an opportunity to respond to the new thread. RTETCR runs on the new thread. CMS calls RTETCR with this parameter list:

Word 0
Reserved for IBM

Word 1
Address of the environment manager anchor word for the owning process

Word 2
Address of a thread-specific anchor word reserved for use by the environment manager in managing this thread's environment.

RTETCR is expected to perform thread-related setup operations and return to CMS. RTETCR should not attempt to call the multitasking API, as the complete API is not yet available for the new thread.

Because RTETCR runs on the new thread, and because the thread must be dispatched before it can run to call RTETCR, programmers will observe that for a given thread, the owning process' context switch exit is driven before RTETCR. The first time the context switch exit is driven for a given thread, the environment manager's thread-specific anchor word will be zero. This gives the context switch exit a way to know whether RTETCR has been driven yet for the thread. RTETCR must be MP-capable.

Thread Deletion

When it deletes a thread, CMS calls RTETDE to give the environment manager an opportunity to respond to the deletion. CMS calls RTETDE with this parameter list:

Word 0
Reserved for IBM

Word 1
Address of the environment manager anchor word for the owning process

Word 2
Address of the environment manager anchor word for the deleted thread.

RTETDE is expected to tear down the run-time environments for any uncompleted calls to RTERR. It is also expected to perform any other thread-related cleanup operations.

Depending on the priorities and behaviors of the threads involved, RTETDE may be invoked for a given thread even though RTETCR was never invoked. This happens when a thread is created but is deleted before ever being dispatched. RTETDE should be written to handle such occurrences.

RTETDE should not attempt to call the multitasking API, as portions of the process environment have already been dismantled for the terminating thread. RTETDE must be MP-capable.

Run a Routine in Context

To run a user routine, such as a new thread or an event trap, CMS calls RTERR, passing information describing both the routine to be executed and the process and thread in which the routine will run. RTERR runs on the thread on which the routine will run.

RTERR might be called more than once on a given thread. If this happens, the invocations will be nested in LIFO fashion; that is, a given routine whose execution is in progress through a call to RTERR will not have completed if CMS calls RTERR to run a subsequent routine on the same thread. When this happens, the environment manager may assume that these routines complete in the reverse order from which they were started.

CMS calls RTERR with this parameter list:

Word 0
Reserved for IBM

Word 1
Address of the environment manager anchor word for the owning process

Word 2
Address of the environment manager anchor word for the thread on which the routine will run

Word 3
Routine's entry point address

Word 4
Address of parameter list to be passed to the routine

Word 5
Length of parameter list (in words) to be passed to the routine.

In addition, the AS mode is primary.

When RTERR is driven to run the first thread of the application (APPLMAIN), word 4 of the RTERR parameter list points to a parameter list for APPLMAIN organized as follows:

Word 0
A pointer to a copy of the extended parameter list CMS passed to the tasking module, or zero if no extended parameter list was provided.

Word 1
A pointer to a copy of the tokenized parameter list CMS passed to the tasking module.

Word 2
A copy of the R2 value CMS passed to the tasking module. If the tasking module is loaded as a nucleus extension, then this pointer points to the nucleus extension's SCBLOCK.

Word 3
A pointer to a copy of the USERSAVE area CMS provided for the module. This copy is provided so that the program can interrogate the USERINFO doubleword in USERSAVE.

RTERR is expected to establish the run-time environment for the routine and then give control to the routine in the manner it expects. Because RTERR is entered disabled for interrupts, it should enable before passing control to the thread so the thread can begin execution enabled for interrupts.

RTERR needs to maintain a data structure that allows the context switch exit to determine which environment is active on a given thread. CMS guarantees that the environment active on a thread is the environment associated with the last uncompleted call to RTERR. This suggests that RTERR should maintain a stack of environment descriptions.

Depending on the routine's behavior, it may terminate by returning to RTERR. If it does so, then RTERR is expected to tear down the run-time environment it created and return to CMS with the routine's return code in R15. If the routine ends in some other fashion (for example, call to ThreadDelete), then control never returns to RTERR and RTETDE is responsible for cleanup.

Note that RTERR must be reentrant. Note also that RTERR should make provisions for RTETDE to deallocate any storage RTERR may have allocated, including save areas, in the event that RTERR never regains control.

CMS does not use RTERR to drive named trap routines. Named trap routines must be written in assembler and must not rely on the existence of any high-level-language context. RTERR must be MP-capable.

Context Switching

Depending on the semantics and execution model being implemented by the environment manager, it may be necessary for the manager to get control as CMS switches among threads. CMS provides for this by allowing RTEPCR to specify the address of an exit that should be driven when context switching operations occur. To provide for additional flexibility, CMS allows the context switch exit to be different for each process.

Context switching is a four-part operation as follows:

1. The execution state of the currently-executing thread is saved. The currently-executing thread is also called the *old* thread.
2. A thread is selected for execution. The chosen thread is called the *new* thread.
3. The execution state of the new thread is restored.
4. Control is passed to the new thread.

These steps are always the same, regardless of the processes and dispatch classes to which the old and new threads belong.

After CMS saves the execution state of the old thread, it calls the environment manager's context switch exit associated with the process owning the old thread. In response to this call, the environment manager must save any environment information (*extended context*, if you will) associated with the old thread. The context switch exit may assume that the environment active on the old thread is the one associated with the last uncompleted call to RTERR. The parameter list passed to the exit is as follows:

Word 0
X'0000' (indicates *save* operation)

Word 1
Address of environment manager anchor word for the process owning the old thread

Word 2
Address of environment manager anchor word for the old thread.

Before CMS restores the execution state of the new thread, it calls the environment manager's context switch exit associated with the process owning the new thread. In response to this call, the environment manager must restore any environment information (again, *extended context*) associated with the new thread. Again, the context switch exit may assume that the environment active on the thread is the one associated with the last uncompleted call to RTERR. The parameter list passed to the exit is as follows:

Word 0
X'0001' (indicates *restore* operation)

Word 1
Address of environment manager anchor word for the process owning the new thread

Word 2
Address of environment manager anchor word for the new thread.

For a given thread, the context switch exit (RESTORE function) is the first exit to be driven. In particular, it is driven before RTETCR. The first time the context switch exit is driven for a given thread, the environment manager's thread-specific anchor word will be zero. This gives the context switch exit a way to know whether RTETCR has been driven yet for the thread.

Together these two exits allow the environment manager to perform environment-switching operations in concert with CMS's execution switching operations.

The context switch exit must not call the multitasking API and it must be MP-capable.

Building a Language Environment Manager

CMS packages its language managers as part of the system. A customer may choose to build his own language environment manager; in this case, he will have to build a module containing his language environment manager and some support code supplied by IBM.

The build procedure for the language environment manager is:

1. Produce, through compilation or assembly steps, the text decks for the RTExxx entry points.
2. Use the LOAD and INCLUDE commands with the RLDSAVE option to load or include the RTExxx text files into storage.
3. Issue `INCLUDE DMSEMT ( RESET DMSEMTIN RLDSAVE` to bring the IBM-supplied language environment manager mainline into storage.
4. Issue GENMOD to write the language environment manager to disk. The name of this module must correspond to the language environment manager name specified in the corresponding language environment selector text file, described below.

The system programmer may choose to put his language environment manager in a logical segment. This will allow the system programmer's site to share a single copy of his module. If he chooses to put the module into a segment, he should specify that the module should be processed as a nucleus extension at segment load time. The system programmer may put the module into IBM's VMMTLIB segment if he wants.

CMS loads the module automatically when it is needed and establishes it as a nucleus extension with the SYSTEM and SERVICE options. If the user NUCXLOADs the module himself, he must also use these options.

In addition to the language environment exits the environment manager must also provide the language environment selector text deck. The format of this text deck is an external because a customer writing his own language environment manager will need to supply one for interrogation by CMS process initiation.

The selector text deck declares a single data area, called the *language environment selector block*, that defines the language environment manager to be used and the address of the application entry point within the application module. The rest of the data area is reserved for use by IBM. The data area is located at externally-visible label DMSLESB. The following code fragment shows how the data area is set up:

```
         EXTRN entrypnt        It is elsewhere
DMSLESB  CSECT                 Selector data area
```

```
            DC    CL8'langmod'    Lang env module name (8 bytes)
            DC    A(entrypnt)     Address of appl entry pt (4 bytes)
            DC    CL20' '         Reserved (20 bytes)
            END
```

So as to ease the placement of this module into a segment, IBM code will not write in this data area.

It is not strictly necessary for the application entry point to bear label APPLMAIN. A customer-supplied language environment selector may indicate any address as the entry point of the application. The IBM-supplied language environment selectors all assume APPLMAIN is the application entry point, though.

Chapter 15. Suggestions for Server Writers

Problems common to most servers include how to communicate with clients, how to handle interrupts, and how to interact with CMS in an efficient manner. The following suggestions are given in light of the CMS execution environment to provide help in finding the best solutions to these problems and others that might occur.

Interrupt Handling

Most CMS-based servers are interrupt driven. They handle interrupts for communications, for I/O and for timer handling. This is accomplished through exit routines established with the CMS HNDEXT, HNDINT, HNDIO, HNDIUCV, and CMSIUCV macros.

CMS provides mechanisms to allow threads to perform their functions in a synchronous manner, with the concurrency among threads providing the asynchrony. So, to combine CMS interrupt handling with CMS multitasking, the exit routine should convert the interrupt into either a message or a signal. The exit routine, which continues to execute in the restricted environment of CMS second-level interrupt handlers, should gather the data from the interrupt, build either a queue message or event data for a signal and issue the QueueSend or EventSignal function. A thread can then process the interrupt in an environment without the CMS interrupt handler restrictions. The server should spend as little time as possible in the interrupt handler exit routine.

Application interrupt handlers run as extensions to CMS interrupt handling. As is the case with other CMS services, some multitasking services should not be used in such an interrupt exit. Operations on events and synchronization objects must be limited to those of session scope. Messages can be sent to session-level queues. The interrupt exit should not cause the thread that it interrupted to become blocked. It must not issue any of the following calls:

- MutexAcquire
- CondVarWait
- EventTrap
- EventWait
- QueueReceiveBlock
- SemWait
- ThreadCreate
- ThreadSuspend (self)
- ThreadDelay
- ThreadYield
- AbnormalEnd
- VCPUCreate
- VCPUSelect.

An interrupt exit can use QueueSend to transmit a message to a session-level queue. The queue must be identified as a service queue and the interrupt exit must use the queue's service ID as the handle in the call to QueueSend. The message will appear to have come from the interrupted process.

Because the interrupt handler may run under any process, it should not depend upon process-level resources.

Communication

Most service virtual machines communicate with clients by using the Inter-User Communications Vehicle (IUCV), or Advanced Program-to-Program Communication/VM (APPC/VM). The server uses each of these

© Copyright IBM Corp. 1992, 2023

by means of HNDEXT or HNDIUCV exits. In other words, the server handles interrupts to receive messages and learn of the completion of communication functions.

If the server needs to use one of these communication methods, the technique to apply is to turn the interrupt into a signal or a queue message, as described above. In this way, the client can continue to communicate with the server using the communication mechanisms it currently uses.

If the client is another CMS virtual machine in a VM collection or connected through an SNA network, both the client and the server can communicate with queues. This method best integrates communication into the natural concurrent programming structure of the server. But, the client must be on a VM system.

A variation of the queue approach is to use the QueueIdentifyCarrier function to provide your own carrier for queue communication. The server can use the queue API, with the installed queue carrier sending and receiving data according to whatever communication protocol the client uses. This allows the server to exploit the concurrent programming power of queues and still allow the client to use the communication mechanisms available on its system. However, to accomplish this, the server writer must provide the queue carrier, and some component on the client system must understand the line flows.

Another alternative is to use the SAA Common Programming Interface Communications. This high-level language API for APPC communications is made up of thread synchronous calls. It also provides a set of communications events that can be handled with Event Management Services.

For more information, see the *Common Programming Interface Communications Reference (https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf)*.

Data Management

The main consideration when managing data in the server is how various types of file operations affect the synchrony of the virtual machine. The read and write operations on CMS minidisks are synchronous and as such do not allow for overlapping file I/O with other processing. Many other services are synchronous as well. For servers with intense file I/O requirements, asynchronous interfaces should be used.

The CMS Shared File System (SFS) provides asynchronous interfaces for file I/O. By using the asynchronous interfaces and distributing work between several CMS work units, both overlap and parallelism can be achieved. For information on using asynchronous requests for CMS multitasking applications, see the *z/VM: CMS Application Development Guide*.

For more information on the CMS Shared File System, see the *z/VM: CMS File Pool Planning, Administration, and Operation*.

Another alternative is to use the CP DASD Block I/O System Service. This service provides block-oriented I/O directly to a minidisk in an asynchronous manner. This approach provides the best performance but requires the most work by the server writer.

Minidisk data can also be mapped into a data space through the mapping services of the CP MAPMDISK macro. The data can then be manipulated directly with CP committing changes to DASD.

For more information on the DASD Block I/O System Service and the MAPMDISK macro, see *z/VM: CP Programming Services*.

General Guidelines

Here are some general guidelines to follow:

- Do not obtain a mutex within a CMS interrupt exit. Because no threads can be dispatched on the CPU until the exit routine returns to CMS, this is a great opportunity for deadlock.
- Servers designed to execute in multiprocessor virtual machines should not rely on the validity of fields in page zero. An attempt to reference a NUCON field from a CPU other than the base CPU results in a reference to the prefixed page zero of that CPU. The field at the referenced address may not be valid.

- Only the CMS branch entries accessed through the CMS macro API can be reliably used on multiple CPU's. Branching directly to a CMS service from a non-base CPU opens non-kernel CMS functions to multiprocessor effects, which they cannot handle.

Chapter 16. Using CMS Multitasking with OpenExtensions Services

Many OpenExtensions POSIX services are only alternate interfaces to CMS Multitasking and related services. The POSIX concepts are defined in terms of CMS process model concepts. These definitions are:

POSIX Process
A CMS process with POSIX-defined resources maintained local to the process.

POSIX Thread
A CMS thread with additional run-time library support.

POSIX Alarm
A CMS timer which causes the generation of a POSIX signal.

POSIX Signal
An interrupt-like message carried between processes by CMS IPC. The generation and delivery of POSIX signals also result in the signalling of CMS events, as described below.

In general, CMS services and POSIX services can be used without restriction within an application. Further, OpenExtensions and non-OpenExtensions applications can invoke each other and can coexist in processes within a CMS session. Additional specific rules and definitions follow:

- All CMS services are signal safe.
- In an OpenExtensions application, event trap routines are not run when the corresponding monitor is deleted due to process termination. For explicit monitor deletion, the traps are run as usual.
- OpenExtensions services must not be issued from interrupt handlers.
- Thread entry points in OpenExtensions C applications are regular C functions; that is, they do not require `#pragma linkage(`*`entryname`*`,OS)`. Threads can be created at these entry points by using ThreadCreate, EventTrap, or pthread_create.
- OpenExtensions C applications' initial threads start at routine `main()` (they do not use the `applmain()` convention) with the usual *argv* and *argc* entry conditions.
- The `spawn()` and `spawnp()` functions behave differently when executed by a REXX EXEC from the Commands process. In this case, these two functions are synchronous, as if a `waitpid()` had been issued after the `spawn()` or `spawnp()`.

CMS Events For OpenExtensions Signals

To promote integration of the POSIX programming environment into CMS application multitasking, CMS defines a system event representing the generation and delivery of a POSIX signal. This allows ordinary CMS programs to be cognizant of POSIX signal activity and, more generally, permits any CMS program to use CMS event management to wait for combinations of events including POSIX signals, other CMS system events, and user events.

The POSIX signal event has the name VMPOSGNL and is defined with the following attributes:

- Process scope - VMPOSGNL may be monitored only in the process for which the POSIX signal has been generated.
- Broadcast signals - all eligible VMPOSGNL event monitors are notified essentially simultaneously.
- Asynchronous signals - execution of the thread that causes a VMPOSGNL event to be signaled is not affected by the handling of that signal.
- Loose signal limit 0 - VMPOSGNL event signals are not retained by CMS if no eligible event monitors are defined.

The event data is 20 bytes in length and is mapped by the following structure:

© Copyright IBM Corp. 1992, 2023

Table 23. Event Data

Hex	Dec	Type	Len	Name	Description
00	0	Signed	4	vm_posgnl_signum	POSIX signal number
04	4	Signed	4	vm_posgnl_cmstid	Target CMS thread ID
08	8	Signed	4	vm_posgnl_sigact	POSIX signal action code
0C	12	Opaque	8	vm_posgnl_thid	Target POSIX thread ID

The vm_posgnl_signum field may assume any of the following values:

Value Code	Meaning
vm_posgnl_sigabnd	Abend
vm_posgnl_sigabrt	Abnormal termination signal
vm_posgnl_sigalrm	Timeout signal
vm_posgnl_sigchld	Child process terminated or stopped
vm_posgnl_sigcncl	Internal cancel signal generated by the `pthread_cancel()` function (cancellation cleanup handlers are run)
vm_posgnl_sigcont	Continue if stopped
vm_posgnl_sigdce	Reserved for exclusive use of DCE**
vm_posgnl_sigfpe	Erroneous arithmetic operation
vm_posgnl_sighup	Hangup detected on controlling terminal or death of controlling process
vm_posgnl_sigill	Detection of an incorrect hardware instruction
vm_posgnl_sigint	Interactive attention signal
vm_posgnl_sigio	Completion of input or output
vm_posgnl_sigioer	Input or output error
vm_posgnl_sigkill	Termination signal (cannot be caught or ignored)
vm_posgnl_sigpipe	Write on pipe with no readers
vm_posgnl_sigqsce	Internal cancel signal generated by a quiesce_threads operation (cancellation cleanup handlers are *not* run)
vm_posgnl_sigquit	Interactive termination signal
vm_posgnl_sigsegv	Detection of an invalid memory reference
vm_posgnl_sigstop	Stop signal (cannot be caught or ignored)
vm_posgnl_sigterm	Termination signal
vm_posgnl_sigtstp	Interactive stop signal
vm_posgnl_sigttin	Read from controlling terminal attempted by a member of a background process group
vm_posgnl_sigttou	Write to controlling terminal attempted by a member of a background process group
vm_posgnl_sigusr1	Reserved as application-defined signal 1
vm_posgnl_sigusr2	Reserved as application-defined signal 2

The possible values of the vm_posgnl_sigact field are as follows:

Value Code	Meaning
vm_posgnl_generate	Specified POSIX signal is being generated
vm_posgnl_ignore	The action for the specified POSIX signal is ignore, and the signal is not being delivered
vm_posgnl_continue	The default action of continue is being taken for the specified POSIX signal
vm_posgnl_stop	The default action of stop is being taken for the specified POSIX signal
vm_posgnl_catch	The action for the specified POSIX signal is catch, and the signal is being delivered to the signal catcher
vm_posgnl_intercept	A default action applies to the signal but the run-time library signal interface routine (SIR) has requested an intercept; the signal is delivered to the SIR which determines what action to take
vm_posgnl_terminate	The default action of terminate is being taken for the specified POSIX signal
vm_posgnl_sigwaiter	The specified POSIX signal is being delivered by satisfying an outstanding `sigwait()`

The event key consists of the first three fields in the above structure: the POSIX signal number, the CMS thread ID, and the action code. None of these fields will ever receive a value containing a match key wildcard character. Thus, match keys for the VMPOSGNL event can be constructed in a straightforward manner, without concern for the possible presence of wildcard characters in the data.

At generation time the kernel attempts to select a target thread based on its signal interest model. When a default action of stop or continue is being applied, all threads are targeted; otherwise, a single target thread is chosen. (A default action of terminate applies to all threads too, but the signal is generated for one thread only. At delivery time, if the run-time library has not requested interception, the kernel simply terminates the process.) If no thread can be selected as the target because none has issued a `sigwait()` for the signal or has it unblocked, the signal is generated for the process (and does not become pending for any thread until some thread indicates interest in it with `sigprocmask()`, `sigpending()`, `sigsuspend()`, or `sigwait()`). In this case, both the CMS thread ID and the POSIX thread ID in the VMPOSGNL event data will be set to binary zero (which is not valid as a thread ID of either flavor).

VMPOSGNL events are always signaled in the target process. VMPOSGNL events with the action code vm_posgnl_generate are signaled immediately after the POSIX signal has been generated; those with the various delivery action codes occur just before the delivery action is effected or attempted. If the run-time library "puts back" a signal that has been delivered to it (using the queue_interrupt (BPX1SPB) callable service), the subsequent redelivery does not cause another VMPOSGNL event.

In general, therefore, there are exactly two VMPOSGNL events for each POSIX signal for which delivery is attempted. In some cases, however, signal interaction rules cause the kernel to discard pending signals before delivery when other signals are generated (that is, generation of SIGKILL or SIGCONT causes pending stop signals to be discarded, and generation of SIGSTOP or SIGTSTP causes pending continue signals to be discarded). No VMPOSGNL delivery event occurs for signals discarded in this manner.

Appendix A. Return and Reason Code Values

This section discusses return and reason codes for process management, synchronization, event services, trace services, accounting services, interprocess communication, timer services, VCPU services and CMS monitor data.

For Process Management

Symbolic name	Value
vm_pro_success	0
vm_pro_warning	4
vm_pro_error	8
vm_pro_bad_name_len	2
vm_pro_out_of_storage	4
vm_pro_no_ids	7
vm_pro_bad_priority	8
vm_pro_bad_flags	9
vm_pro_no_such_thread	10
vm_pro_dup_name	14
vm_pro_no_such_process	16
vm_pro_internal_failure	21
vm_pro_bad_lineage	34
vm_pro_bad_plist_len	60
vm_pro_bad_dspclass_len	61
vm_pro_bad_flags_len	62
vm_pro_plist_too_big	195
vm_pro_not_dispatchable	220
vm_pro_ckpt_already_taken	226
vm_pro_data_not_available	227
vm_pro_name_truncated	228
vm_pro_interval_invalid	230
vm_pro_not_mt	243

For Synchronization

Symbolic name	Value
vm_syn_success	0
vm_syn_warning	4

© Copyright IBM Corp. 1992, 2023

Symbolic name	Value
vm_syn_error	8
vm_syn_mutex_already_exists	35
vm_syn_bad_scope_of_mutex	36
vm_syn_bad_mutex_name_len	37
vm_syn_insufficient_storage	38
vm_syn_handle_not_found	39
vm_syn_mutex_not_held	40
vm_syn_indeterminate_state	41
vm_syn_mutex_deleted	42
vm_syn_bad_wait_on_mutex	43
vm_syn_mutex_already_held	44
vm_syn_mutex_held_by_caller	45
vm_syn_name_not_found	46
vm_syn_not_mutex_creator	48
vm_syn_not_condvar_creator	49
vm_syn_bad_cnv_name_len	50
vm_syn_cnv_deleted	51
vm_syn_cnv_mutex_deleted	52
vm_syn_sem_already_exists	53
vm_syn_bad_scope_of_sem	54
vm_syn_bad_sem_name_len	55
vm_syn_not_sem_creator	56
vm_syn_cnv_already_exists	57
vm_syn_sem_deleted	58
vm_syn_limit_reached	59
vm_syn_mutex_not_reacquired	191

For Event Services

Symbolic name	Value
vm_evn_success	0
vm_evn_warning	4
vm_evn_error	8
vm_evn_bad_flag_size	63
vm_evn_bad_num_of_events	64
vm_evn_no_monitor	65

Symbolic name	Value
vm_evn_no_active_monitor	66
vm_evn_flag_truncated	67
vm_evn_key_truncated	68
vm_evn_name_truncated	69
vm_evn_event_truncated	70
vm_evn_bad_index	71
vm_evn_monitor_still_active	73
vm_evn_monitor_inactive	74
vm_evn_bad_data_len	76
vm_evn_data_truncated	80
vm_evn_already_waiting	91
vm_evn_cannot_satisfy	92
vm_evn_monitor_deleted	93
vm_evn_event_deleted	94
vm_evn_signal_lost	96
vm_evn_bad_key_len	104
vm_evn_bad_key_offset	105
vm_evn_bad_key	106
vm_evn_no_name	108
vm_evn_timeout	109
vm_evn_bad_time	111
vm_evn_bad_limit	112
vm_evn_bad_flag	114
vm_evn_dup_name	115
vm_evn_bad_event_count	118
vm_evn_bad_name_len	121
vm_evn_bad_token_size	127
vm_evn_token_truncated	130
vm_evn_bad_mask	139
vm_evn_not_authorized	144
vm_evn_name_too_long	171
vm_evn_insufficient_storage	208
vm_evn_not_mt	244

For Trace Services

Symbolic name	Value
vm_trc_success	0
vm_trc_warning	4
vm_trc_error	8
vm_trc_bad_func	164
vm_trc_bad_wrap_size	165
vm_trc_insufficient_storage	166
vm_trc_bad_numtype	167
vm_trc_array_bad_value	168
vm_trc_bad_datalen	170

For Accounting Services

Symbolic name	Value
vm_act_success	0
vm_act_warning	4
vm_act_error	8
vm_act_bad_func	172
vm_act_bad_numtype	173
vm_act_array_bad_value	174
vm_act_bad_id_flag	175
vm_act_insufficient_storage	176
vm_act_bad_time	190

For Interprocess Communication

Symbolic name	Value
vm_ipc_success	0
vm_ipc_warning	4
vm_ipc_error	8
vm_ipc_bad_search_seq	182
vm_ipc_bad_search_seq_len	183
vm_ipc_no_such_queue	184
vm_ipc_already_open	185
vm_ipc_bad_key_len	186
vm_ipc_bad_msg_len	187
vm_ipc_bad_handle	188

Symbolic name	Value
vm_ipc_queue_deleted	189
vm_ipc_not_authorized	192
vm_ipc_no_msg_available	193
vm_ipc_buf_too_small	194
vm_ipc_queue_not_empty	196
vm_ipc_not_implemented	197
vm_ipc_bad_name_len	198
vm_ipc_bad_export_level	199
vm_ipc_already_exists	200
vm_ipc_out_of_storage	201
vm_ipc_msgs_discarded	202
vm_ipc_bad_kokl	203
vm_ipc_bad_reply_handle	204
vm_ipc_reply_queue_deleted	205
vm_ipc_msg_discarded	206
vm_ipc_bad_reply_token	207
vm_ipc_primary_queue	208
vm_ipc_queue_closed	221
vm_ipc_bad_service_id	222
vm_ipc_service_undefined	223
vm_ipc_old_name_truncated	224
vm_ipc_bad_signal_flag	225
vm_ipc_timeout	232
vm_ipc_bad_timeout	233
vm_ipc_queue_in_use	234
vm_ipc_sid_in_use	237
vm_ipc_comm_retry	240
vm_ipc_comm_lost	241

For Timer Services

Symbolic name	Value
vm_tmr_success	0
vm_tmr_warning	4
vm_tmr_error	8
vm_tmr_format_invalid	209

Symbolic name	Value
vm_tmr_type_invalid	210
vm_tmr_cycle_invalid	211
vm_tmr_intervalunit_invalid	212
vm_tmr_interval_invalid	213
vm_tmr_invalid_call	214
vm_tmr_insufficient_storage	215
vm_tmr_tod_invalid	216
vm_tmr_zone_invalid	217
vm_tmr_cpqtime_failed	218
vm_tmr_unrecognized_token	219
vm_tmr_interval_exceeds_limit	229
vm_tmr_bad_min_format	300
vm_tmr_bad_sub_format	301
vm_tmr_bad_dif_format	302
vm_tmr_bad_format_combination	303
vm_tmr_bad_min_window_type	304
vm_tmr_bad_sub_window_type	305
vm_tmr_bad_dif_window_type	306
vm_tmr_min_conversion_error	307
vm_tmr_sub_conversion_error	308
vm_tmr_dif_conversion_error	309
vm_tmr_bad_min_length	313
vm_tmr_bad_sub_length	314
vm_tmr_bad_dif_length	315
vm_tmr_dif_truncated	316

For VCPU Services

Symbolic name	Value
vm_cpu_success	0
vm_cpu_warning	4
vm_cpu_error	8
vm_cpu_not_xa	235
vm_cpu_no_more_vcpus	236
vm_cpu_bad_number_to_create	238
vm_cpu_insufficient_storage	239

Symbolic name	Value
vm_cpu_bad_selection	242

For CMS Monitor Data

Symbolic name	Value
vm_mon_success	0

Appendix B. CMS Trace Record Formats

This information is provided for diagnosis purposes only.

The trace data is preceded by header information. The format of this header can be found in Table 13 on page 76.

Note: Each of the items listed in the Data column of these tables is a 4-byte field, unless otherwise indicated by a value in parentheses, such as thid (8).

Communication Trace Record Formats (Type 1)

Event	Subtype	Data
QueueCreate	0	caller tsd, qcb, oqb, level, caller
QueueOpen	1	caller tsd, local flag, qcb or qnqb, oqb, caller
QueueSend	2	caller tsd, qcb, oqb, qmcb, msg len, caller
QueueClose	3	caller tsd, qcb, oqb, caller
QueueQuery	4	caller tsd, qcb, oqb, caller
QueueReceiveImmed	5	caller tsd, qcb, oqb, qmcb, rcb, caller
QueueSendBlock	6	caller tsd, qcb, oqb, qmcb, msg len, caller
QueueReceiveBlock	7	caller tsd, qcb, oqb, qmcb, rcb, caller
QueueSendReply	8	caller tsd, qcb, oqb or -sid, qmcb, msg len, rqcb, roqb, rcb, caller
QueueReply	9	caller tsd, qcb, rcb, qmcb, msg len, caller
QueueDelete	10	caller tsd, qcb, caller
IPC TT Exit	11	caller tsd, whyblk, whyunbl, qcbptr, caller
QueueIdentifyService	12	caller tsd, -sid, old qcb, new qcb, caller
QueueSignalEvents	13	caller tsd, qcb, flag, caller
QueueIdentifyCarrier	14	caller tsd, first four bytes of carrier name, -sid, caller
IPC remote open	15	caller tsd, carrier RC, carrier RE, carrier token, caller
IPC surrogate open	16	caller tsd, local flag, qcb, oqb, oqb use count, caller
IPC remote close	17	caller tsd, carrier RC, carrier RE, carrier token, caller
IPC surrogate close	18	caller tsd, qcb, oqb, oqb count, caller
IPC remote send	19	caller tsd, carrier RC, carrier RE, carrier token, caller
IPC surrogate send	20	caller tsd, qcb, oqb, qmcb, msg len, caller
IPC remote send-block	21	caller tsd, carrier RC, carrier RE, carrier token, caller
IPC surrogate send-block	22	caller tsd, qcb, oqb, qmcb, msg len, caller
IPC remote send-reply	23	caller tsd, carrier RC, carrier RE, carrier token, caller
IPC surrogate send-reply	24	caller tsd, qcb, oqb, qmcb, msg len, caller
IPC remote reply	25	caller tsd, carrier RC, carrier RE, caller

© Copyright IBM Corp. 1992, 2023

Event	Subtype	Data
IPC surrogate reply	26	caller tsd, qcb, oqb, qmcb, msg len, caller
cmssigsetup	256	SIR addr, userdata, ovrdmask(8), termmask(8), retval, retcode, rsncode
cmsunsigsetup	257	SIR addr, userdata, ovrdmask(8), termmask(8), retval, retcode, rsncode
kill	258	pid, signal, options, retval, retcode, rsncode, footprints
pause	259	retval, retcode, rsncode
pthread_kill	260	thid(8), signal, options, retval, retcode, rsncode, footprints
queue_interrupt	261	retval, retcode, rsncode, footprints, signal, sigmask(8), penmask(8)
sigaction	262	signal, newsahdlr, newsamask(8), newsaflags, oldsahdlr, oldsamask(8), oldsaflags, userdata, retval, retcode, rsncode
sigpending	263	penmask(8), retval, retcode, rsncode
sigprocmask	264	how, newsigmask(8), oldsigmask(8),retval, retcode, rsncode
sigsuspend	265	sigmask(8), retval, retcode, rsncode
sigwait	266	sigmask(8), retval, retcode, rsncode
signal delivery	272	ppsd, footprints, signal, action, pprt, sigmask(8), penmask(8)
signal inheritance	273	newtpose, inputpprt, newpprt or inhe, footprints, sigmask(8), penmask(8)
extra exec inheritance	274	execpprt, sigmask(8), penmask(8)
signal generation	275	pprt, ppst, signal, options, sigda, action, senderpid, footprints, thid(8), sigmask(8), penmask(8)
kill target appendage	277	senderpid, signal, options, sigda, footprints
signal check utility	278	retcode, footprints, signal, action, pprt, sigmask(8), penmask(8)
kernel post	288	eventlist, retcode, kser, kserflags, tsd
entering kernel wait	289	entrycode, retcode, kser, kserflags, tsd
exiting kernel wait	290	entrycode, retcode, kser, kserflags, tsd

Dispatch Trace Record Formats (Type 2)

Event	Subtype	Data
block	0	caller tsd, caller
unblock	1	caller tsd, object tsd, caller
promote	2	caller tsd, chosen tsd, object dcd, caller
switch	3	incoming tsd, outgoing tsd, CPU time (two words), caller
schedule	4	caller tsd, object tsd, caller
unschedule	5	caller tsd, object tsd, caller

Process Management Trace Record Formats (Type 3)

Event	Subtype	Data
ThreadCreate	0	caller tsd, new tsd, new entry point, new thread's DCD, caller
ThreadDelete	1	caller tsd, object tsd, caller
ThreadSuspend	2	caller tsd, object tsd, new susp count, caller
ThreadResume	3	caller tsd, object tsd, new susp count, caller
ThreadSetPriority	4	caller tsd, object tsd, new prio, caller
ThreadSetDispatchClass	5	caller tsd, object tsd, new dcd, caller
ThreadYield	6	caller tsd, caller
ThreadDelete (other)	7	caller tsd, object tsd, my_cpuid, tsd stap field, caller (Note: In the next to last two fields, only the lower halfword is significant.)
ThreadDelete (me)	8	caller tsd, object tsd, caller
forced delete	9	caller tsd, caller
thrdel exit driver	10	caller tsd, last thread flag, caller
thrdel rm exit ret	11	caller tsd, rm exit rc, caller
process cleanup	12	caller tsd, caller psd, object psd, caller

Language Adapter Trace Record Formats (Type 4)

Event	Subtype	Data
HOTL from RTERR	0	A(TA), A(EP), EnvHandle, caller
HOTU from RTERR	1	A(TA), A(EP), EnvHandle, caller
HOTT from RTERR	2	A(TA), A(EP), EnvHandle, caller
HOTT from RTETDE	3	A(TA), 0, EnvHandle, caller
RTETDE complete	4	A(TA), TA, caller
RTETCR complete	5	A(TA), A(CRTEAB), caller

Synchronization Trace Record Formats (Type 5)

Event	Subtype	Data
CondVarCreate	0	caller tsd, handle, cvcb, mcb, caller
CondVarDelete	1	caller tsd, handle, caller
CondVarSignal	2	caller tsd, handle, caller
CondVarWait	3	caller tsd, handle, caller
MutexAcquire	4	caller tsd, handle, caller
MutexCreate	5	caller tsd, handle, mcb, caller
MutexDelete	6	caller tsd, handle, caller
MutexRelease	7	caller tsd, handle, caller

Event	Subtype	Data
SemCreate	8	caller tsd, handle, scb, caller
SemDelete	9	caller tsd, handle, caller
SemReInit	10	caller tsd, handle, caller
SemSignal	11	caller tsd, handle, caller
SemWait	12	caller tsd, handle, caller

Miscellaneous Trace Record Formats (Type 6)

Event	Subtype	Data
MemAllocate	0	caller tsd, size, block address, caller
MemRelease	1	caller tsd, size, block address, caller
Abend	2	error code, type, tid, abend address, general registers

Appendix C. Remote IPC Support

This appendix describes CMS's implementation of remote interprocess communication (IPC). It provides information describing the interface between the CMS kernel and carriers, and it also gives information about the operation of the CMS APPC/VM carrier. It should be read by systems programmers interested in providing extensions or modifications to support remote IPC operations.

Functional Overview

To implement remote IPC operations, the CMS kernel relies upon service threads, known as *IPC carrier threads*, to provide the communication services connecting one instance of CMS with another. The basic idea is that CMS associates a communication carrier with each network queue of local interest, the mapping being defined in $QUEUES$ NAMES. When an application program performs an IPC operation, CMS examines the request and determines whether remote activity is required. If remote operations are needed, CMS passes the request to the carrier associated with the queue. The carrier conveys the request to the remote kernel and responds to the local kernel when the operation is complete. In addition to responding to requests generated locally, the carrier responds to requests arriving on its communication medium. When these occur, the carrier calls the local kernel, gathers the kernel's response, and ships the response out over its medium.

CMS is not limited to supporting one carrier per session. Multiple carriers can exist simultaneously and operate without knowledge of one another. CMS routes remote IPC requests to the respective carriers as needed. CMS is also not limited to supporting one thread per carrier. A single carrier may be composed of multiple threads if desired.

The interface between CMS and carriers is based upon service queues. The CMS kernel routes remote IPC operations to a carrier by placing messages representing those operations into a service queue defined by the carrier. The carrier processes the operations by reading those messages from the service queue, taking appropriate action, and replying to the messages as the operations complete. Similarly, when remote activity arrives at the carrier, the carrier sends a request to a kernel service queue and waits for a response. When the response appears, the carrier ships the response out over the communication medium.

To identify itself to the kernel, a carrier performs two operations:

1. Using QueueIdentifyService, it *identifies a service queue*. This queue is the one into which CMS places messages representing remote IPC requests requiring the attention of this carrier. The carrier will learn of new remote IPC work by reading requests from its service queue.

   The carrier may choose to isolate service requests by having the CMS kernel place them in a queue of their own. If so, the carrier first needs to use QueueCreate to create the service queue. This isolation is not required, though. For example, the carrier may choose to use its process' primary queue as its service queue.

2. Using QueueIdentifyCarrier, it *identifies itself as a remote IPC carrier*. This binds the carrier's name, such as *Internet*, to the service queue identified on the call to QueueIdentifyService. The carrier name can be up to 16MB-1 bytes long.

After the carrier performs these two operations, the CMS kernel begins routing remote IPC operations to the carrier. These remote operations arrive at the carrier as messages in the carrier's service queue. To collect request messages from the service queue, the carrier may use QueueReceiveBlock or QueueReceiveImmed. To respond to those requests, the carrier should use QueueReply. Figure 30 on page 308 shows how this portion of the carrier might be organized.

© Copyright IBM Corp. 1992, 2023

```
          START
Initialize communication medium;
QueueCreate ( service_queue );
QueueIdentifyService ( service_queue, service_id );
QueueIdentifyCarrier ( carrier_name, service_id );
Do forever;
   QueueReceiveBlock ( some request );
   Transmit request to remote carrier;
   Receive response from remote carrier;
   QueueReply ( response );
End;
Return;
          END
```

Figure 30. Skeleton of an IPC Carrier (Kernel-Initiated Requests)

Just as it must respond to requests generated by the local kernel, the carrier must also respond to requests arriving on its communication medium. For these requests, the carrier works with the local kernel to satisfy the request. For this kind of interaction, CMS provides a service queue to which carriers should send requests representing remote activity. This service queue is called the *network IPC service queue*.

The network IPC service queue has service ID **-5**. To send messages to the queue, the carrier uses the -5 service ID as the queue handle in its QueueSendReply calls. The kernel responds to such service requests by using QueueReply. These responses are placed in the queue specified by the carrier on its QueueSendReply call. The carrier may wish to use its own service queue as the reply queue. This is perfectly acceptable. A carrier should use one of the receive calls (QueueReceiveBlock or QueueReceiveImmed) to pick up responses from the kernel.

Figure 31 on page 309 shows how this portion of the carrier might be organized.

```
            START

Initialize communication medium;

Do forever;

   Receive a remote request from communication medium;

   Build description of request to send to local kernel;

   QueueSendReply ( rc, re, -5, ... );

   QueueReceiveBlock ( rc, re, ... );

   Build response for remote requester;

   Transmit response on communication medium;

End;

            END
```

Figure 31. Skeleton of an IPC Carrier (Carrier-Initiated Requests)

When a CMS application ends, it may be necessary for the carrier to perform cleanup operations (deallocate conversations, release storage, and so forth). CMS does not specifically notify carriers that an application has terminated. Rather, a carrier may use Event Services to monitor the VMPROCESSEND event. When the event is signaled, the carrier's monitor will be activated and the carrier can perform cleanup operations.

CMS notifies carriers as network-level queues come and go in the local session. This lets carriers manage the cleanup of remotely-originating operations on a per-queue basis if they desire and deactivate themselves when no network-level queues exist locally.

Interface Definition

Of the entire CMS IPC API, only six functions require the assistance of IPC carriers for remote activity. These functions are:

- QueueOpen
- QueueSend
- QueueSendBlock
- QueueSendReply
- QueueReply
- QueueClose.

In addition, CMS notifies IPC carriers when the following functions create network-level queues locally:

- QueueCreate
- QueueDelete.

CMS performs the rest of the queue operations without the help of carriers. Thus, the messages exchanged between CMS and IPC carriers are concerned with only these operations.

All requests and responses exchanged between CMS and a carrier are passed in a data structure known as the *Queue Carrier Request Block* (QCRB). When either component (kernel or carrier) wishes service from the other, it builds a QCRB describing the request and sends the QCRB to the other component through QueueSendReply. The receiving component performs the operation, builds a QCRB containing the response, and sends the QCRB using QueueReply. Neither component is allowed to respond to the other until the request is completed; that is, remote IPC operations are fully synchronous. This allows

applications to be written such that they are insensitive to whether they are working in a distributed fashion.

Figure 32 on page 310 illustrates the exchange of QCRBs in a sample client-server environment. For a request to go from the client to the server, the following procedure takes place:

1. The client program sends a message to a server's queue, using QueueSendReply.
2. On the client side, a request QCRB flows from the kernel to the IPC carrier. This QCRB contains the string "IPC0" in its message key.
3. The client-side carrier uses the communication medium to send the request to the server-side carrier.
4. On the server side, the carrier gives the request to the kernel in a request QCRB. This QCRB contains the string "IPC2" in its message key.
5. The server-side kernel places the message in the server's queue.
6. The server-side kernel responds to the carrier indicating that the message has been placed in the queue. To do so, it uses a QCRB containing the string "IPC3" in its message key.
7. The server-side carrier uses the communication medium to tell the client-side carrier that the operation completed.
8. On the client side, the carrier tells the kernel that the operation completed. To do so, it uses a QCRB containing the string "IPC1" in its message key.
9. The client thread's QueueSendReply operation completes.

At this point the client thread is released, learning of the operation's completion. Note that though the client thread is blocked while this operation occurs, the client program is free to use additional threads to accomplish work while the message is being delivered.

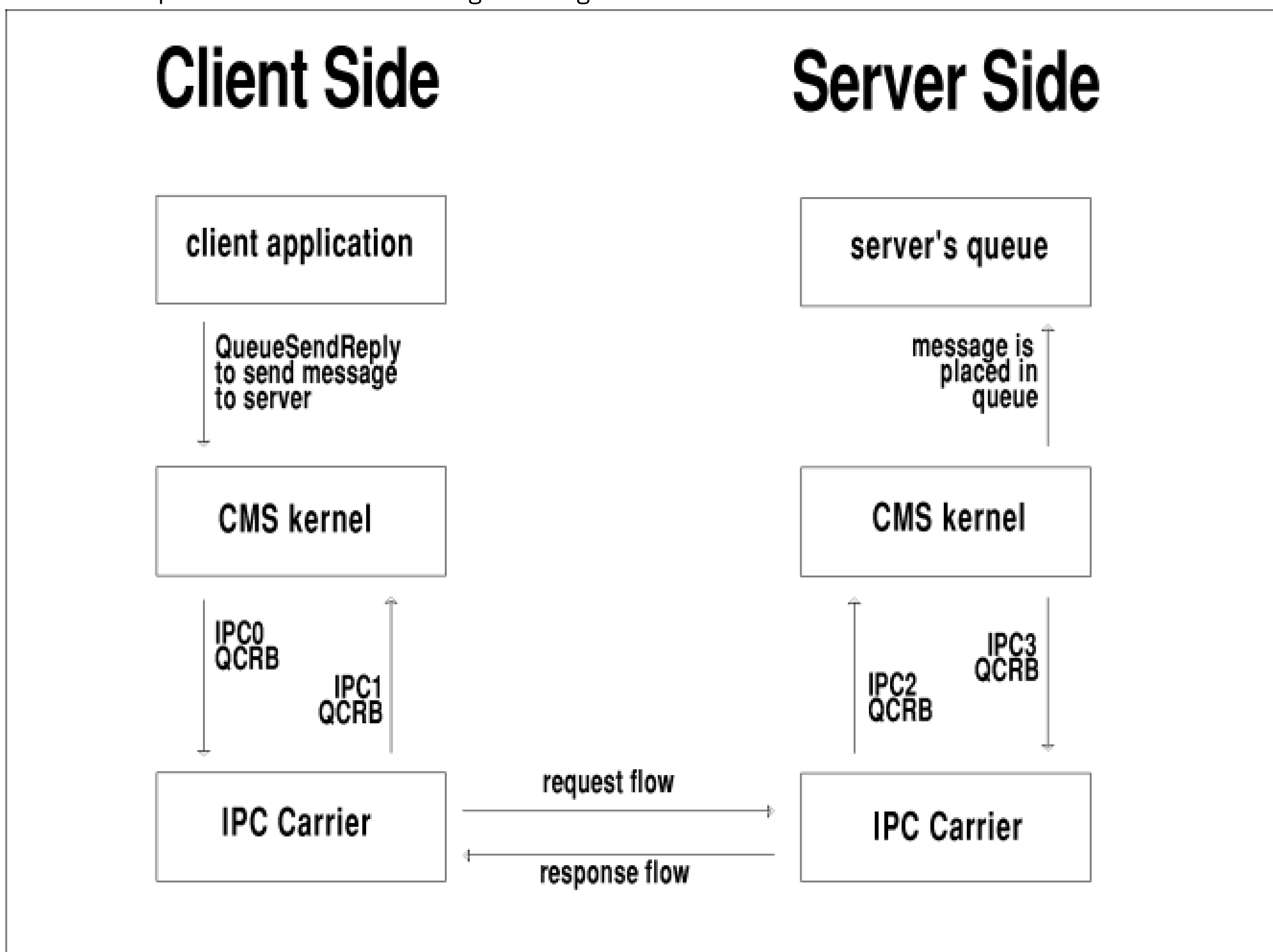


Figure 32. Flow of Requests and Responses for Distributed IPC

Of course, for the server program to send a response to the client program, the procedure occurs again, this time in reverse. Thus an IPC carrier must be prepared to handle both activity originating at the local kernel and activity originating remotely.

No matter whether it contains a request or response, a QCRB is made up of a header followed by an array of fullwords. The header describes the kind of request and contains sequence numbers and other control information. The array of fullwords contains request-specific parameters or data.

The header portion of a QCRB is organized as follows:

Offset	Field	Description
0	QCRB type identifier	A 4-byte EBCDIC string identifying the kind of QCRB, as follows: **IPC0** Request from kernel to carrier **IPC1** Response from carrier to kernel **IPC2** Request from carrier to kernel **IPC3** Response from kernel to carrier
4	Sequence number	For request QCRBs (IPC0 and IPC2), this number is generated by the requester and uniquely identifies the request. The number must be usable in a match key. For response QCRBs (IPC1 and IPC3), this number is the number of the provoking request.
8	Request type	Tells what kind of request or response the QCRB represents. The field contains one of these values: **Value** **Meaning** **0** QueueOpen **1** QueueSend **2** QueueSendBlock **3** QueueSendReply **4** QueueReply **5** QueueClose **6** QueueCreate **7** QueueDelete

Offset	Field	Description
12	Parameter count	Given in fullwords. Tells how many fullwords are in use in the parameter array.
16	Parameter array	An 16-word array of fullwords, the use of which is request-dependent. Though some of the fullwords may not be used for some kinds of QCRBs, the transmitter must always send all 16 fullwords.

When a request QCRB is sent, it must be sent with a key offset of 0 and a key length of 4. When a response QCRB is sent, it must be sent with a key offset of 0 and a key length of 8. Again, the sequence numbers generated by the request originators must be usable in match keys. This means that the numbers' bytes must not contain any byte values that might be interpreted as key wildcards.

The following tables describe the use of the parameter arrays in the four kinds of QCRBs. In the tables, the following abbreviations are used:

Abbr
  Meaning

tagp
  Pointer to tag structure

cqt
  Carrier queue token

mp
  Pointer to message text

ml
  Length of message text

ko
  Key offset

kl
  Key length

uidp
  Pointer to 8-byte user ID of sender

pid
  Process ID of sender

to
  Timeout value

rqh
  Reply queue handle

rpid
  Process ID of owner of reply queue

csn
  Carrier sequence number

rc
  Return code

re
  Reason code

kqt
  Kernel queue token

qnp
  Queue name pointer

qnl
Queue name length

csid
Carrier service ID

cuword
Carrier user word

In addition, some cells in the tables refer to the numbered usage notes appearing after the tables themselves.

IPC0 QCRBs (Kernel Request to Carrier)

Table 24. IPC0 QCRB Parameter Usage (Kernel Request to Carrier)

Index	QOpen	QSend	QSendB	QSendR	QReply	QClose	QCreate	QDelete
1	tagp (note "1" on page 314)	cqt	cqt	cqt	-	cqt	tagp (note "1" on page 314) (note "13" on page 315)	tagp (note "1" on page 314) (note "13" on page 315)
2	-	mp	mp	mp	mp	-	-	-
3	-	ml	ml	ml	ml	-	-	-
4	-	ko	ko	ko	ko	-	-	-
5	-	kl	kl	kl	kl	-	-	-
6	-	uidp	uidp	uidp	uidp	-	-	-
7	-	pid	pid	pid	pid	-	-	-
8	-	-	to	rqh (note "3" on page 315)	rpid (note "4" on page 315)	-	-	-
9	-	-	-	-	rqh (note "5" on page 315)	-	-	-
10	-	-	-	-	csn (note "6" on page 315)	-	-	-
11	-	-	-	-	cuword (note "7" on page 315)	-	-	-
12..16	-	-	-	-	-	-	-	-

IPC1 QCRBs (Carrier Response to Kernel)

Table 25. IPC1 QCRB Parameter Usage (Carrier Response to Kernel)

Index	QOpen	QSend	QSendB	QSendR	QReply	QClose
1	rc (note "11" on page 315)	rc	rc	rc	rc	rc

Table 25. IPC1 QCRB Parameter Usage (Carrier Response to Kernel) (continued)

Index	QOpen	QSend	QSendB	QSendR	QReply	QClose
2	re	re	re	re	re	re
3	cqt (note "2" on page 315)	-	-	-	-	-
4..16	-	-	-	-	-	-

IPC2 QCRBs (Carrier Request to Kernel)

Table 26. IPC2 QCRB Parameter Usage (Carrier Request to Kernel)

Index	QOpen	QSend	QSendB	QSendR	QReply	QClose
1	qnp	kqt	kqt	kqt	-	kqt
2	qnl	mp	mp	mp	mp	-
3	-	ml	ml	ml	ml	-
4	-	ko	ko	ko	ko	-
5	-	kl	kl	kl	kl	-
6	-	uidp	uidp	uidp	uidp	-
7	-	pid	pid	pid	pid	-
8	-	-	to	rqh	rpid	-
9	-	-	-	csid (note "9" on page 315)	rqh (note "5" on page 315)	-
10	-	-	-	cuword (note "10" on page 315)	-	-
11..16	-	-	-	-	-	-

IPC3 QCRBs (Kernel Response to Carrier)

Table 27. IPC3 QCRB Parameter Usage (Kernel Response to Carrier)

Index	QOpen	QSend	QSendB	QSendR	QReply	QClose
1	rc (note "12" on page 315)	rc	rc	rc	rc	rc
2	re	re	re	re	re	re
3	kqt (note "8" on page 315)	-	-	-	-	-
4..16	-	-	-	-	-	-

Usage Notes

1. The tags structure is organized as follows:

   Offset

   Use

0
: Reserved for kernel

4
: Reserved for kernel

8
: Reserved for kernel

12
: Number of tags found

16
: Reserved for kernel

20
: Address of length array

24
: Address of pointer array

Each entry in the pointer array points to a character string buffer containing a single tag and its corresponding value. The tag appears first in the buffer, and the value follows it. The tag and value are separated from one another by a space character.

Each entry in the length array gives the length in bytes of the string pointed to by the corresponding entry in the pointer array.

The *number of tags found* field gives the length in fullwords of the pointer and length arrays.

If the carrier requires a persistent copy of the tags structure, it must make one. The carrier should not assume that the tags structure will persist.

2. The token is generated by the carrier and is used by the kernel to identify the opened queue on subsequent IPC0 requests.
3. This is the handle under which the sender has the reply queue open.
4. This is the process ID of the process owning the reply queue.
5. This is the handle under which the process owning the reply queue has the reply queue open.
6. This is the sequence number the carrier used when it submitted the IPC2 request QCRB for the provoking QueueSendReply.
7. This is the carrier user word the carrier passed when it submitted the IPC2 request QCRB for the provoking QueueSendReply.
8. The token is generated by the kernel and should be used by the carrier to identify the queue on subsequent IPC2 requests.
9. This is the service ID of the carrier's service queue.
10. The kernel will send this value back to the carrier when the corresponding QueueReply occurs. This assists the carrier in keeping track of where the reply should be sent.
11. These return and reason codes are delivered directly to the CMS application. The carrier author should carefully consider whether returning values other than those shown in the function descriptions is warranted.
12. These return and reason codes generated by the kernel match those mentioned earlier in the function descriptions.
13. The QueueCreate and QueueDelete notifications are sent to the carrier through QueueSend, *not* QueueSendReply. The carrier should not reply to these notifications but simply take whatever action makes sense for its environment.

APPC/VM Carrier Line Flows

This section describes the protocols and line flows used between two instances of the CMS APPC/VM IPC carrier. The flows are documented for two reasons:

1. To help other carrier authors see how they might organize their own line flows.
2. To help programmers on other APPC-supporting platforms see how to write code that imitates both the CMS APPC/VM carrier and CMS's IPC subsystem. This might be useful for putting a functional equivalent of the CMS IPC API on another platform.

Structure

The following structure notes describe the overall design of the APPC/VM carrier.

- Two CMS APPC/VM IPC carriers maintain one APPC/VM conversation connecting them. All IPC activity between the two kernels flows on this one conversation.
- APPC/VM conversations are allocated only in response to QueueOpen requests and are never deallocated.
- If a conversation should become deallocated (for example, the other side re-IPLs), all queues open over the conversation must be re-opened.
- The conversation is an APPC basic conversation and uses SYNCLVL=NONE.
- On each side of the link, the transaction program name on the other side is assumed to be *VMIPC*.

Request Flows

When the APPC/VM carrier sends a request, it sends a *request header record* followed by zero or more *request data records*. The header record describes the request and gives parameter information to describe the request. The data records pass request-specific data.

A request header record is always 64 bytes long. Table 28 on page 317 describes the formats of the request header records for the various request types. The following abbreviations are used in the table cells:

Abbr
: **Meaning**

cseq
: Sequence number assigned by requesting carrier

pid
: Process ID of originating process

-
: Reserved for future use

rtok
: Queue token assigned by remote carrier

qnl
: Queue name length

ml
: Message length

ko
: Key offset

kl
: Key length

uid1
: First 4 bytes of user ID

uid2
: Second 4 bytes of user ID

to
: Timeout value

rqh
: Reply queue handle

rpid
ID of process to which reply should be delivered

Table 28. Request Header Record Formats

Offset	QOpen	QSend	QSendB	QSendR	QReply	QClose
0	0	0	0	0	0	0
4	cseq	cseq	cseq	cseq	cseq	cseq
8	0	1	2	3	4	5
12	-	rtok	rtok	rtok	-	rtok
16	qnl	ml	ml	ml	ml	-
20	-	ko	ko	ko	ko	-
24	-	kl	kl	kl	kl	-
28	-	uid1	uid1	uid1	uid1	-
32	-	uid2	uid2	uid2	uid2	-
36	-	pid	pid	pid	pid	-
40	-	-	to	-	-	-
44	-	-	-	rqh	rqh	-
48	-	-	-	-	rpid	-
52..63	-	-	-	-	-	-

Request data is sent in request data records, 32,765 bytes in each record, until all request data has been sent. Only the last request data record may contain less than 32,765 bytes of request data. Request data records are used as follows:

Function	Usage
QueueOpen	Queue name
QueueSend	Message text
QueueSendBlock	Message text
QueueSendReply	Message text
QueueReply	Message text
QueueClose	not used (no transmission)

Response Flows

When the APPC/VM carrier responds to a request, it transmits a *response header record*.

Note: Because no responses require additional data, there is no notion of a "response data record". The "header" terminology is used, though, for symmetry.

A response header record is always 64 bytes long. Table 29 on page 318 describes the formats of the response header records for the various response types. The following abbreviations are used in the table cells:

Abbr
Meaning

cseqp
Sequence number of provoking request

rc
: Return code

re
: Reason code

-
: Reserved for future use

rtok
: Queue token assigned by responding carrier

Table 29. Response Header Record Formats

Offset	QOpen	QSend	QSendB	QSendR	QReply	QClose
0	1	1	1	1	1	1
4	cseqp	cseqp	cseqp	cseqp	cseqp	cseqp
8	rc	rc	rc	rc	rc	rc
12	re	re	re	re	re	re
16	rtok	-	-	-	-	-
20..63	-	-	-	-	-	-

Appendix D. Example of a C Multitasking Program

To help you get started writing multitasking applications, this section presents a simple multi-threaded C program. This program illustrates the key parts of any multitasking-based application and a simple but powerful method of handling concurrent work.

The example uses an approach in which a primary thread determines what work needs to be done and distributes this work to worker threads for processing. It also shows how to structure an application according to a message-object model. In this model each thread represents an operation to be performed on some object. Receiving a message on a queue causes a thread to perform its operation using the data in the message.

The problem being solved is fairly trivial and could easily be solved without multitasking. This allows the example to focus on the multitasking principle rather than the difficulties of the problem itself. The main program simply reads commands from the CMS console and sends them to the thread that handles this particular command. The valid commands are:

DUPLICATE *string*
: Display the string on the user's terminal

REPEAT *n string*
: Display the string on the user's terminal n times

REVERSE *string*
: Transpose the string and display it on the user's terminal.

The simple structure of this small program is applicable to large servers whose threads perform complex functions.

```
/*
   Read a command from the command line and forward it via a queue
   to the thread that processes this command.

   The valid commands are:

   DUPLICATE - Display the data on the user's terminal
   REPEAT n  - Display the data on the user's terminal n times,
               when n is from 0 to 9
   REVERSE   - Transpose the data and display it on the user's terminal

   All other commands result in an error message.
*/

#include <stdio.h>
#include <ctype.h>
#include "vmcmt.h"

#define LINESIZE 131
#define FLAGSIZE 5
#define CMDNUM 3
#define DUPLICATE 0
#define REPEAT 1
#define REVERSE 2
#define NO 0
#define YES 1

char cmd_queue[16] = { "CommandQueue\0" };
char *cmd_list[CMDNUM] = { "DUPLICATE\0", "REPEAT\0", "REVERSE\0" };
int cmd_q_han;                        /* Handle of command queue         */

#pragma linkage(applmain,OS)

int applmain ( ext_plist, tok_plist )
   char * ext_plist;
   char * tok_plist;

{
   extern int rev_thd ( void );     /* Thread for REVERSE              */
   extern int rep_thd ( void );     /* Thread for DUPLICATE and REPEAT */

   int rc,                          /* Return code                     */
```

```
        re,                         /* Reason code                    */
        index,                      /* Multipurpose index for looping  */
        valid_cmd,                  /* Flag to indicate valid command  */
        repeat_tid,                 /* Thread ID for REPEAT thread     */
        duplicate_tid,              /* Thread ID for DUPLICATE thread  */
        reverse_tid,                /* Thread ID for REVERSE thread    */
        number_of_threads,          /* Number of threads created       */
        tc_flags[2],                /* Flags for ThreadCreate          */
        tc_plist[1],                /* Parameter list for threads      */
        command_len,                /* Length of command name entered  */
        data_len;                   /* Length of user input            */

    char prompt[] = {"Enter a command or press [enter] to quit.\0"},
         error_msg1[] = {"The command you entered is invalid.\n\0"},
         error_msg2[] = {"Please enter a valid command or press \0"},
         error_msg3[] = {"[enter] to quit.\n\0"};

    char command_line[LINESIZE],           /* Buffer for user input    */
         msg_line[FLAGSIZE + LINESIZE],    /* Buffer to send message   */
         rcv_line[FLAGSIZE];               /* Buffer to receive message*/

    char cont[FLAGSIZE+1] = { "CONT \0" }; /* Continuation indicator   */
    char *curr_ptr,                        /* Index into user input    */
         *start_ptr,                       /* Start of actual command  */
         *end_ptr;                         /* End of user input + 1    */

    char done[FLAGSIZE] = { "DONE\0" };    /* Work completed indicator */
    char sender_UID[8];                    /* User ID of message sender*/
    int  sender_PID,                       /* Process ID of sender     */
         message_length,                   /* Length of message rcvd   */
         key_offset,                       /* Offset of key in message */
         key_length,                       /* Length of message key    */
         reply_token;                      /* Queue reply token        */

    /*
       Create a process-level queue on which to send the commands.
    */

    QueueCreate
    (
       &rc,                            /* return code                  */
       &re,                            /* reason code                  */
       cmd_queue,                      /* name of command queue        */
       strlen(cmd_queue),              /* length of command queue name */
       vm_ipc_plevel,                  /* scope of queue               */
       &cmd_q_han                      /* handle of queue              */
    );

    if (rc != vm_rc_success)
    {
       printf ("QueueCreate failed with ");
       printf ("Return code = %d. Reason code = %d.\n", rc, re);
       printf ("Trying to create %s.\n", cmd_queue);
       return (8);
    }

    /*
       Create the threads to process the commands.  Create each thread
       in its own class so that it will have the opportunity to run
       concurrently.  Make each thread a slightly higher priority than
       the main program so that it is given the opportunity to process
       the current command before the main program gets another one.
       For threads that require a parameter list, have each make a
       copy of the parameter list because the main program will be
       reusing it.
    */
    number_of_threads = 0;

    tc_flags[0] = vm_pro_new_class;
    tc_flags[1] = vm_pro_copy_plist;

    /*
       Create a thread to handle DUPLICATE.  The code for this thread
       handles both DUPLICATE and REPEAT, so we must pass the command
       name as a parameter to the thread.
    */

    tc_plist[0] = (int) cmd_list[DUPLICATE];
    ThreadCreate
    (
       &rc,             /*  rc                */
       &re,             /*  re                */
```

```
      &duplicate_tid,  /*  thread ID         */
      tc_flags,        /*  flags             */
      2,               /*  flags length      */
      1,               /*  relative priority */
      (int) rep_thd,   /*  entry point       */
      tc_plist,        /*  plist address     */
      1                /*  plist length      */
   );

   if (rc != vm_rc_success)
   {
      printf ("ThreadCreate failed with ");
      printf ("Return code = %d. Reason code = %d.\n", rc, re);
      printf ("Trying to create thread for %s.\n",
               cmd_list[DUPLICATE]);
      cleanup ( msg_line, number_of_threads );
      return (8);
   }

   /*
      Create a thread to handle REPEAT.  The code for this thread
      handles both DUPLICATE and REPEAT, so we must pass the command
      name as a parameter to the thread.
   */

   number_of_threads += 1;
   tc_plist[0] = (int) cmd_list[REPEAT];
   ThreadCreate
   (
      &rc,             /*  rc                */
      &re,             /*  re                */
      &repeat_tid,     /*  thread ID         */
      tc_flags,        /*  flags             */
      2,               /*  flags length      */
      1,               /*  relative priority */
      (int) rep_thd,   /*  entry point       */
      tc_plist,        /*  plist address     */
      1                /*  plist length      */
   );

   if (rc != vm_rc_success)
   {
      printf ("ThreadCreate failed with ");
      printf ("Return code = %d. Reason code = %d.\n", rc, re);
      printf ("Trying to create thread for %s.\n",
               cmd_list[REPEAT]);
      cleanup ( msg_line, number_of_threads );
      return (8);
   }

   /*
      Create a thread to handle REVERSE.  It handles only this command
      so there is no need to pass it a parameter; the command name
      has been declared globally.
   */

   number_of_threads += 1;
   ThreadCreate
   (
      &rc,             /*  rc                */
      &re,             /*  re                */
      &reverse_tid,    /*  thread ID         */
      tc_flags,        /*  flags             */
      2,               /*  flags length      */
      1,               /*  relative priority */
      (int) rev_thd,   /*  entry point       */
      tc_plist,        /*  plist address     */
      0                /*  plist length      */
   );

   if (rc != vm_rc_success)
   {
      printf ("ThreadCreate failed with ");
      printf ("Return code = %d. Reason code = %d.\n", rc, re);
      printf ("Trying to create thread for %s.\n",
               cmd_list[REVERSE]);
      cleanup ( msg_line, number_of_threads );
      return (8);
   }
   number_of_threads += 1;

   /*
```

```
      Each thread is waiting to receive a message on the CommandQueue
      in the following format:  key user_data

      where key consists of the following:  flag command

      where flag tells the thread whether or not this is the notice to
      quit, and command is the name of the command being processed.
      Now we must initialize the flag so the threads will process
      the commands.
   */

   strcpy( msg_line, cont );

   /*
      Prompt the user for input
   */

   printf ("%s\n", prompt);

   /*
      Read commands until the user presses only the [Enter] key.
   */

   curr_ptr = gets( command_line );
   data_len = strlen( command_line );
   while ( curr_ptr != NULL && data_len != 0 )
   {
      end_ptr = curr_ptr + data_len;  /* find the end of the input   */

      /*
         Scan past leading blanks to find the command name.
      */

      while ( isspace( (int) *curr_ptr ) != 0 &&
              curr_ptr < end_ptr )
      {
         curr_ptr += 1;               /* point to next character     */
         data_len -= 1;               /* decrement length of input   */
      }

      /*
         Scan until we find a space (blank, tab or newline character)
         or reach the end of the line, translating each character
         to uppercase if it is currently in lowercase.  Track the
         size of the command name.
      */

      start_ptr = curr_ptr;
      command_len = 0;
      while ( isspace( (int) *curr_ptr ) == 0 &&
              curr_ptr < end_ptr )
      {
         if ( islower( (int) *curr_ptr ) != 0 )
            *curr_ptr = toupper( (int) *curr_ptr );
         command_len += 1;
         curr_ptr += 1;
      }

      /*
         Search our list of commands to see if the command is valid.
      */

      index = 0;
      valid_cmd = NO;
      while ( index < CMDNUM && valid_cmd == NO )
         if ( command_len == strlen( cmd_list[index] ) &&
              memcmp( start_ptr, cmd_list[index], command_len ) == 0)
            valid_cmd = YES;
         else
            index += 1;

      if ( valid_cmd == YES )
      {

         /*
            This is a valid command; concatenate it to the flag and
            send it to be processed.
         */

         curr_ptr = msg_line + FLAGSIZE; /* point past the flag */
         memcpy( curr_ptr, start_ptr, data_len + 1 );
```

```
          QueueSendReply
          (
             &rc,                    /* return code             */
             &re,                    /* reason code             */
             cmd_q_han,              /* queue handle            */
             msg_line,               /* message                 */
             strlen(msg_line),       /* message length          */
             0,                      /* key offset              */
             FLAGSIZE + command_len, /* key length              */
             cmd_q_han               /* queue to reply on       */
          );

         if (rc != vm_rc_success)
         {
            printf ("QueueSendReply failed with ");
            printf ("Return code = %d. Reason code = %d.\n", rc, re);
            printf ("Message sent was %s.\n", msg_line);
            printf ("Key offset = 0, key length = %d.\n",
                     FLAGSIZE + command_len);
            curr_ptr = NULL;
         }
         else
         {

            /*
               Wait until processing is complete.
            */

            QueueReceiveBlock
            (
              &rc,              /* return code                         */
              &re,              /* reason code                         */
              cmd_q_han,        /* queue handle of command queue       */
              done,             /* key to match on                     */
              strlen(done),     /* length of key                       */
              20,               /* timeout period                      */
              rcv_line,         /* receive message buffer              */
              sizeof(rcv_line), /* size of message buffer              */
              &message_length,  /* length of message received          */
              &key_offset,      /* offset of key in message            */
              &key_length,      /* length of key                       */
              sender_UID,       /* sender user ID                      */
              &sender_PID,      /* sender process ID                   */
              &reply_token      /* reply token                         */
            );

            if (rc != vm_rc_success)
            {
               printf ("QueueReceiveBlock failed with ");
               printf ("Return code = %d. Reason code = %d.\n", rc, re);
               curr_ptr = NULL;
            }
            else

               /*
                  Prompt for more input.
               */

               printf ("%s\n", prompt);
         }
       }
       else
       {

          /*
             This is an invalid command; inform the user.
          */

          printf ("%s", error_msg1);
          printf ("%s", error_msg2);
          printf ("%s", error_msg3);
       }

       /*
          Continue to read, if we have not encountered a critical error.
       */

       if ( curr_ptr != NULL )
       {
          curr_ptr = gets( command_line );
          data_len = strlen( command_line );
       }
```

```
    }

    cleanup ( msg_line, number_of_threads );
    return ( 0 );
}

/*
    Inform all of the threads that processing should stop, and wait
    until all the threads have completed their commands.  Then delete
    the queue.  The threads will return by themselves.
*/

cleanup  ( msg_line, number_of_threads )
    char msg_line[FLAGSIZE + LINESIZE]; /* Message buffer to use       */
    int number_of_threads;              /* Number of threads to inform */

{
    int rc, re, index;
    char quit[FLAGSIZE+1] = { "QUIT \0" };

    /*
       Tell all the threads that processing should stop. Wait until
       they receive this message. This means that they have
       completed processing all of their commands.
    */

    strcpy( msg_line, quit );
    for ( index = 0; index < number_of_threads; index++ )
    {
       msg_line[FLAGSIZE] = '\0';
       strcat( msg_line, cmd_list[index] );
       msg_line[FLAGSIZE + strlen( cmd_list[index] )] = '\0';

       QueueSendBlock
       (
          &rc,                   /* return code             */
          &re,                   /* reason code             */
          cmd_q_han,             /* queue handle            */
          msg_line,              /* message                 */
          strlen(msg_line),      /* message length          */
          0,                     /* key offset              */
          strlen(msg_line),      /* key length              */
          0                      /* timeout period          */
       );

      if (rc != vm_rc_success)
      {
         printf ("QueueSendBlock failed with ");
         printf ("Return code = %d. Reason code = %d.\n", rc, re);

         /*
            Delete all the other threads in the process because
            we cannot send them the message to quit.
         */

         ThreadDelete
         (
            &rc,
            &re,
            -1
         );

         /*
            No point in looping any longer.
         */

         index = number_of_threads;
       }
    }

    /*
       Now it is safe to delete the queue.
    */

    QueueDelete
    (
       &rc,                         /* return code             */
       &re,                         /* reason code             */
       cmd_queue,                   /* name of queue to delete */
       strlen(cmd_queue),           /* length of queue name    */
       vm_ipc_plevel                /* scope of queue          */
    );
```

```
   if (rc != vm_rc_success)
   {
      printf ("QueueDelete failed with ");
      printf ("Return code = %d. Reason code = %d.\n", rc, re);
      printf ("Trying to delete %s.\n", cmd_queue);
      return (8);
   }

   return;
}

/*

   Process the REVERSE command.

   Receive from a queue, the data entered by the user.  Transpose
   this data, then display it on the terminal.

   The format of the incoming message is as follows:

   key user_data

   where key is in the following format:  flag command_name
         and flag is either CONT (continue) or QUIT
   and   user_data is the data to be displayed

*/

#include <stdio.h>
#include "vmcmt.h"

#define LINESIZE 131   /* Maximum length of user input + 1 for NULL */
#define FLAGSIZE 5     /* Size of continue/quit flag + 1 for NULL   */
#define CMDNUM 3       /* Total number of commands defined          */
#define REVERSE 2      /* Position of REVERSE command               */

#pragma linkage (rev_thd,OS)
int rev_thd ( )

{

   extern int cmd_q_han;            /* Handle of command queue      */
   extern char *cmd_list[CMDNUM];   /* List of defined commands     */

   int rc, re, index, out_index;

   char match_key[20], in_message[LINESIZE + FLAGSIZE], sender_UID[8],
        out_message[LINESIZE], compkey[FLAGSIZE+1];
   int message_length, key_offset, key_length, sender_PID, reply_token;

   char cont[FLAGSIZE+1] = { "CONT \0" };
   char done[FLAGSIZE] = { "DONE \0" };

   /*
      Set up the key for our command.
   */

   strcpy (match_key, "*");
   strcat (match_key, cmd_list[REVERSE]);

   /*
      Receive commands until the main program says to quit.
   */

   strcpy ( compkey, cont );
   while ( strcmp( compkey, cont ) == 0 )
   {

      /*
         Read a single command, waiting if one is not currently on
         the queue.
      */

      QueueReceiveBlock
      (
        &rc,                  /* return code                            */
        &re,                  /* reason code                            */
        cmd_q_han,            /* queue handle of command queue          */
        match_key,            /* key to match on                        */
        strlen(match_key),    /* length of key                          */
        0,                    /* timeout period                         */
```

```
         in_message,           /* receive message buffer             */
         sizeof(in_message),   /* size of message buffer             */
         &message_length,      /* length of message received         */
         &key_offset,          /* offset of key in message           */
         &key_length,          /* length of key                      */
         sender_UID,           /* sender user ID                     */
         &sender_PID,          /* sender process ID                  */
         &reply_token          /* reply token                        */
       );

       /*
          Check the flag to see if this is a real command, or quit.
       */

       memcpy( compkey, in_message, FLAGSIZE );
       if ( strcmp( compkey, cont ) == 0 )
       {

          /*
             Pick off the command, and transpose the rest of the string,
             working backward on the input string.
          */

          out_index = 0;
          for (index = message_length - 1; index >= key_length; index--)
          {
             out_message[out_index] = in_message[index];
             out_index += 1;
          }
          out_message[out_index] = '\0';
          printf ("%s\n", out_message);

          /*
             Tell the primary that we have completed our work.
          */

          QueueReply
          (
            &rc,               /* return code                        */
            &re,               /* reason code                        */
            reply_token,       /* reply token                        */
            done,              /* message                            */
            strlen(done),      /* length of message                  */
            0,                 /* key offset                         */
            strlen(done)       /* entire message is the key          */
          );
       }
    }

    /*
       The main program has indicated that it is time to quit.
    */

    return;

}

/*

   Process the DUPLICATE and REPEAT n commands.

   Receive from a queue, the data entered by the user.  If it is
   DUPLICATE, simply display the data on the user's terminal.
   If it is REPEAT, verify that the user has requested between 0 and
   9 copies of the data.  If so, display the data the specified
   number of times on the user's terminal.

   The format of the incoming message is as follows:

   key user_data

   where key is in the following format:  flag command_name
         and flag is either CONT (continue) or QUIT
   and   user_data is the data to be displayed
*/

#include <stdio.h>
#include "vmcmt.h"

#define LINESIZE 131   /* Maximum length of user input + 1 for NULL */
#define FLAGSIZE 5     /* Size of continue/quit flag + 1 for NULL   */
#define CMDNUM 3       /* Total number of commands defined          */
```

```
#define DUPLICATE 0    /* Position of DUPLICATE command             */
#define REPEAT 1       /* Position of REPEAT command                */

#pragma linkage (rep_thd,OS)
 rep_thd ( cmd_name )
   char *cmd_name;     /* Name of command being processed           */

{

   extern int cmd_q_han;            /* Handle of command queue      */
   extern char *cmd_list[CMDNUM];   /* List of defined commands     */

   int rc, re, rep_count, index;

   char match_key[20], in_message[LINESIZE + FLAGSIZE], sender_UID[8],
        compkey[FLAGSIZE + 1];
   int message_length, key_offset, key_length,
       sender_PID, reply_token;

   char cont[FLAGSIZE + 1] = { "CONT \0" };
   char done[FLAGSIZE] = { "DONE \0" };
   char *data_ptr, *end_ptr;

   /*
      Initialize the key for our command.
   */

   strcpy (match_key, "*");
   strcat (match_key, cmd_name);

   /*
      Receive commands until the main program says to quit.
   */

   strcpy ( compkey, cont );
   while ( strcmp( compkey, cont ) == 0 )
   {

      /*
         Read a single command, waiting if one is not currently on
         the queue.
      */

      QueueReceiveBlock
      (
        &rc,                  /* return code                          */
        &re,                  /* reason code                          */
        cmd_q_han,            /* queue handle of command queue        */
        match_key,            /* key to match on                      */
        strlen(match_key),    /* length of key                        */
        0,                    /* timeout period                       */
        in_message,           /* receive message buffer               */
        sizeof(in_message),   /* size of message buffer               */
        &message_length,      /* length of message received           */
        &key_offset,          /* offset of key in message             */
        &key_length,          /* length of key                        */
        sender_UID,           /* sender user id                       */
        &sender_PID,          /* sender process id                    */
        &reply_token          /* reply token                          */
      );

      /*
         Check the flag to see if this is a real command, or quit.
      */

      memcpy( compkey, in_message, FLAGSIZE );
      if ( strcmp( compkey, cont ) == 0 )
      {

         /*
            This is a real command.  Initialize pointers to the user
            data (this will point to the space after the command name)
            and the end of the data (make sure this is NULL).
         */

         data_ptr = in_message + key_length;
         end_ptr = in_message + message_length;
         *end_ptr = '\0';

         /*
            If the command is REPEAT (only need to check if the first
            letter is R), find out how many times to display the
```

```
            data, and only display the data that follows this number.
         */

         if ( in_message[FLAGSIZE] == cmd_list[REPEAT][0] )
         {

            /*
               Scan the data after the command name until we find
               a non-space character, or reach the end of the data.
            */

            while ( isspace( (int) *data_ptr ) != 0 &&
                    data_ptr < end_ptr )
               data_ptr += 1;

            /*
               See if this is a decimal digit.
            */

            if ( isdigit( (int) *data_ptr ) != 0  )

               /*
                  Yes, this is a valid decimal digit.
                  If you are not at the end of the data, the next
                  character must be some type of space.
               */

               if ( data_ptr < ( end_ptr - 1 ) &&
                    isspace( (int) *(data_ptr + 1) ) == 0 )
               {
                  rep_count = 0;
                  printf ("Must specify a number between 0 and 9.\n");
               }
               else
               {

                  /*
                     Convert from char to integer, then move past it.
                  */

                  rep_count = *data_ptr - '0';
                  data_ptr += 1;
               }
            else
            {
               rep_count = 0;
               printf ("Must specify a number between 0 and 9.\n");
            }
         }
         else
            rep_count = 1;  /* Command is DUPLICATE */

         /*
            Display the data the appropriate number of times.
            If rep_count is 0, do not display anything.
         */

         for (index = 1; index <= rep_count; index++)
            printf ("%s\n", data_ptr);

         /*
            Tell the primary that we have completed our work.
         */

         QueueReply
         (
           &rc,               /* return code                        */
           &re,               /* reason code                        */
           reply_token,       /* reply token                        */
           done,              /* message                            */
           strlen(done),      /* length of message                  */
           0,                 /* key offset                         */
           strlen(done)       /* entire message is the key          */
         );
      }
   }
/*
   The main program has indicated that it is time to quit.
*/
```

```
return;
}
```

Appendix E. Supplementary Information on System Defined Events

This appendix provides additional information about system events.

System Event Characteristics

The following table identifies the characteristics used when system events are defined. When an event is signaled, the signaler may also include data as part of the signal. Additionally that data may also contain a key to help with selectively handling signals.

Table 30. System Event Characteristics

Name	Scope	Signal delivery	Signaler treat-ment	Loose signal limit	Signal timeout period	Signal Data?	Key with Signal?	Notes
VMACCOUNT	Session	Broadcast	Async	-1	0	Yes	Yes	Must enable with AccountControl
VMCONINPUT	Session	Broadcast	Async	0	0	No	No	
VMCON1ECB	Session	Broadcast	Async	0	0	No	No	
VMCPIC	Session	Broadcast	Async	0	0	Yes	Yes	
VMERROR	Process	LIFO	Sync Process	0	0	Yes	Yes	
VMERRORCHILD	Process	LIFO	Sync Process	0	0	Yes	Yes	
VMIPC	Process	Broadcast	Async	0	0	Yes	Yes	Must enable with QueueSignal-Events
VMPOSGNL	Process	Broadcast	Async	0	0	Yes	Yes	Created only for POSIX processes
VMPROCESSEND	Session	Broadcast	Sync Thread	0	0	Yes	Yes	
VMSFSASYNC	Session	Broadcast	Sync Thread	0	0	Yes	Yes	
VMSOCKET	Session	Broadcast	Async	1	0	Yes	No	Defined only while RXSOCKET is active
VMTIMECHANGE	Session	Broadcast	Async	0	0	Yes	Yes	
VMTIMER	Session	Broadcast	Async	-1	0	Yes	Yes	
VMTRACE	Session	Broadcast	Async	50	0	Yes	Yes	Also see the CMS TRACECTL command

© Copyright IBM Corp. 1992, 2023

VMCONINPUT and VMCON1ECB

Two of the system events pertain to console activity:

- **VMCONINPUT** is signaled when an unsolicited attention is received at the console.
- **VMCON1ECB** is signaled when input is available at the console.

For many applications VMCONINPUT and VMCON1ECB may be used interchangeably. Every time an unsolicited attention interrupt is received on the console, VMCONINPUT is signaled. However, the data represented by the signal may be consumed by CMS before the application has a chance to read it (for example, an immediate command was entered).

VMCON1ECB is signaled less frequently than VMCONINPUT. VMCON1ECB is signaled only when data is available for the application to read (it is not signaled for immediate commands, for example). When VMCON1ECB is signaled, the application is guaranteed that if it issues a read the virtual machine will not end up in VM READ.

VMSOCKET Signal Data

VMSOCKET is signaled by REXX Sockets when a Select request completes. What normally would have been returned from a Select subfunction, namely a count of completed events followed by a list of the events, is provided as the signal data.

Most system events provide signal data. VMSOCKET is unique among system defined events in that it provides signal data but no key. Broadly speaking, with the system events, the key is used to provide a degree of uniqueness between events. Rather than a key, REXX Sockets allows a unique event name to be provided for each outstanding Select request.

When the socket Terminate function is called, if outstanding Select requests remain, they are signaled with a completed event count of zero.

For more information, see the *z/VM: REXX/VM Reference*.

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

© Copyright IBM Corp. 1992, 2023

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information may contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Programming Interface Information

This manual documents intended Programming Interfaces that allow the customer to write programs to obtain services of z/VM.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., in the United States and/or other countries. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on IBM Copyright and trademark information (https://www.ibm.com/legal/copytrade).

Adobe is either a registered trademark or trademark of Adobe Systems Incorporated in the United States, and/or other countries.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Terms and Conditions for Product Documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal Use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering’s use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see:

- The section entitled **IBM Websites** at IBM Privacy Statement (https://www.ibm.com/privacy)
- Cookies and Similar Technologies (https://www.ibm.com/privacy#Cookies_and_Similar_Technologies)

Bibliography

This topic lists the publications in the z/VM library. For abstracts of the z/VM publications, see *z/VM: General Information*.

Where to Get z/VM Information

The current z/VM product documentation is available in IBM Documentation - z/VM (https://www.ibm.com/docs/en/zvm).

z/VM Base Library

Overview

- *z/VM: License Information*, GI13-4377
- *z/VM: General Information*, GC24-6286

Installation, Migration, and Service

- *z/VM: Installation Guide*, GC24-6292
- *z/VM: Migration Guide*, GC24-6294
- *z/VM: Service Guide*, GC24-6325
- *z/VM: VMSES/E Introduction and Reference*, GC24-6336

Planning and Administration

- *z/VM: CMS File Pool Planning, Administration, and Operation*, SC24-6261
- *z/VM: CMS Planning and Administration*, SC24-6264
- *z/VM: Connectivity*, SC24-6267
- *z/VM: CP Planning and Administration*, SC24-6271
- *z/VM: Getting Started with Linux on IBM Z*, SC24-6287
- *z/VM: Group Control System*, SC24-6289
- *z/VM: I/O Configuration*, SC24-6291
- *z/VM: Running Guest Operating Systems*, SC24-6321
- *z/VM: Saved Segments Planning and Administration*, SC24-6322
- *z/VM: Secure Configuration Guide*, SC24-6323

Customization and Tuning

- *z/VM: CP Exit Customization*, SC24-6269
- *z/VM: Performance*, SC24-6301

Operation and Use

- *z/VM: CMS Commands and Utilities Reference*, SC24-6260
- *z/VM: CMS Primer*, SC24-6265
- *z/VM: CMS User's Guide*, SC24-6266
- *z/VM: CP Commands and Utilities Reference*, SC24-6268

© Copyright IBM Corp. 1992, 2023

- *z/VM: System Operation*, SC24-6326
- *z/VM: Virtual Machine Operation*, SC24-6334
- *z/VM: XEDIT Commands and Macros Reference*, SC24-6337
- *z/VM: XEDIT User's Guide*, SC24-6338

Application Programming

- *z/VM: CMS Application Development Guide*, SC24-6256
- *z/VM: CMS Application Development Guide for Assembler*, SC24-6257
- *z/VM: CMS Application Multitasking*, SC24-6258
- *z/VM: CMS Callable Services Reference*, SC24-6259
- *z/VM: CMS Macros and Functions Reference*, SC24-6262
- *z/VM: CMS Pipelines User's Guide and Reference*, SC24-6252
- *z/VM: CP Programming Services*, SC24-6272
- *z/VM: CPI Communications User's Guide*, SC24-6273
- *z/VM: ESA/XC Principles of Operation*, SC24-6285
- *z/VM: Language Environment User's Guide*, SC24-6293
- *z/VM: OpenExtensions Advanced Application Programming Tools*, SC24-6295
- *z/VM: OpenExtensions Callable Services Reference*, SC24-6296
- *z/VM: OpenExtensions Commands Reference*, SC24-6297
- *z/VM: OpenExtensions POSIX Conformance Document*, GC24-6298
- *z/VM: OpenExtensions User's Guide*, SC24-6299
- *z/VM: Program Management Binder for CMS*, SC24-6304
- *z/VM: Reusable Server Kernel Programmer's Guide and Reference*, SC24-6313
- *z/VM: REXX/VM Reference*, SC24-6314
- *z/VM: REXX/VM User's Guide*, SC24-6315
- *z/VM: Systems Management Application Programming*, SC24-6327
- *z/VM: z/Architecture Extended Configuration (z/XC) Principles of Operation*, SC27-4940

Diagnosis

- *z/VM: CMS and REXX/VM Messages and Codes*, GC24-6255
- *z/VM: CP Messages and Codes*, GC24-6270
- *z/VM: Diagnosis Guide*, GC24-6280
- *z/VM: Dump Viewing Facility*, GC24-6284
- *z/VM: Other Components Messages and Codes*, GC24-6300
- *z/VM: VM Dump Tool*, GC24-6335

z/VM Facilities and Features

Data Facility Storage Management Subsystem for z/VM

- *z/VM: DFSMS/VM Customization*, SC24-6274
- *z/VM: DFSMS/VM Diagnosis Guide*, GC24-6275
- *z/VM: DFSMS/VM Messages and Codes*, GC24-6276
- *z/VM: DFSMS/VM Planning Guide*, SC24-6277

- *z/VM: DFSMS/VM Removable Media Services*, SC24-6278
- *z/VM: DFSMS/VM Storage Administration*, SC24-6279

Directory Maintenance Facility for z/VM

- *z/VM: Directory Maintenance Facility Commands Reference*, SC24-6281
- *z/VM: Directory Maintenance Facility Messages*, GC24-6282
- *z/VM: Directory Maintenance Facility Tailoring and Administration Guide*, SC24-6283

Open Systems Adapter

- Open Systems Adapter-Express Customer's Guide and Reference (https://www.ibm.com/support/pages/node/6019492), SA22-7935
- Open Systems Adapter-Express Integrated Console Controller User's Guide (https://www.ibm.com/support/pages/node/6019810), SC27-9003
- Open Systems Adapter-Express Integrated Console Controller 3215 Support (https://www.ibm.com/docs/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.ioa/ioa.htm), SA23-2247
- Open Systems Adapter/Support Facility on the Hardware Management Console (https://www.ibm.com/docs/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.ioa/ioa.htm), SC14-7580

Performance Toolkit for z/VM

- *z/VM: Performance Toolkit Guide*, SC24-6302
- *z/VM: Performance Toolkit Reference*, SC24-6303

RACF® Security Server for z/VM

- *z/VM: RACF Security Server Auditor's Guide*, SC24-6305
- *z/VM: RACF Security Server Command Language Reference*, SC24-6306
- *z/VM: RACF Security Server Diagnosis Guide*, GC24-6307
- *z/VM: RACF Security Server General User's Guide*, SC24-6308
- *z/VM: RACF Security Server Macros and Interfaces*, SC24-6309
- *z/VM: RACF Security Server Messages and Codes*, GC24-6310
- *z/VM: RACF Security Server Security Administrator's Guide*, SC24-6311
- *z/VM: RACF Security Server System Programmer's Guide*, SC24-6312
- *z/VM: Security Server RACROUTE Macro Reference*, SC24-6324

Remote Spooling Communications Subsystem Networking for z/VM

- *z/VM: RSCS Networking Diagnosis*, GC24-6316
- *z/VM: RSCS Networking Exit Customization*, SC24-6317
- *z/VM: RSCS Networking Messages and Codes*, GC24-6318
- *z/VM: RSCS Networking Operation and Use*, SC24-6319
- *z/VM: RSCS Networking Planning and Configuration*, SC24-6320

TCP/IP for z/VM

- *z/VM: TCP/IP Diagnosis Guide*, GC24-6328
- *z/VM: TCP/IP LDAP Administration Guide*, SC24-6329
- *z/VM: TCP/IP Messages and Codes*, GC24-6330

- *z/VM: TCP/IP Planning and Customization*, SC24-6331
- *z/VM: TCP/IP Programmer's Reference*, SC24-6332
- *z/VM: TCP/IP User's Guide*, SC24-6333

Prerequisite Products

Device Support Facilities

- Device Support Facilities (ICKDSF): User's Guide and Reference (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ickug00_v2r5.pdf), GC35-0033

Environmental Record Editing and Printing Program

- Environmental Record Editing and Printing Program (EREP): Reference (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ifc2000_v2r5.pdf), GC35-0152
- Environmental Record Editing and Printing Program (EREP): User's Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ifc1000_v2r5.pdf), GC35-0151

Related Products

XL C++ for z/VM

- *XL C/C++ for z/VM: Runtime Library Reference*, SC09-7624
- *XL C/C++ for z/VM: User's Guide*, SC09-7625

Additional Publications

XL C/C++ for z/VM: Runtime Library Reference, SC09-7624
XL C/C++ for z/VM: User's Guide, SC09-7625
z/OS: XL C/C++ Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/cbcpx01_v2r5.pdf), SC09-4765
z/OS: XL C/C++ User's Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/cbcux01_v2r5.pdf), SC09-4767

Index

Special Characters

#include statement 81
$QUEUES$ NAMES
    entry format and defaults 32
    sample 32
$SERVER$ NAMES 34
$SERVER$ NAMES (CMS private resource registration file)
    entries for network queues 35
    required format 35

A

abend services
    examples 72
    monitoring error events 68
abnormal termination (abend)
    a process 97
    of process 4
    of thread 4
AbnormalEnd routine 97
access a critical section protected by a mutex 49
access remote queues 34
access to local queues 34
AccountControl routine
    reference 99
    usage 65
AccountIdentify routine
    reference 102
    usage 65
accounting
    record format 65
    records, define 99
    services
        examples 66
        format of accounting record 65
        return and reason code values 298
address of CMS monitor data area, obtaining 171
advanced error recovery 70
API (application program interface)
    assembler
        binding files 82
        building a multitasking program 88
        calling multitasking functions 86
        general information 85
        outline of an application 87
    C
        binding files 81
        building an applmain() enabled program 84
        calling multitasking functions 85
        general information 83
        using the applmain() linkage 83
        using the POSIX entry linkage 83
    CMS and OS/2 process management 8
    function descriptions 95
    general considerations 92
    programming language binding files 81
API (application program interface) *(continued)*
    REXX/VM
        binding files 82
        calling multitasking functions 89
        general information 89
        using binding files with REXX procedures 91
    VMMTLIB callable services library 81
APPC/VM (Advanced Program-to-Program Communications/VM)
    as the carrier for network-level queues
        considerations 34
    carrier considerations 34
    IPC carrier line flows 315
    IPC carrier request flows 316
    IPC carrier response flows 317
    IPC carrier structure 316
application entry point, assembler 85
application entry point, C 83
application information, user 76
application program interface (API)
    assembler
        binding files 82
        building a multitasking program 88
        calling multitasking functions 86
        general information 85
        outline of an application 87
    C
        binding files 81
        building an applmain() enabled program 84
        calling multitasking functions 85
        general information 83
        using the applmain() linkage 83
        using the POSIX entry linkage 83
    CMS and OS/2 process management 8
    function descriptions 95
    general considerations 92
    programming language binding files 81
    REXX/VM
        binding files 82
        calling multitasking functions 89
        general information 89
        using binding files with REXX procedures 91
    VMMTLIB callable services library 81
APPLMAIN 85
applmain() 83
assembler application outline 87
assembler call to ThreadYield example 86
assembler example 87
assembler macros provided 82
asynchronous event handler, defining 167
authorization rules of queues 35
automatic queue program startup, considerations for 35

B

basic semaphore processing 49
binding files, programming language

binding files, programming language *(continued)*
- assembler programming 86
- list 81
- special considerations for REXX programs 91

building a message key and match key 30
building a parameter list 86

C

C call to ThreadYield, example of 85
C language
- header files provided 81
- programming restrictions 85

c89 command 83
CALL macro 87
callable services library, VMMTLIB 81
callable timer facility 59
calling CMS functions 86
cancel
- a timer 260, 263
- all timers 262

carrier request block (QCRB)
- definition 309
- header description 311
- IPC0 (kernel request to carrier) 313
- IPC1 (carrier response to kernel) 313
- IPC2 (carrier request to kernel) 314
- IPC3 (kernel response to carrier) 314

carrier request to kernel (IPC2 QCRBs) 314
carrier response to kernel (IPC1 QCRBs) 313
carrier skeleton, IPC (carrier-initiated requests) 308
carrier skeleton, IPC (kernel-initiated requests) 307
carriers
- interface between CMS and 307
- line flows, APPC/VM 315
- multiple 307
- request flows 316
- response flows 317
- single 307
- structure 316
- threads 307

categories of queue functions 27
CHAR notation in parameter descriptions 95
characters, wildcard 30, 31
close an open queue 190
close and delete queues example 44
CMS (Conversational Monitor System)
- API restrictions and considerations 92
- CALL macro 87
- communications directory 34
- comparison with OS/2 8
- monitor data area address, obtaining 171
- Shared File System (SFS) 288
- trace entry format 76
- trace information structure 75
- trace table 75

CMS multitasking
- abend services 67
- accounting services 65
- and OS/2 process management and related APIs 8
- as a run-time extension 1
- calling functions 86
- comparison with OS/2 8
- creating application
    - using assembler 85
    - using C 83
    - using REXX 89
- event services 17
- example program, C 319
- initialization entry routine, VMSTART
    - creating a process 12
    - included in assembler multitasking application module 88
    - included in C multitasking application module 84
    - invoked by REXX execs 89
- interface between carriers and 307
- interprocess communication 27
- introduction 1
- kernel 307
- list of functions 5
- monitor data 79
- multiprocessor configuration control 57
- overview 1
- process management 11
- restrictions 85
- synchronization 47
- system exits 277
- trace entry format 76
- trace services 75
- trace table 75
- using HELP for 95
- using OpenExtensions services 291

CMS private resource registration file ($SERVER$ NAMES) 34
CMSIUCV 14
Common Programming Interface (CPI) Communications
- VMCPIC 17

communication suggestions for server writers 287
communication trace record formats 303
communications directory, CMS 34
comparison between CMS and OS/2 8
condition variable
- definition 47

CondVarCreate routine 104
CondVarDelete routine 106
CondVarGetHandle routine 108
CondVarSignal routine 110
CondVarWait routine 111
considerations for APPC/VM carrier 34
considerations for automatic queue program startup 35
considerations for local access of a queue 32
considerations for remote access of a queue 32
constructing message and match keys 30
context switching 283
conventions, system exit linkage 277
COPY files provided, REXX 82
CP DASD Block I/O system service 288
CPI Communications
- VMCPIC 17

CPU
- guidelines 289
- virtual 57

create
- a condition variable 104
- a queue 191
- a semaphore 218
- a thread 227

create *(continued)*
a virtual processor 274
an event definition 128
threads 11, 281
create and open queues example 38
critical section
accessing a critical section protected by a mutex 49
definition 47
CSL routines
AccountControl 99
AccountIdentify 102
CondVarGetHandle 108
CondVarSignal 110
CondVarWait 111

D

DASD Block I/O system service, CP 288
data management suggestions for server writers 288
DateTimeGet routine 113
DateTimeSubtract routine
reference 115
usage 60
decrement thread suspend count 243
define
asynchronous event handler 167
event handling environment 139
define virtual CPUs 57
defining an event with EventCreate 17
definition of an event with EventCreate 128
definition of queue 27
delaying a thread 231
delete
a condition variable 106
a process 281
a queue 193
a semaphore 220
an event definition 131
an event monitor 142
threads 232, 282
delete and close queues example 44
description of services 4
differences between CMS and OS/2 process models 8
difficulties in constructing keys 30
directory
CMS communications 34
disable
monitor activation 135
specific monitors 144
dispatch trace record formats 304
dispatching classes of threads 11

E

enable
monitor activation 135
specific monitors 144
enable access to local queues 34
entry point
USRRTHD 278
USRSINIT 277
USRTINIT 278
USRTTERM 278
environment exits, programming language 279
environment manager 279
error
data format 67
events, monitoring 68
format of abend error data 67
in high-level language environment 69
recovery 68
retry routines 69
error event data 67
error event handlers in a process 4
escape character 30, 31
event
based model 14
definition 17
definition with EventCreate 128
examples 22
key 18
management
examples 22
monitoring error 68
monitors
processing 19
name 17
signaling 18, 20
system 17
event management services
CMS trace table 75
event definition 17
event monitor processing 19
event monitors 19
event signal processing 20
event signaling 18
examples 22
interaction with trace services 75, 76
interactions with queues 38
overview of functions 20
return and reason code values 296
event-based structure 14
EventCreate routine 128
EventDelete routine 131
EventDiscard routine 133
EventEnable routine 135
EventModify routine 137
EventMonitorCreate routine
maintaining trace services 75
reference 139
usage 19
EventMonitorDelete routine 142
EventMonitorEnable routine 144
EventMonitorQuery routine 146
EventMonitorReset routine 150
EventMonitorSelect routine 152
EventQuery routine 154
EventQueryAll routine 157
EventRetrieve routine
maintaining trace services 76
reference 159
events, system
VMACCOUNT
characteristics 331
using 65
VMCON1ECB
characteristics 331

events, system *(continued)*
VMCON1ECB *(continued)*
contrasted with VMCONINPUT 332
VMCONINPUT
characteristics 331
contrasted with VMCON1ECB 332
VMCPIC
characteristics 331
VMERROR
characteristics 331
using 67
VMERRORCHILD
characteristics 331
using 67
VMIPC
characteristics 331
using 216
VMPOSGNL
characteristics 331
using 291
VMPROCESSEND
characteristics 331
using 12
VMSOCKET
characteristics 331
signal data 332
VMTIMECHANGE
characteristics 331
using 59
VMTIMER
characteristics 331
using 59
VMTRACE
characteristics 331
using 75
EventSelect routine 161
EventSignal routine
reference 163
usage 18
EventTest routine
reference 165
usage 19
EventTrap routine
reference 167
usage 19
EventWait routine
maintaining trace services 75
reference 169
usage 19
exact match keys 29
example
abend services 72
accounting services 66
assembler call to ThreadYield 86
C call to ThreadYield 85
event management
broadcast signals 23
EventTest 23
EventTrap 22
EventWait 22
loose and bound signal limits 24
process level events 25
sequentially propagated signals 24
interprocess communication
example *(continued)*
interprocess communication *(continued)*
closing and deleting queues 44
creating and opening queues 38
rendezvous 40
sharing a request queue 42
simple message transmission 39
using replies 41
using service queues 43
network-level queues 44
process management event-based structure 14
process management queue-based structure 13
program, C 319
REXX exec call to TraceControl 90
synchronization
accessing a critical section protected by a mutex 49
basic semaphore processing 49
going beyond WAIT/POST using semaphores 50
multiple waiters using a semaphore 50
producer and consumer processes 52
shared monitor 53
timer services 59
exit
system
programming language environment 279
root process 278
session initialization 277
thread initialization 278
thread termination 278
exit linkage conventions, system 277
export
queues 28
export level search order 29

F

find blocked threads 185
find suspended threads 188
format of accounting records 65
format of CMS trace entries 76
format of trace records 303
FORTRAN programs 58
fuzzy match keys 29

G

general guidelines for server writers 288
getting the value of a semaphore 223
going beyond WAIT/POST using semaphores 50
guidelines for defining virtual CPUs 57
guidelines for server writers 287, 288

H

handle, queue 28
HELP, online 95
HNDEXT exit 288
HNDIUCV exit 288

I

identify
communication carrier 195

implicit process creation 12
information, user application 76
initialization entry routine, VMSTART
 creating a process 12
 included in assembler multitasking application module 88
 included in C multitasking application module 84
 invoked by REXX execs 89
initialization exit, session 277
initialization exit, thread 278
INT notation in parameter descriptions 95
interaction of sessions, processes, and threads 2
interactions between queues and event services 38
interactions between queues and process management 38
interface between CMS and carriers 307
interface definition of remote IPC support 309
interprocess communication
 authorization 35
 concepts 3
 examples 38
 export level search order 29
 interactions with event services 38
 interactions with process management 38
 keys, message and match 29
 network-level queues 31
 primary queue 29
 properties of queues 28
 queue definition 27
 queue names 28
 queue operation 27
 replies 36
 return and reason code values 298
 service queues 37
 timeouts 37
interrupt handling suggestions for server writers 287
introduction 1
invocation with APPLMAIN 85
invocation with applmain() 83
invoking queue functions 27
IPC carrier skeleton (carrier-initiated requests) 308
IPC carrier skeleton (kernel-initiated requests) 307
IPC carrier threads 307
IPC service queue, network 308
IPC support, remote
 APPC/VM carrier line flows 315
 interface definition 309
 overview 307
 skeleton
  carrier-initiated requests 308
  kernel-initiated requests 307
IPC0 QCRBs (kernel request to carrier) 313
IPC1 QCRBs (carrier response to kernel) 313
IPC2 QCRBs (carrier request to kernel) 314
IPC3 QCRBs (kernel response to carrier) 314
IUCV (Inter-User Communications Vehicle) 14, 287

K

kernel 307
kernel request to carrier (IPC0 QCRBs) 313
kernel response to carrier (IPC3 QCRBs) 314
key
 definition 18
 match 29
key *(continued)*
 message 29
 tips on constructing 30
 types 29
key descriptor of message prefix 27

L

language adapter trace record formats 305
language environment exits, programming 279
LIFO propagation 68
linkage conventions, system exit 277
list of CMS functions for multitasking 5
local access considerations for queues 32
local queues
 enabling access to 34
 local access considerations 32
loose signals 18

M

macros provided, assembler 82
management, process
 APIs, CMS and OS/2 8
 creating threads 11
 dispatching classes 11
 examples 13
 process termination 12
 return and reason code values 295
manager, environment 279
managing events 17
match keys
 fuzzy and exact 29
 tips on constructing 30
match-all match key (*) 30
message
 definition 27
 key
  tips on constructing 30
 prefix 27
 text 27
message transmission example 39
miscellaneous trace record formats 306
model, event-based 14
model, process
 concepts 1
 differences between CMS and OS/2 8
 resemblance to OS/2 2
model, queue-based 13
modify
 characteristics of event definition 137
monitor
 error events 68
 functions provided by CMS 17
monitor data area address, CMS, obtaining 171
MonitorBufferGet routine 171
monitoring of events 19
MP-capable 87
multiple carriers 307
multiple waiters 50
multiprocessor configuration control
 guidelines for defining virtual CPUs 57
multiprocessor-capable 87

multitasking routines
AccountControl 99
AccountIdentify 102
CondVarGetHandle 108
CondVarSignal 110
CondVarWait 111
multitasking, CMS
abend services 67
accounting services 65
and OS/2 process management and related APIs 8
as a run-time extension 1
calling functions 86
comparison with OS/2 8
creating application
using assembler 85
using C 83
using REXX 89
event services 17
example program, C 319
initialization entry routine, VMSTART
creating a process 12
included in assembler multitasking application module 88
included in C multitasking application module 84
invoked by REXX execs 89
interface between carriers and 307
interprocess communication 27
introduction 1
kernel 307
list of functions 5
monitor data 79
multiprocessor configuration control 57
overview 1
process management 11
restrictions 85
synchronization 47
system exits 277
trace entry format 76
trace services 75
trace table 75
using HELP for 95
using OpenExtensions services 291
mutex
accessing a critical section protected by a mutex 49
definition 47
MutexAcquire routine 173
MutexCreate routine 175
MutexDelete routine 177
MutexGetHandle routine 179
MutexRelease routine 181

N

name of events 17
name scopes, queue 28
names of queues 28
network IPC service queue 308
network-level queues
definition 28
example of how to set up 44
local access considerations 32
overview 31
remote access considerations 32
normal termination
normal termination *(continued)*
of process 4
of thread 4
notation used in parameter descriptions 95

O

obtain the thread ID and process ID 234
occurrence of an event 17
online HELP Facility, using 95
open a queue 199
open and create queues example 38
operation of queues 27
OS/2
comparison with CMS 8
relationship to CMS multitasking process model 2
table of process management APIs 8
outline of an assembler application 87
overview of CMS multitasking 1

P

parallel processing 57
parameter list
building 86
POSIX(OFF) mode 83
POSIX(ON) mode 83
preemption, definition 3
prefix, message
key descriptor 27
sender ID 27
text length 27
primary queue 3, 29
priority
thread 11
process
and threads 2
creating 3
creation 3
definition 2
deletion 281
end event name (VMPROCESSEND) 12
exit, root 278
interaction with threads and sessions 2
management
APIs, CMS and OS/2 8
creating threads 11
dispatching classes 11
event-based structure example 14
examples 13
interactions with queues 38
overview 11
process termination 12
queue-based structure example 13
return and reason code values 295
trace record formats 305
model
concepts 1
differences between CMS and OS/2 8
resemblance to OS/2 2
priority 3
state snapshot 182
termination

process *(continued)*
termination *(continued)*
abnormal 4
description 12
normal 4
process creation, implicit 12
process-level queues 28
ProcessCheckPoint routine 182
ProcessGetID routine 184
processing event signal 20
ProcessQueryBlocked routine 185
ProcessQuerySuspended routine 188
producer and consumer processes 52
program, C, example 319
programming language binding files
assembler programming 86
list 81
special considerations for REXX programs 91
programming language environment exits 279
programming restrictions, C 85
properties of queues 28

Q

QCRB (carrier request block)
definition 309
header description 311
IPC0 (kernel request to carrier) 313
IPC1 (carrier response to kernel) 313
IPC2 (carrier request to kernel) 314
IPC3 (kernel response to carrier) 314
query
a timer 265, 267
all event names and monitor tokens 157
an event definition 154
blocked threads 185
dispatch class of a thread 235
entry point of a thread 237
event monitor information 146
parameter list of a thread 238
priority of a thread 240
suspend count of a thread 241
suspended threads 188
time and date 113
waiting message count 201
querying thread user data 242
queue
accessing remote queues 34
authorization rules 35
based model 13
communication carrier, identifying 195
definition 27
enabling access to local queues 34
events, signalling 216
example of a rendezvous 40
example of closing and deleting 44
example of creating and opening 38
example of replying 41
example of service queues 43
example of sharing a request queue 42
example of simple message transmission 39
function categories 27
handle 28
interactions with event services 38
queue *(continued)*
interactions with process management 38
interprocess communication 3, 27
message 27
name scopes 28
names 28
network IPC service 308
network-level
APPC/VM carrier considerations 34
local access considerations 32
remote access considerations 32
operation 27
primary 3, 29
properties 28
replies 36
reply 36
service 37
service, identifying 197
timeouts 37
queue program startup, automatic 35
queue-based structure 13
QueueClose routine 190
QueueCreate routine 191
QueueDelete routine 193
QueueIdentifyCarrier routine
reference 195
use in remote IPC support 307
QueueIdentifyService routine
reference 197
use in remote IPC support 307
QueueOpen routine 199
QueueQuery routine 201
QueueReceiveBlock routine
reference 203
timeout value 37
use in remote IPC support 307
QueueReceiveImmed routine
reference 206
use in remote IPC support 307
QueueReply routine
example 41
reference 208
usage 37
use in remote IPC support 307
QueueSend routine 210
QueueSendBlock routine
reference 212
timeout value 37
QueueSendReply routine
reference 214
usage 36
usage information 37
QueueSignalEvents routine
reference 216
usage 38

R

reason code values
accounting services 298
CMS monitor data 301
event services 296
interprocess communication 298
process management 295

reason code values (continued)
- timer services 299
- trace services 298
- VCPU services 300

receive a message from a queue 203, 206
record formats, trace 303
recovery, error 68
reentrant code 86
reinitialize a semaphore's value 224
remote access considerations for queues 32
remote IPC support
- APPC/VM carrier line flows 315
- interface definition 309
- overview 307
- skeleton
  - carrier-initiated requests 308
  - kernel-initiated requests 307

remote queues
- accessing 34

rendezvous queue example 40
reply
- to a message 208

reply queue 36
request data record 317
request flows of APPC/VM carrier 316
request header record 316
request special virtual CPU dispatching 276
reset
- a monitor 20
- an event monitor 150

resource ownership 2
response flows of APPC/VM carrier 317
response header record 317
restrictions
- C programming 85
- CMS API 92

resume
- threads 243

retrieve
- data from an event signal 159

retry routines 69
return code values
- accounting services 298
- CMS monitor data 301
- event services 296
- interprocess communication 298
- process management 295
- timer services 299
- trace services 298
- VCPU services 300

REXX/VM interpreter
- COPY files provided 82
- example of call to TraceControl 90
- supplementary information for programmers 89

root process
- exit 278

routines
- AbnormalEnd 97
- AccountControl 99
- AccountIdentify 102
- CondVarCreate 104
- CondVarDelete 106
- CondVarGetHandle 108
- CondVarSignal 110

routines (continued)
- CondVarWait 111
- DateTimeGet 113
- DateTimeSubtract 115
- EventCreate 128
- EventDelete 131
- EventDiscard 133
- EventEnable 135
- EventModify 137
- EventMonitorCreate 139
- EventMonitorDelete 142
- EventMonitorEnable 144
- EventMonitorQuery 146
- EventMonitorReset 150
- EventMonitorSelect 152
- EventQuery 154
- EventQueryAll 157
- EventRetrieve 159
- EventSelect 161
- EventSignal 163
- EventTest 165
- EventTrap 167, 169
- MonitorBufferGet 171
- MutexAcquire 173
- MutexCreate 175
- MutexDelete 177
- MutexGetHandle 179
- MutexRelease 181
- ProcessCheckPoint 182
- ProcessGetID 184
- ProcessQueryBlocked 185
- ProcessQuerySuspended 188
- QueueClose 190
- QueueCreate 191
- QueueDelete 193
- QueueIdentifyCarrier 195
- QueueIdentifyService 197
- QueueOpen 199
- QueueQuery 201
- QueueReceiveBlock 203
- QueueReceiveImmed 206
- QueueReply 208
- QueueSend 210
- QueueSendBlock 212
- QueueSendReply 214
- QueueSignalEvents 216
- SemCreate 218
- SemDelete 220
- SemGetHandle 221
- SemQueryValue 223
- SemReInit 224
- SemSignal 225
- SemWait 226
- ThreadCreate 227
- ThreadDelay 231
- ThreadDelete 232
- ThreadGetID 234
- ThreadQueryDispatchClass 235
- ThreadQueryEntryPoint 237
- ThreadQueryParameterList 238
- ThreadQueryPriority 240
- ThreadQuerySuspendCount 241
- ThreadQueryUserData 242
- ThreadResume 243

routines *(continued)*
ThreadSetDispatchClass 244
ThreadSetPriority 246
ThreadSetUserData 248
ThreadSuspend 249
ThreadYield 251
TimerStartInt 253
TimerStartMicros 256
TimerStartTOD 258
TimerStop 260
TimerStopAll 262
TimerStopMicros 263
TimerTest 265
TimerTestMicros 267
TraceControl 269
TraceSignal 272
VCPUCreate 274
VCPUSelect 276
rules for authorization of queues 35
run a routine in context 282
run-time extension, CMS multitasking as a 1

S

SAA CPI Communications
VMCPIC 17
sample program, C 319
scopes, queue name 28
search order, export level 29
section, critical 47
semaphore
basic processing 49
definition 47
going beyond WAIT/POST using semaphores 50
multiple waiters 50
SemCreate routine 218
SemDelete routine 220
SemGetHandle routine 221
SemQueryValue routine 223
SemReInit routine 224
SemSignal routine 225
SemWait routine 226
send
message and request reply 214
message to queue 210, 212
sender ID of message prefix 27
serially-reusable code 86
server writers' suggestions 287
service IDs 37
service queue, network IPC 308
service queues 37, 307
services
abend 67
accounting 65
description 4
monitor data 79
trace 75
session
definition 2
interaction with threads and processes 2
session initialization exit 277
session-level queues 28
setting thread user data 248
Shared File System (SFS), CMS 288
shared monitor 53
sharing a request queue example 42
signal events 18
signaling of events 20
signalling a semaphore 225
signalling the occurrence of an event 163
signals, event 18
simple message transmission example 39
skeleton
carrier-initiated requests 308
kernel-initiated requests 307
start
interval timer 253, 256
TOD timer 258
stop
a timer 260, 263
all timers 262
structure of APPC/VM carrier 316
suggestions for server writers 287
supplementary information for REXX/VM programmers 89
suspend
threads 249
switching context 283
synchronization
examples
accessing a critical section protected by a mutex 49
basic semaphore processing 49
going beyond WAIT/POST using semaphores 50
multiple waiters using a semaphore 50
producer and consumer processes 52
shared monitor 53
mutex definition 47
synchronization trace record formats 305
system event characteristics 331
system events
VMACCOUNT
characteristics 331
using 65
VMCON1ECB
characteristics 331
contrasted with VMCONINPUT 332
VMCONINPUT
characteristics 331
contrasted with VMCON1ECB 332
VMCPIC
characteristics 331
VMERROR
characteristics 331
using 67
VMERRORCHILD
characteristics 331
using 67
VMIPC
characteristics 331
using 216
VMPOSGNL
characteristics 331
using 291
VMPROCESSEND
characteristics 331
using 12
VMSOCKET
characteristics 331
signal data 332

system events *(continued)*
VMTIMECHANGE
characteristics 331
using 59
VMTIMER
characteristics 331
using 59
VMTRACE
characteristics 331
using 75
system exit linkage conventions 277
system exits
programming language environment 279
root process 278
session initialization 277
thread initialization 278
thread termination 278
system service , CP DASD Block I/O 288

T

table of CMS and OS/2 process management APIs 8
table of CMS functions for multitasking 5
table, trace 75
termination
abnormal
of process 4
normal
of process 4
of thread 4
termination exit, thread 278
termination, process 12
testing for occurrence of events 165
text length of message prefix 27
thread
and processes 2
at creation 3
carrier 307
creating 11, 281
definition 2
delay 231
deleting 282
dispatching classes of 11
initialization exit 278
interaction with sessions and processes 2
priority 3, 11
suspend count 243
synchronization 47
termination exit 278
termination of 4
ThreadCreate routine
reference 227
usage 11
ThreadDelay routine 231
ThreadDelete routine 232
ThreadGetID routine 234
ThreadQueryDispatchClass routine 235
ThreadQueryEntryPoint routine 237
ThreadQueryParameterList routine 238
ThreadQueryPriority routine 240
ThreadQuerySuspendCount routine 241
ThreadQueryUserData routine 242
ThreadResume routine 243
threads sharing a request queue example 42
ThreadSetDispatchClass routine 244
ThreadSetPriority routine 246
ThreadSetUserData routine 248
ThreadSuspend routine 249
ThreadYield routine
example of assembler call 86
example of call from C 85
reference 251
time stamps, converting and manipulating
DateTimeSubtract routine description 115
examples 60
timeouts 37
timer services
examples 59
overview 59
return and reason code values 299
TimerStartInt routine 253
TimerStartMicros routine 256
TimerStartTOD routine 258
TimerStop routine 260
TimerStopAll routine 262
TimerStopMicros routine 263
TimerTest routine 265
TimerTestMicros routine 267
tips on constructing keys 30
trace entry format in CMS 76
trace header 76
trace information structure, CMS 75
trace record formats
communication 303
dispatch 304
language adapter 305
miscellaneous 306
process management 305
synchronization 305
trace services
return and reason code values 298
trace table 75
user application information 76
trace table, CMS 75
TraceControl routine
maintaining trace services 75
reference 269
TraceSignal routine 272
transmission of messages example 39
two threads sharing a request queue example 42

U

user application information 76
using CMS multitasking 81
using service queues example 43
USRRTHD entry point 278
USRSINIT entry point 277
USRTINIT entry point 278
USRTTERM entry point 278

V

variable, condition 47
VCPU services
return and reason code values 300
VCPUCreate routine 274

VCPUSelect routine 276
virtual CPUs 57
VMACCOUNT system event
    characteristics 331
    using 65
VMCON1ECB system event
    characteristics 331
    contrasted with VMCONINPUT 332
VMCONINPUT system event
    characteristics 331
    contrasted with VMCON1ECB 332
VMCPIC system event
    characteristics 331
VMERROR system event
    characteristics 331
    using 67
VMERRORCHILD system event
    characteristics 331
    using 67
VMIPC system event
    characteristics 331
    using 216
VMIPC, APPC/VM private resource name 34
VMMTLIB callable services library 81
VMPOSGNL system event
    characteristics 331
    using 291
VMPROCESSEND system event
    characteristics 331
    using 12
VMSFSASYNC system event
    characteristics 331
VMSOCKET system event
    characteristics 331
    signal data 332
VMSTART multitasking initialization entry routine
    creating a process 12
VMTIMECHANGE system event
    characteristics 331
    using 59
VMTIMER system event
    characteristics 331
    using 59
VMTRACE system event
    characteristics 331
    using 75

W

wildcard characters 30, 31
writers of servers, suggestion for 287
writing multitasking applications 81

Y

yielding control to other threads 251

Z

z/VM HELP Facility, using 95

Product Number: 5741-A09

Printed in USA

SC24-6258-73