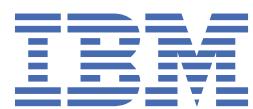


z/VM  
7.4

*Group Control System*



**Note:**

Before you use this information and the product it supports, read the information in “[Notices](#)” on page [541](#).

This edition applies to version 7, release 4 of IBM® z/VM® (product number 5741-A09) and to all subsequent releases and modifications until otherwise indicated in new editions.

Last updated: 2024-09-18

© Copyright International Business Machines Corporation 2001, 2024.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

---

# Contents

<b>Figures.....</b>	<b>ix</b>
<b>Tables.....</b>	<b>xi</b>
<b>About This Document.....</b>	<b>xiii</b>
Intended Audience.....	xiii
Syntax, Message, and Response Conventions.....	xiii
Where to Find More Information.....	xv
Links to Other Documents and Websites.....	xv
<b>How to provide feedback to IBM.....</b>	<b>xvii</b>
<b>Summary of Changes for z/VM: Group Control System.....</b>	<b>xix</b>
SC24-6289-74, z/VM 7.4 (September 2024).....	xix
SC24-6289-73, z/VM 7.3 (December 2023).....	xix
SC24-6289-73, z/VM 7.3 (September 2022).....	xix
SC24-6289-01, z/VM 7.2 (September 2020).....	xix
<b>Chapter 1. Group Control System Overview.....</b>	<b>1</b>
What GCS Is.....	1
What Applications GCS Supports.....	2
How GCS Relates to CMS.....	3
Virtual Machine Groups.....	4
Building the Group.....	4
Joining a Virtual Machine Group.....	5
Implementation of a GCS Group.....	5
GCS Recovery Machine.....	5
GCS Group Communication.....	6
Communicating Between Machines in a Group.....	7
Authorization.....	7
Controlling Access to the GCS Supervisor.....	8
Controlling Access to Supervisor State.....	9
Controlling Access to CP Commands.....	9
GCS Storage.....	10
Overview of GCS Storage Layout.....	11
Private Storage.....	12
Common Storage.....	13
Whole Picture at a Glance.....	13
GCS Scenario.....	14
Establish the Path Between System and Console.....	14
GCS Task Management.....	17
Adding and Discarding Tasks.....	18
Dispatching Tasks.....	19
GCS System Tasks.....	19
Task Dispatching and Multi-tasking Services.....	19
Coordinating Dependent Tasks.....	20
Coordinating Shared Resources.....	22
Terminating Tasks.....	22
Abend Processing.....	23

General I/O (GENIO) Facility.....	23
GCS Real I/O.....	24
<b>Chapter 2. Planning for GCS.....</b>	<b>25</b>
Planning GCS Storage Layout.....	25
Calculating Storage Requirements.....	25
Preparing to Build Other Saved Segments.....	27
Shared Segments Recognized by GCS.....	28
Private Segments for Applications.....	28
Making VSAM Available to GCS.....	28
Authorizing Access to Supervisor State.....	28
Authorizing Access to GCS.....	29
Authorizing Commands for Virtual Machines.....	29
Authorizing Machines for Real I/O.....	29
Using AUTOLOG Functions.....	29
Using a PROFILE GCS File.....	29
Preparing CP Directory Entries.....	30
Operation.....	30
Initializing GCS (How to Join a Group).....	30
Starting and Stopping Programs.....	31
Replying to Messages.....	32
Querying Information.....	32
<b>Chapter 3. GCS Programming and Command Processing.....</b>	<b>35</b>
Linkage Registers.....	35
Establishing a Base Register.....	35
Providing a Save Area.....	35
Example of Chaining Save Areas in a Nonreenterable Program.....	36
Example of Chaining Save Areas in a Reentrant Program.....	36
Summary of Conventions for Passing and Receiving Control.....	37
GCS Program Exits.....	37
GCS Commands Operation.....	38
Example of an Application Program in GCS.....	38
Console and Command Support.....	41
OS Management Services.....	43
Native GCS Services.....	47
Data Management Services.....	51
<b>Chapter 4. GCS Commands.....</b>	<b>57</b>
Immediate Commands.....	58
ACCESS.....	59
CLEAR.....	61
CONFIG.....	62
DLBL.....	64
ERASE.....	70
ESTATE/ESTATEW.....	71
ETRACE.....	73
EXECIO.....	76
Extended Descriptions and Use Information.....	83
EXECIO Return Codes.....	90
Explanation of Message GCTEIO632E.....	91
EXECIO Abend Codes.....	92
FILEDEF.....	94
GDUMP.....	98
GLOBAL.....	101
GROUP.....	102
GROUP Panels.....	103

Function Keys.....	105
HX.....	107
ITRACE.....	108
LOADCMD.....	112
OSRUN.....	116
QUERY.....	117
QUERY ADDRESS.....	119
QUERY AUTHUSER.....	120
QUERY COMMON.....	121
QUERY DISK.....	122
QUERY DLBL.....	124
QUERY DUMP.....	126
QUERY DUMPLOCK.....	127
QUERY DUMPVM.....	128
QUERY ETRACE.....	129
QUERY FILEDEF.....	130
QUERY GCSLEVEL.....	131
QUERY GROUP.....	132
QUERY IPOLL.....	133
QUERY ITRACE.....	134
QUERY LOADALL.....	135
QUERY LOADCMD.....	136
QUERY LOADLIB.....	137
QUERY LOCK.....	138
QUERY MODDATE.....	139
QUERY REPLY.....	140
QUERY REXXSTOR.....	141
QUERY SEARCH.....	142
QUERY SYSNAMES.....	143
QUERY TRACETAB.....	144
QUERY TSLICE.....	145
RELEASE.....	146
REPLY.....	147
SET.....	149
SET DUMP.....	150
SET DUMPLOCK.....	151
SET IPOLL.....	152
SET REXXSTOR.....	153
SET SYSNAME.....	154
SET TSLICE.....	155

<b>Chapter 5. GCS Macros.....</b>	<b>157</b>
GCS Macro Level and Parameter Lists.....	157
Addressing Mode and the Macros.....	158
GCS Macro Formats.....	158
GCS Macro Coding Conventions.....	158
Formatting Conventions.....	159
Parameter Notation Conventions.....	161
ABEND.....	162
ADSR.....	164
ATTACH.....	165
AUTHCALL.....	172
AUTHNAME.....	174
AUTHUSER.....	179
BLDL.....	181
CALL.....	184
CHAP.....	187

CMDSI.....	189
CONFIG.....	193
CONTENTS.....	197
CVT.....	200
DELETE.....	202
DEQ.....	204
DETACH.....	208
DEVTYPE.....	210
ECVT.....	212
ENQ.....	213
ESPIE.....	219
ESTAE.....	223
EXECCOMM.....	229
FLS.....	231
FREEMAIN.....	233
GCSLEVEL.....	238
GCSSAVE.....	240
GCSSAVI.....	241
GCSTOKEN.....	242
GENIO.....	247
GETMAIN.....	257
GTRACE.....	265
IDENTIFY.....	270
IHADVA.....	272
IHASDWA.....	273
IUCVCOM.....	275
IUCVINI.....	286
LINK.....	293
LOAD.....	298
LOCKWD.....	302
MACHEXIT.....	305
PGLOCK.....	310
PGULOCK.....	312
POST.....	314
RDJFCB.....	317
RESSTOR.....	320
RETURN.....	322
SAVE.....	324
SCHEDEX.....	326
SDUMP.....	329
SDUMPX.....	333
SEGMENT.....	338
SETRP.....	340
SPLEVEL.....	343
STIMER.....	345
SYMREC.....	348
SYNCH.....	350
TASKEXIT.....	354
TIME.....	359
TTIMER.....	361
VALIDATE.....	362
WAIT.....	365
WTO.....	368
WTOR.....	370
XCTL.....	373

Using QSAM and BSAM.....	379
CHECK (BSAM).....	380
CLOSE (BSAM/QSAM).....	382
DCB (BSAM/QSAM).....	385
DCBD (BSAM/QSAM).....	391
GET (QSAM).....	394
NOTE (BSAM).....	396
OPEN (BSAM/QSAM).....	398
POINT (BSAM).....	402
PUT (QSAM).....	404
READ (BSAM).....	406
SYNADAF (BSAM/QSAM).....	409
SYNADRLS (BSAM/QSAM).....	411
WRITE (BSAM).....	413

## **Chapter 7. VSAM Data Management Service Macros.....417**

Using VSAM.....	417
ACB.....	418
BLDVRP.....	423
CHECK.....	425
CLOSE.....	427
DLVRP.....	429
ENDREQ.....	430
ERASE.....	432
EXLST.....	434
GENCB.....	437
GET.....	451
MODCB.....	453
OPEN.....	465
POINT.....	468
PUT.....	470
RPL.....	472
SHOWCAT.....	476
SHOWCB.....	480
TESTCB.....	490
WRTBFR.....	505

## **Appendix A. Tailoring and Building the GCS Nucleus.....507**

Changing GCS Nucleus Options.....	507
Creating a New GCS Nucleus Build List.....	507
Changing GCS Default Definitions.....	512
Rebuilding and Saving the GCS Nucleus.....	513

## **Appendix B. Using VSAM.....517**

VSAM I/O Operations under GCS.....	517
Control-Block Manipulation Macros.....	518
VSAM Macro Addresses.....	518
List Format.....	518
List Address Format.....	518
Execute Format.....	518
Generate Format.....	518
Parameter Notation for GENCB, MODCB, SHOWCB, and TESTCB Macros.....	519
GENCB Macro.....	520
MODCB Macro.....	521
SHOWCB Macro.....	522
TESTCB Macro.....	522
Feedback Field Codes.....	524

When the Return Code in Register 15 is 0.....	524
When the Return Code in Register 15 is 8.....	525
When the Return Code in Register 15 is 12.....	528
<b>Appendix C. Appendix for QUERY ADDRESS and QUERY MODDATE.....</b>	<b>529</b>
<b>Appendix D. Data Compression Services.....</b>	<b>539</b>
Compression and Expansion Services.....	539
<b>Notices.....</b>	<b>541</b>
Programming Interface Information.....	542
Trademarks.....	542
Terms and Conditions for Product Documentation.....	542
IBM Online Privacy Statement.....	543
<b>Bibliography.....</b>	<b>545</b>
Where to Get z/VM Information.....	545
z/VM Base Library.....	545
z/VM Facilities and Features.....	546
Prerequisite Products.....	548
Related Products.....	548
<b>Index.....</b>	<b>549</b>

---

# Figures

1. Group Control System, an Interface between Applications and the Control Program.....	1
2. A Virtual Machine Group and Supported Applications.....	2
3. Storage Management Anchor Block.....	11
4. GCS Storage Layout.....	12
5. GCS in z/VM.....	14
6. CP Intercepts Instructions from the Virtual Machine.....	15
7. Transferring Data to the Machine Running VSCS.....	16
8. Path of Data Moving through the VTAM Machine.....	17
9. Data Traveling from VTAM to the Virtual Console.....	17
10. Diagram of a Task's Family Tree.....	18
11. Task Block Dispatch Priority.....	20
12. How Tasks Can Use WAIT and POST Macros.....	21
13. Ideal Locations of Common and Private Storage in Two Virtual Machine Group Members.....	25
14. Obtaining Modules Requested by a GCS Program.....	45
15. Determining Which VSAM Catalog to Use .....	67
16. GROUP Primary Option Menu Panel.....	103
17. GROUP Authorized VM User IDs Panel.....	103
18. Saved System Information Panel, Page 1.....	104
19. GROUP Saved System Information Panel, Page 2.....	104
20. GROUP Automatic Saved Segment Links Panel.....	105
21. GROUP User IDs Requiring Reserved Storage for VSAM Panel.....	105



---

# Tables

1. Examples of Syntax Diagram Conventions.....	xiii
2. Authorization in the GCS Environment.....	8
3. Problem State Versus Supervisor State.....	9
4. Automatic Disk Access at IPL.....	31
5. Supported Commands.....	43
6. Loading Functions.....	45
7. The AUTHCALL Macro.....	47
8. Opening Multiple DCBs.....	52
9. Valid ANSI Control Characters for Carriage Control.....	87
10. Function Keys Used with the GROUP Panels.....	105
11. Query commands.....	117
12. Set commands.....	149
13. GCS Macros (Part 1 of 2).....	157
14. GCS Macros (Part 2 of 2).....	157
15. Supported Device Characteristics Information.....	272
16. Information returned by RDJFCB for BSAM/QSAM.....	318
17. Information returned by RDJFCB for VSAM.....	318
18. SDUMPX LISTD parameter list format.....	334
19. SDUMPX SUMLSTL parameter list format.....	335
20. Exit List Format.....	387
21. DCB Exit List Codes.....	387
22. Register content when error routine receives control.....	389



# About This Document

---

This document provides information on how to plan for, set up, and operate the IBM® z/VM® Group Control System (GCS). It also contains complete reference information for all of the GCS commands and macros.

## Intended Audience

---

This document is intended for system programmers and administrators who need to plan for GCS and run it on their z/VM system. This document is also for application programmers who need to write programs to run under GCS.

## Syntax, Message, and Response Conventions

---

The following topics provide information on the conventions used in syntax diagrams and in examples of messages and responses.

### How to Read Syntax Diagrams

Special diagrams (often called *railroad tracks*) are used to show the syntax of external interfaces.

To read a syntax diagram, follow the path of the line. Read from left to right and top to bottom.

- The **►—►** symbol indicates the beginning of the syntax diagram.
- The **——►** symbol, at the end of a line, indicates that the syntax diagram is continued on the next line.
- The **◄——** symbol, at the beginning of a line, indicates that the syntax diagram is continued from the previous line.
- The **——►◄** symbol indicates the end of the syntax diagram.

Within the syntax diagram, items on the line are required, items below the line are optional, and items above the line are defaults. See the examples in [Table 1 on page xiii](#).

*Table 1. Examples of Syntax Diagram Conventions*

Syntax Diagram Convention	Example
<b>Keywords and Constants</b>  A keyword or constant appears in uppercase letters. In this example, you must specify the item KEYWORD as shown.  In most cases, you can specify a keyword or constant in uppercase letters, lowercase letters, or any combination. However, some applications may have additional conventions for using all-uppercase or all-lowercase.	<b>►— KEYWORD —►</b>
<b>Abbreviations</b>  Uppercase letters denote the shortest acceptable abbreviation of an item, and lowercase letters denote the part that can be omitted. If an item appears entirely in uppercase letters, it cannot be abbreviated.  In this example, you can specify KEYWO, KEYWOR, or KEYWORD.	<b>►— KEYWOrd —►</b>

Table 1. Examples of Syntax Diagram Conventions (continued)

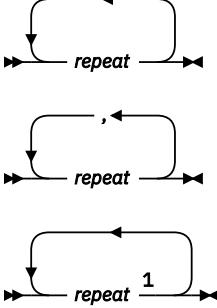
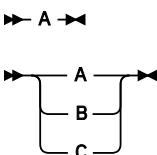
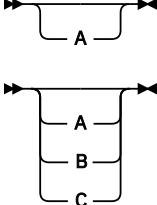
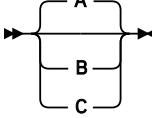
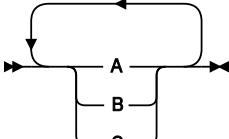
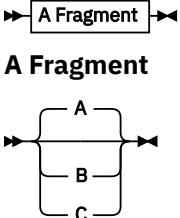
Syntax Diagram Convention	Example
<b>Symbols</b> You must specify these symbols exactly as they appear in the syntax diagram.	* Asterisk : Colon ,, Comma = Equal Sign - Hyphen 0 Parentheses . Period
<b>Variables</b> A variable appears in highlighted lowercase, usually italics. In this example, <i>var_name</i> represents a variable that you must specify following KEYWORD.	► KEYWORD — <i>var_name</i> ►
<b>Repetitions</b> An arrow returning to the left means that the item can be repeated. A character within the arrow means that you must separate each repetition of the item with that character. A number (1) by the arrow references a syntax note at the bottom of the diagram. The syntax note tells you how many times the item can be repeated. Syntax notes may also be used to explain other special aspects of the syntax.	 <p>Notes:  <sup>1</sup> Specify <i>repeat</i> up to 5 times.</p>
<b>Required Item or Choice</b> When an item is on the line, it is required. In this example, you must specify A. When two or more items are in a stack and one of them is on the line, you must specify one item. In this example, you must choose A, B, or C.	
<b>Optional Item or Choice</b> When an item is below the line, it is optional. In this example, you can choose A or nothing at all. When two or more items are in a stack below the line, all of them are optional. In this example, you can choose A, B, C, or nothing at all.	

Table 1. Examples of Syntax Diagram Conventions (continued)

Syntax Diagram Convention	Example
<b>Defaults</b> When an item is above the line, it is the default. The system will use the default unless you override it. You can override the default by specifying an option from the stack below the line.  In this example, A is the default. You can override A by choosing B or C.	
<b>Repeatable Choice</b> A stack of items followed by an arrow returning to the left means that you can select more than one item or, in some cases, repeat a single item.  In this example, you can choose any combination of A, B, or C.	
<b>Syntax Fragment</b> Some diagrams, because of their length, must fragment the syntax. The fragment name appears between vertical bars in the diagram. The expanded fragment appears in the diagram after a heading with the same fragment name.  In this example, the fragment is named "A Fragment."	

## Examples of Messages and Responses

Although most examples of messages and responses are shown exactly as they would appear, some content might depend on the specific situation. The following notation is used to show variable, optional, or alternative content:

**xxx**

Highlighted text (usually italics) indicates a variable that represents the data that will be displayed.

**[]**

Brackets enclose optional text that might be displayed.

**{}**

Braces enclose alternative versions of text, one of which will be displayed.

**|**

The vertical bar separates items within brackets or braces.

**...**

The ellipsis indicates that the preceding item might be repeated. A vertical ellipsis indicates that the preceding line, or a variation of that line, might be repeated.

## Where to Find More Information

Other documents in the z/VM library are shown in the [“Bibliography” on page 545](#).

## 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.

## **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.



# Summary of Changes for z/VM: Group Control System

---

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.

## SC24-6289-74, z/VM 7.4 (September 2024)

---

This edition supports the general availability of z/VM 7.4. Note that the publication number suffix (-74) indicates the z/VM release to which this edition applies.

### [7.4] Linear service

z/VM 7.4 introduces linear service to the product for components at the 740 function level. Corrective service updates — in the form of fix packs, hot fixes, and hardware support — and new functions in feature packs are released in service stream PTFs. The latest PTF identifies the requisites of all fixes and features for a component up to that point. For more information about linear service, including schedules and the types of APARs, see:

[Introducing z/VM Linear Service \(https://www.vm.ibm.com/service/linear.html\)](https://www.vm.ibm.com/service/linear.html)

### New format for service level

The product service level, as reported by z/VM components with a query service command, uses new format *ffxx*:

- *ff* indicates the latest feature pack number.
- *xx* indicates the fix pack number for the latest feature pack.

The format of the service level number is updated in the following topic: [“QUERY GCSLEVEL” on page 131](#)

## SC24-6289-73, z/VM 7.3 (December 2023)

---

This edition includes changes to support product changes that are provided or announced after the general availability of z/VM 7.3.

## SC24-6289-73, 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.

## SC24-6289-01, z/VM 7.2 (September 2020)

---

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



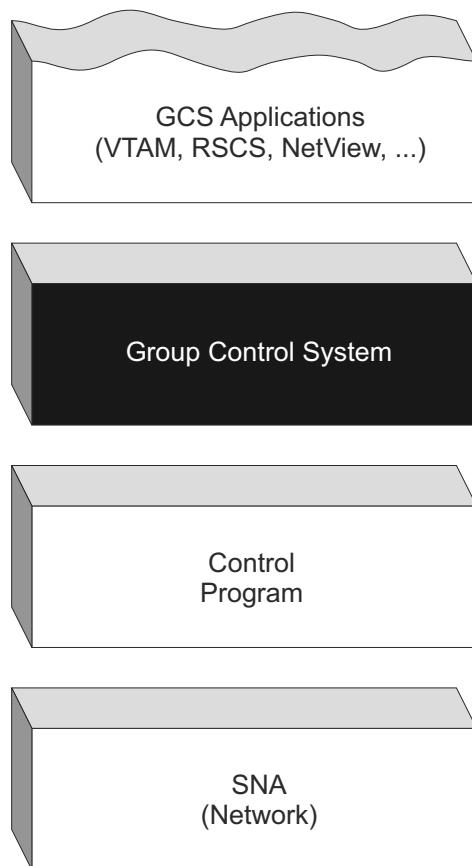
# Chapter 1. Group Control System Overview

## What GCS Is

The Group Control System (GCS) is:

- A component of z/VM. It consists of a named, shared segment in storage that you can IPL and run in a virtual machine.
- A virtual machine supervisor. It bands many virtual machines together in a group and supervises their operations. See [Figure 2 on page 2](#).
- An interface between applications. Some of the applications are:
  - Virtual Telecommunications Access Method (VTAM<sup>®</sup>)
  - Remote Spooling Communications Subsystem (RSCS)
  - NetView<sup>®</sup>
  - z/VM's Control Program (CP), [Figure 1 on page 1](#).

GCS provides *multitasking* services that allow numerous tasks to remain active in the virtual machine at one time.



*Figure 1. Group Control System, an Interface between Applications and the Control Program*

The specific function of GCS for z/VM is to support a native VM/SNA network — a network that functions as part of your z/VM system without help from a second operating system. This System Network Architecture (SNA) network relies on ACF/VTAM, VTAM SNA Console Support (VSCS), and other network applications to manage its collection of links between terminals, controllers, and processors. In turn,

## Group Control System Overview

ACF/VTAM, VSCS, and the others rely on GCS to provide services for them. This arrangement eliminates your need for VM/VCNA (VTAM Communications Network Application) and a second operating system like VS1 or VSE.

GCS runs in an XA machine and is installed with z/VM. This allows all virtual machines in a GCS group to run in XA mode or XC mode.

- XA mode entails running with the full capabilities of the Extended System Architecture. Either 24-bit or 31-bit addressing can be used (thus allowing addresses below and above 16MB), as well as the more efficient XA I/O using the Channel Subsystem.
- ESA/XC architecture is a virtual machine architecture in which DAT-off programs can create and access additional address spaces called data spaces. These additional address spaces can also be shared with programs running in other virtual machines. GCS applications must not be in Access Register (AR) mode when using GCS supervisor services, whether using a branch or an SVC interface, or returning from a user exit. GCS will abend any application that attempts to use GCS supervisor services while in AR mode.
- For migration purposes GCS supports mixed mode groups which include XC and XA mode.

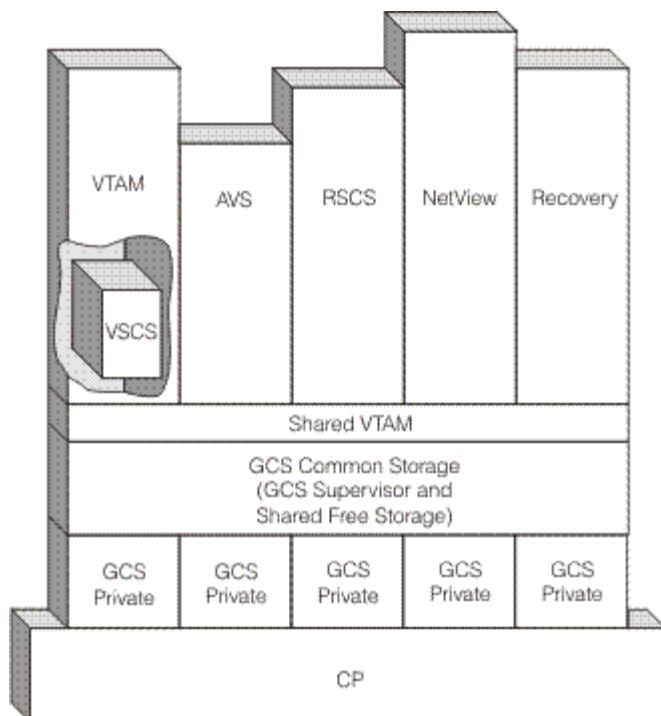


Figure 2. A Virtual Machine Group and Supported Applications

## What Applications GCS Supports

GCS supports these applications:

### VTAM (Virtual Telecommunications Access Method)

The specific version of VTAM designed for GCS is ACF/VTAM Version 3.3 (for z/VM). ACF/VTAM controls data flow between SNA network devices and programs running in other group machines.

Part of ACF/VTAM provides a shared VTAM interface that other applications like RSCS, NCCF, and NetView pass information through. (See [Figure 2 on page 2](#).) RSCS uses this shared VTAM interface to communicate with SNA devices; NCCF and NetView perform network management functions through it. For more information, see the *ACF/VTAM General Information (for VM)* book.

**VSCS (VTAM SNA Console Support)**

This is a VTAM component that lets SNA-connected terminals function as virtual machine consoles. VSCS succeeds the earlier VM/VCNA product, and makes a guest System Control Program (SCP), like VSE or VS1, unnecessary. For more information, see the *ACF/VTAM General Information (for VM)* book.

**SSP (Systems Support Program)**

With GCS, parts of SSP are VTAM subtasks. SSP does utility functions for the SNA network's communication control unit. Actually, SSP aids the Network Control Program (NCP), which governs the communication control unit. That control unit, in turn, manages network lines and routing of data. For more information, see the *ACF/VTAM Network Program Products Planning* book.

**AVS (APPC/VM VTAM Support)**

Is a z/VM-supplied VTAM application that runs in a GCS virtual machine. It provides the functions necessary for APPC/VM programs within a TSAF collection to communicate with APPC programs anywhere in an SNA network. VTAM provides the LU 6.2 services necessary to communicate with a remote LU. AVS handles the transformation between APPC/VTAM and APPC/VM. AVS can coexist with VSCS in the same system, GCS group, and virtual machine. For more information, see [z/VM: Connectivity](#).

**RSCS (Remote Spooling Communications Subsystem)**

RSCS, designed as a GCS application, runs in a group virtual machine and relies on ACF/VTAM to help transfer information through the SNA network. RSCS also can run in a group by itself, spooling files and transmitting messages through non-SNA links. For more information, see the *RSCS Networking General Information* book.

**NetView**

NetView is an enhanced network management program. It is an optional but recommended VTAM application that helps the operation and control of a SNA network. It permits your network operator to control any portion of the network regardless of its physical location. NetView includes the function of the following network management products that are also supported by GCS, plus enhancements in the areas of function, usability, installability, and operability:

- NCCF
- NPDA (Network Problem Determination Application)
- NLDM (Network Logical Data Manager).

For more information, see the *ACF/VTAM Network Program Products Planning* book.

## How GCS Relates to CMS

---

GCS, like CMS, is a z/VM component. Although these two components share a few similarities, they have very different functions. GCS supports more than 70 OS macros. Over 50 of them have CMS counterparts (though some of the supported parameters differ), while the remaining macros are unique to GCS. CMS supports its OS macros at the MVS/SP Release 2.2.0 level and DFP Release 2.3.0 level, while GCS supports its OS macros at the MVS/SP Release 2.2.0 level.

Some GCS commands resemble ones that exist in CMS. These commands share the same or slightly modified formats in both environments:

ACCESS	FILEDEF	QUERY
DLBL	GLOBAL	RELEASE
ERASE	HX	SET
EXECIO	OSRUN	

The actual command formats are described later in this book.

In addition, the VSAM interface supported by GCS is the same as the one used by CMS, with VSAM disks in VSE/VSAM format. In fact, the VSAM macros GCS uses reside in the CMS macro library named OSVSAM MACLIB.

Also, GCS has many of the same REXX capabilities as CMS. For more information on the exceptions, see [“Entering Commands to GCS” on page 42](#).

Beyond these similarities and differences, GCS and CMS have another relationship: GCS relies on CMS for its interactive capabilities. For example, you have to complete the GCS build and installation process using CMS. For more information on the explanation of the process, see [z/VM: Installation Guide](#). Even after you have created GCS, you still need CMS for:

- Editing, assembling, and link-editing GCS programs
- Initializing disks and creating catalogs (utility functions)
- Creating VTAM's network definition files
- Creating REXX files of file type GCS
- Building VSAM saved segments
- Examining and printing dumped storage information.

Files used by GCS **cannot** reside in an SFS file pool. SFS is a part of CMS. GCS does not use CMS to do its file I/O.

## Virtual Machine Groups

---

A virtual machine group is an extension to the current virtual machine supported by the Control Program (CP), which allows several virtual machines to be in a common group and controlled by a common supervisor. More than one group may be active at any given time in a single processor. A *group* is one or more virtual machines that have IPLed the same GCS shared segment.

Virtual machine groups can consist of multiple user groups or single user groups. The group environment is defined with the GRP121 screen at system installation time. Multiple user groups share common storage space and a supervisor and can communicate with each other. Single user groups do not need to share storage space or supervisor because there are no other machines in the group and they do not need to have the ability to communicate with other machines in a group. Therefore, applications that do not require group communication are able to IPL and run without the overhead of group initialization and multiple virtual machines.

GCS governs the group's machines. It is a base that holds the group together and a supervisor that provides many services for each member machine. The type of services available depends on the *authorization* of individual group members. Unauthorized members run only in problem state and are prevented from using certain GCS services. Authorized members can run in supervisor state and use more GCS services.

In a single user group, the user authorization is initialized as specified in the configuration file. The user may change the authorization by using the AUTHUSER parameter of the CONFIG command.

[Figure 2 on page 2](#) shows the structure of a virtual machine group. The virtual machine group, with a built-in supervisor, supports a z/VM operating environment for programs, like VTAM, that once needed guest operating systems.

## Building the Group

When you define and install GCS, you provide information that builds, or configures, your group. This information goes into a group configuration file that resides in GCS private storage. Some of your input to that file includes:

- A name for the supervisor (actually, your GCS system name)
- User IDs of machines authorized to run in supervisor state
- A maximum group size
- The user ID of one virtual machine, called a recovery machine, to *clean up* group resources when other machines leave the group
- Names of other shared segments (like VTAM and others)

- Location of the internal trace table
- User ID that will be accessing the VSAM segment.

After you have built a configuration file and installed your GCS segment, the GCS supervisor admits machines that IPL the shared segment, by name, into the group. On a single z/VM system you can build multiple GCS segments and multiple virtual machine groups.

## Joining a Virtual Machine Group

After you have installed GCS and defined it as a named, saved system, user IDs can IPL it and share a group copy of the GCS shared segment if the segment was not defined as restricted. Those user IDs then share access to GCS supervisor code and common storage.

If the segment was defined as restricted, the user needs to put a NAMESAVE control statement in the directory to IPL it.

To join a virtual machine group, you log on and IPL the GCS shared segment.

Common storage is a read/write area with two parts:

- Common free storage contains free storage space for applications to use.
- Shared GCS code contains the group's shared copy of GCS supervisor code, along with control blocks and data that all members of the group share.

## Implementation of a GCS Group

When a GCS segment is built, CP does not check for changes in the virtual machines that access the GCS segment. GCS has been structured to run in such an unprotected shared segment to gain the advantages of common storage, fast communication between virtual machines and less dispatching overhead for better system performance. Areas of these segments which need to be protected are protected by storage keys.

A GCS shared segment is declared using the DEFSYS command with the VMGROUP parameter. CP is notified that any virtual machine that IPLs this named segment will be running in a virtual machine group. The common storage is described in the DEFSYS command with the SW descriptor code. This allows the pages of this segment to be altered by any authorized program in the group.

## GCS Recovery Machine

The recovery machine manages the different virtual machines and does cleanup operations for virtual machines in the GCS group that are reset. A virtual machine is reset under any of these conditions:

- Logging off
- Issuing the IPL command
- Receiving some types of machine checks
- Issuing the following CP commands:

```
SYSTEM RESET
SYSTEM CLEAR
DEFINE STORAGE
SET MACHINE
```

Authorized applications can define machine exits which will run when a virtual machine leaves the group. All machine exits that have been defined by using the MACHEXIT macro must reside in a shared segment and will be executed in the recovery machine. When a virtual machine leaves the group, the recovery machine is notified. These operations are also performed:

- Locks held by the terminating virtual machine are freed.
- All machine exits that have been defined within the group are executed.
- Cleanup is performed on the control blocks that keep track of the terminating virtual machine.

This allows the GCS supervisor to clean up any resources that were held by the virtual machine. It also provides authorized applications with a mechanism to be notified when a virtual machine leaves the group.

The recovery machine must be the first virtual machine to be IPLed in a GCS group. The user ID for the recovery machine is designated at build time. A GCS group may contain only the recovery machine (a group of one). If the recovery machine itself gets reset, the machines remaining in the virtual machine group will issue a CP SYSTEM RESET, which causes the entire group to reset.

### Single User Group

When running in a single user group, the user's virtual machine is considered the recovery machine and the dump receiving virtual machine regardless of the virtual machine ID specified by the group EXEC for the saved system.

## GCS Group Communication

One of the primary reasons for having groups of virtual machines is to gain performance in communication. This is accomplished by GCS services such as common storage, Inter-User Communication Vehicle (IUCV), Advanced Program to Program Communication (APPC/VM), and CP Signal System Service.

### GCS APPC/VM and CP Signal System Service

GCS supports these communications:

- Task to task within a virtual machine
- Task to task in different virtual machines within the group
- GCS virtual machine to a virtual machine outside the group

This communication is accomplished by using the GCS IUCV or APPC/VM support or GCS services which use the CP Signal System Service. For communications between virtual machines within the group or outside the group, applications should use the APPC/VM protocol and services. The GCS support macros IUCVINI and IUCVCOM must be used by applications which want to communicate using GCS IUCV or APPC/VM services. The IUCVINI macro initializes, alters, or terminates a user's GCS IUCV or APPC/VM environment. The IUCVCOM macro must be used to establish or terminate an IUCV or APPC/VM path for all GCS support users. This allows GCS task termination to sever any IUCV or APPC/VM paths that may have been left by a terminating task.

An authorized application may use IUCV or APPC/VM directly by issuing the function directly to CP, rather than going through GCS through the IUCVCOM macro, for all functions other than connect or sever. This is accomplished by specifying PRIV=YES when initializing the GCS IUCV or APPC/VM environment with the IUCVINI macro. All unauthorized GCS IUCV or APPC/VM users must use the IUCVCOM macro for GCS communications.

GCS also can communicate with other members of the group by using the CP Signal Service. A virtual machine joins a group when the GCS supervisor is IPLed. At initialization time, GCS will issue an IUCV Declare Buffer and an IUCV CONNECT to the Signal System Service. The connection is made by specifying \*SIGNAL as the user ID and indicating that parameter list data will be used (PRMDATA=YES). Only one connection is allowed to the Signal System Service per virtual machine.

When a source virtual machine determines that it needs to communicate with a target virtual machine in the group, GCS places information describing the request for service into a read/write common storage area and chains it into a queue of requests for the target virtual machine. The source virtual machine then issues an IUCV SEND to the Signal System Service specifying a 8-byte parameter list of data and the target virtual machine's signal id. This signal id was assigned at initialization time. When the SEND is issued, CP generates an external interrupt to be queued for the target virtual machine. The next time the target virtual machine is dispatched by CP, and is enabled for interrupts, it processes the request. The request is then processed by the IUCV interrupt handler. The GCS IUCV interrupt handler identifies the interrupt as one from the Signal System Service and sends the request to the appropriate second level

interrupt handler to be processed. This method of communication allows all the virtual machines in the group to communicate on only one IUCV path.

There are several GCS services which use the Signal System Service for communication. One is to allow for cross-machine lock synchronization. If two virtual machines wish to access the same resource they can obtain the common lock. This is done by using the LOCKWD service in GCS. When a requested lock is released, LOCKWD uses the Signal System Service to notify any waiting virtual machines in the group that the lock is now available.

GCS also uses the Signal System Service to allow for cross-machine exits. One virtual machine can schedule an exit to run on another virtual machine. This is done by using the GCS SCHEDEX function. SCHEDEX uses the Signal System Service to generate the external interrupt on the target virtual machine.

The Signal System Service is also used by CP to notify members of a group when one of the virtual machines leaves that group. As part of the virtual machine reset process, CP will issue an IUCV SEND to all of the remaining members of the group. The IUCV SEND generates a Signal-out external interrupt and the departing virtual machines signal id. The Signal-out external interrupt is used by the virtual machine designated as the recovery machine. The recovery machine runs machine termination exits and does any cleanup necessary (see “[GCS Recovery Machine](#)” on page 5 for more details).

## Communicating Between Machines in a Group

Machines in a group communicate with each other through:

- IUCV (Inter-User Communications Vehicle)
  - IUCV handles communication between virtual machines within a single VM system or between a virtual machine and a CP service. For more information on IUCV, see [z/VM: CP Programming Services](#). In addition, it handles communications between routines (*task-users*) within virtual machines. See “[Communicating through IUCV](#)” on page 47 for details.
- APPC/VM (Advanced Program-to-Program Communication/VM)
  - APPC/VM is a means of communication between two virtual machines. It is mappable to the SNA LU 6.2 APPC interface and is based on z/VM IUCV functions. With the Transparent Services Access Facility (TSAF) virtual machine component, APPC/VM provides communication services within a single system and throughout a group of virtual machines on different systems the same way that IUCV provides them within a system. See “[Communicating through IUCV](#)” on page 47.
- CP Signal System Service
  - Each machine receives a unique *signal ID* when it joins a group. When one machine wants to exchange information with a second group member, it:
    - Records this information in common storage, and
    - Notifies the second machine's signal ID of the information waiting in common storage.

**Note:** If you have many groups, and machines in one group want to communicate with machines in another, they must use IUCV instead of the CP Signal System Service. Although the CP Signal System Service provides unique signal IDs within a group, it reuses the same IDs across different groups.

## Single User Group

GCS running in a single user group environment does not use CP signal system service support because a single VM group does not share common storage and because no other virtual machines are in the group. Applications that depend on sending or receiving signals from other virtual machines cannot be run in this environment.

## Authorization

---

GCS provides protection for both the system and its applications by authorities. There are methods for controlling the execution of programs and the protection of data.

## Group Control System Overview

In GCS, applications may be either authorized or unauthorized. An authorized application will run in supervisor state and has the power to process authorized GCS functions. Unauthorized applications run in problem state and cannot access these authorized functions (except when they are provided access from an authorized application).

There are three types of authorization in GCS:

- Virtual machine

A virtual machine is authorized when its user ID is entered at build time using the GCS GROUP EXEC. When an authorized user ID is IPLed, its applications process in supervisor state. Therefore, a program executing in that virtual machine is authorized and may process both authorized and unauthorized programs.

- Task

When a GCS task is authorized, the programs running under that task are executing in supervisor state. This happens when the task is authorized using the SM=SUPV parameter on the ATTACH macro.

- Entry point.

An authorized entry point can be created using the AUTHNAME macro. This entry point must reside in the GCS shared segment. An authorized application can make this entry point available to unauthorized applications by using AUTHNAME. This declares the entry point (and name) to GCS so unauthorized applications can run it by using the AUTHCALL macro, from any virtual machine in the group. When the AUTHCALL macro is called by an unauthorized program, the authorized entry will be given control in supervisor state (authorized). When it returns control to the unauthorized problem, problem (unauthorized) state is restored. In this way, authorized applications can provide unauthorized applications a controlled means of accessing authorized functions.

Validation (see “[VALIDATE](#)” on page 362) means that a check is done to confirm that a program has access to a certain block of storage. An unauthorized program has all parameter list addresses validated by GCS whenever a call is made for a GCS service. An authorized function will not have their parameter lists validated. This may result in a significant performance savings but authorized programs must be careful that they are accessing data and functions correctly. When a program in GCS is authorized, it does not necessarily mean that it is also authorized to a GCS application. For example, in VTAM, a GCS program is not considered authorized unless it uses the AUTHEXIT=YES parameter on the VTAM APPL statement.

GCS data is also protected with storage keys. An authorized application can obtain storage in different storage keys. Unauthorized applications may only obtain storage in key 14. If an application tries to access the storage in a key other than its own, it will receive an error. The organization of GCS storage is discussed in “[GCS Storage](#)” on page 10.

There are three levels of authorization in the GCS environment. With each increasing level of authorization, you receive a greater amount of access to the GCS system. (The first level has the least amount of access. The third level has the most, because it requires authorization at the previous two levels.) You authorize who gets access to each level.

*Table 2. Authorization in the GCS Environment*

At Level	User IDs Have Access To
1	The GCS supervisor and common storage
2	Supervisor State (and privileged GCS functions)
3	Certain restricted CP commands

## Controlling Access to the GCS Supervisor

The GCS supervisor is part of the GCS shared segment. Having access to the supervisor results from being able to IPL the segment. So if you prevent certain user IDs from IPLing your GCS system, you cut off their access to the supervisor. To do this, specify the RSTD parameter on the DEFSSYS command when defining the GCS system.

To enter an IPL command and successfully access the (restricted) GCS supervisor, a user ID must be authorized in the directory with the NAMESAVE control statement.

If universal access to the GCS supervisor is desired (not recommended) the RSTD option may be omitted from the DEFSYS command. It will then be impossible to prevent any user from accessing the supervisor.

## Controlling Access to Supervisor State

After a user ID has access to the GCS supervisor, it will operate in problem state unless you authorize it to run in supervisor state. You provide access to supervisor state by:

- Authorizing the user ID at build time

When defining the virtual machine group (see “[Changing GCS Default Definitions](#)” on page 512) you provide a list of user IDs that will have access to supervisor state and authorized GCS functions.

The virtual machine associated with an authorized user ID is called an *authorized machine*. And, any applications that run under these authorized user IDs are considered *authorized* too.

- Authorizing entry points

You can select a certain entry point, a location in a shared segment, to run in supervisor state. (GCS's AUTHNAME macro lets authorized programs identify these authorized entry points.) A problem state program can pass control to that entry point, which will run in supervisor state. The program later will regain control in problem state.

- The CONFIG command

You can also provide access to supervisor state dynamically by using CONFIG AUTHUSER ADD in the recovery machine.

[Table 3 on page 9](#) describes how problem state and supervisor state differ:

*Table 3. Problem State Versus Supervisor State*

Problem State	Supervisor State
Both authorized and unauthorized user IDs can run applications in this state.	Only authorized user IDs can run applications in this state.
User IDs in this state cannot use privileged GCS functions or macros.	User IDs in this state can use privileged GCS functions or macros.
User IDs can use only storage having a storage protection key of 14.	User IDs can use storage of any key.

## Controlling Access to CP Commands

After IPLing GCS, many CP commands (like SPOOL, LINK, and MESSAGE) will work without disrupting or affecting your system's ability to function. But some CP commands will harm your GCS code, and others have limited usefulness.

For example, the CP commands BEGIN, DISPLAY, DUMP, STORE, TRACE, and VMDUMP permit you to view or alter common storage. Only certain users who are responsible for maintaining and debugging your system should be able to enter them.

With z/VM you can make potentially harmful CP commands unavailable to your GCS user IDs. You must either alter the lists of commands in two existing privilege classes (A through H) or else define two new privilege classes. Depending on how you redefine these privilege classes, you may have to change the *c/* parameter specified on each GCS user ID's USER control statement in your directory.

Whether you choose to define new classes or alter existing ones, make sure you have two privilege classes that contain:

- CP debugging commands for authorized use only

This privilege class should include all current Class G commands. Assign it only to authorized user IDs responsible for maintenance and debugging.

- General-use CP commands for unauthorized user IDs

This privilege class should include all current Class G commands, except TRACE, BEGIN, DISPLAY, DUMP, STORE, and VMDUMP. Assign it to unauthorized user IDs that do not need debugging commands.

**Note:** With this class assignment, unauthorized GCS users cannot use the VMDUMP command. In case of an error, their virtual machines need a way to dump storage. Instead of VMDUMP, they can use the GDUMP command to dump storage and specify where it will go. See “[GDUMP](#)” on page 98.

## GCS Storage

---

To understand the structure of GCS, it is helpful to know the organization and allocation of free storage (see [Figure 3 on page 11](#)). GCS is unique to z/VM because it provides the ability for virtual machines to share read/write storage between them. This is called common storage. Each virtual machine also has its own private storage. These can then be subdivided into different storage keys which can be either fetch protected or nonfetch protected. Fetch protection is enforced by the system architecture. The key of the storage is dependent on the PSW key of the program requesting that storage. GCS supports storage keys 0-15. Some examples of storage allocation are:

- GCS supervisor code runs in key 0, common storage.
- Application storage is assigned key 14, private storage.

An authorized application may change the key of its PSW using the SPKA instruction, thus allowing it to request storage in different storage keys.

Depending on the amount of storage available, GCS can allocate storage over the 16MB line. Both common and private storage can exist below and above the 16MB line. See “[GETMAIN](#)” on page 257, for more details.

When a request for storage comes in, GCS checks its storage chains for storage in the key of the requester. If the request cannot be satisfied, GCS looks for a free full page (or pages) in any key, and changes the key of that page to the key of the requester. The page of storage is then chained onto the appropriate storage chain. This allows any page of storage in GCS to be obtained in any storage key. The page can also be either fetch protected or nonfetch protected.

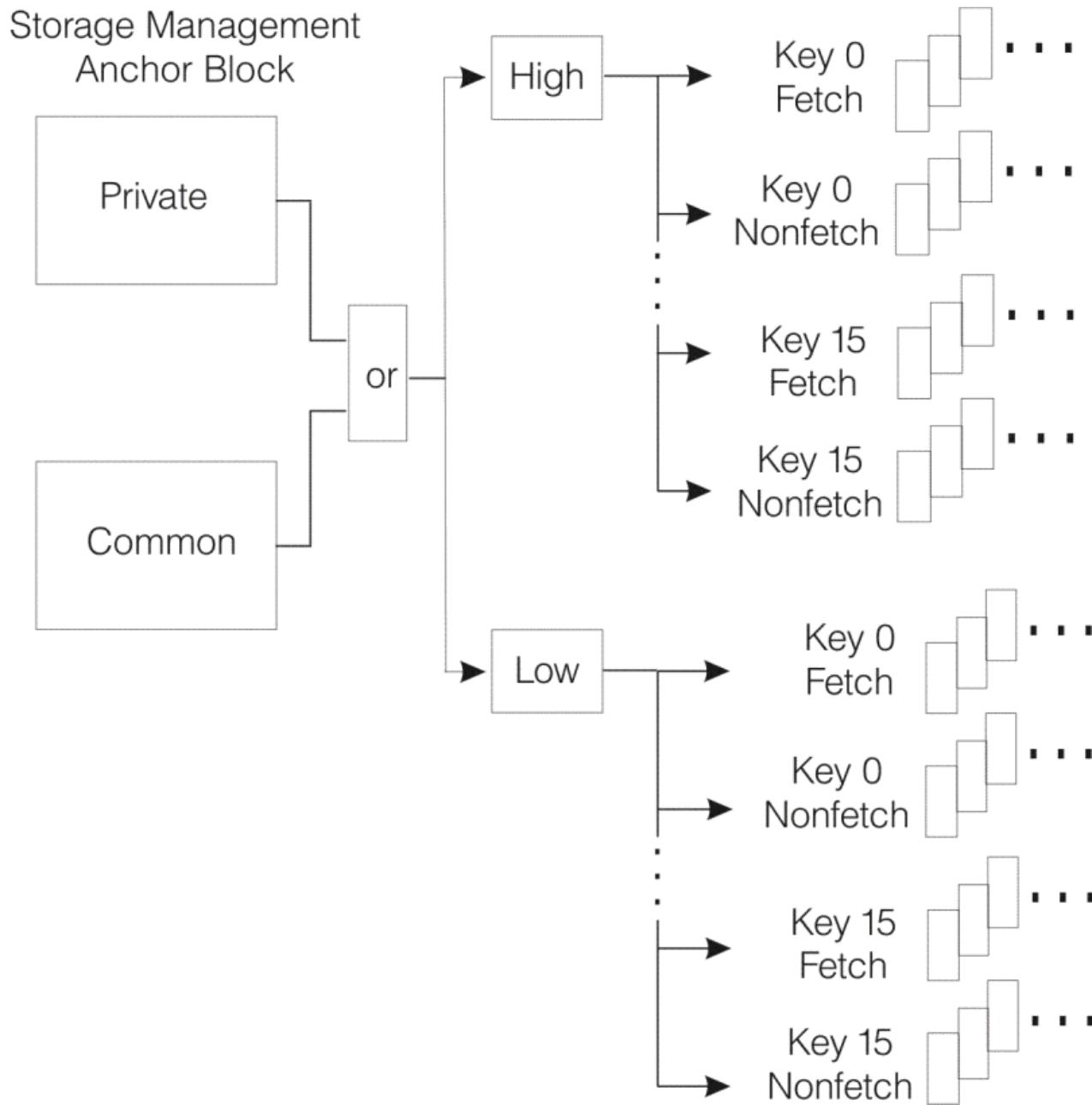


Figure 3. Storage Management Anchor Block

## Overview of GCS Storage Layout

As you can see in [Figure 4 on page 12](#), GCS is divided into two pieces: private storage and common storage.

## Group Control System Overview

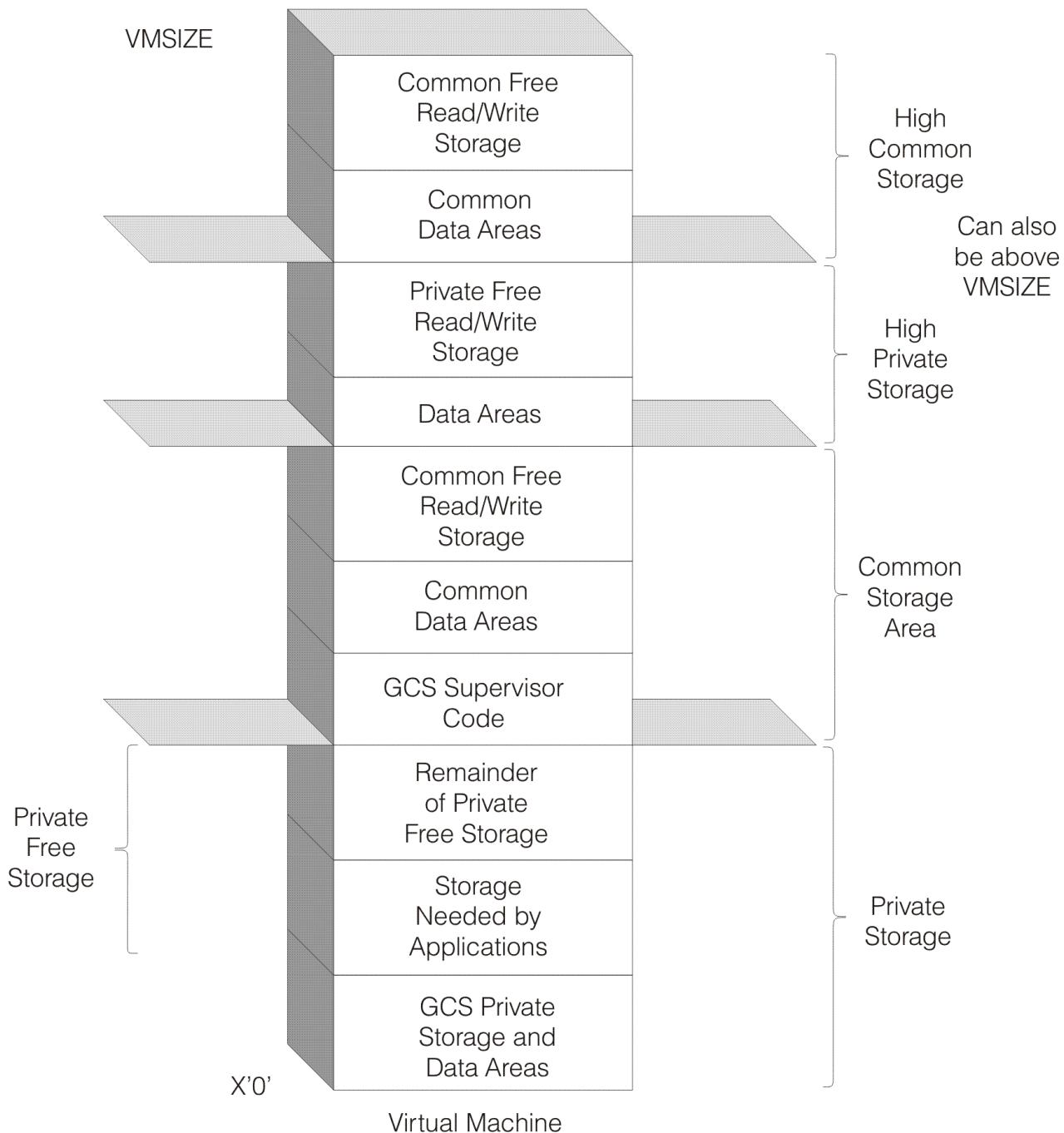


Figure 4. GCS Storage Layout

Private free storage should be contiguous to make the virtual machine more efficient. Private storage is unique to each virtual machine that it is in, but common storage is shared by all users in the group.

## Private Storage

Private storage is divided into:

### GCS Private Storage

Data areas and control blocks that include system pointers, work areas, and the system configuration module. GCS private Storage begins at page 0 of the virtual machine.

### Private Free Storage

Available for GETMAINS and is where application programs are loaded.

## Common Storage

Common storage is divided into:

### GCS Supervisor Code

Common storage that contains all executable modules required to IPL GCS.

### Common Data Areas

Supervisor data that is shared between virtual machines.

### Common Free Storage

Used for GETMAINs, some of this storage is taken for the trace table to be created. Common storage is a shared read/write area of the virtual machine. The common storage is divided into low common storage (below the 16MB line) and high common storage (above the 16MB line).

## Whole Picture at a Glance

---

Figure 5 on page 14 shows a conceptual view of how the Group Control System can fit into your z/VM environment. Familiar elements in the picture include:

- CP, a base for the rest of the system to build on.
- Virtual machines, running various applications.
- CMS, an interactive z/VM component that runs on CP.
- A route to the SNA network, a network that connects virtual machines with remote consoles. (This is just one application of GCS.)

GCS, with its common and private areas, forms a base for a particular group of virtual machines. It runs parallel to CMS as a z/VM component on CP.

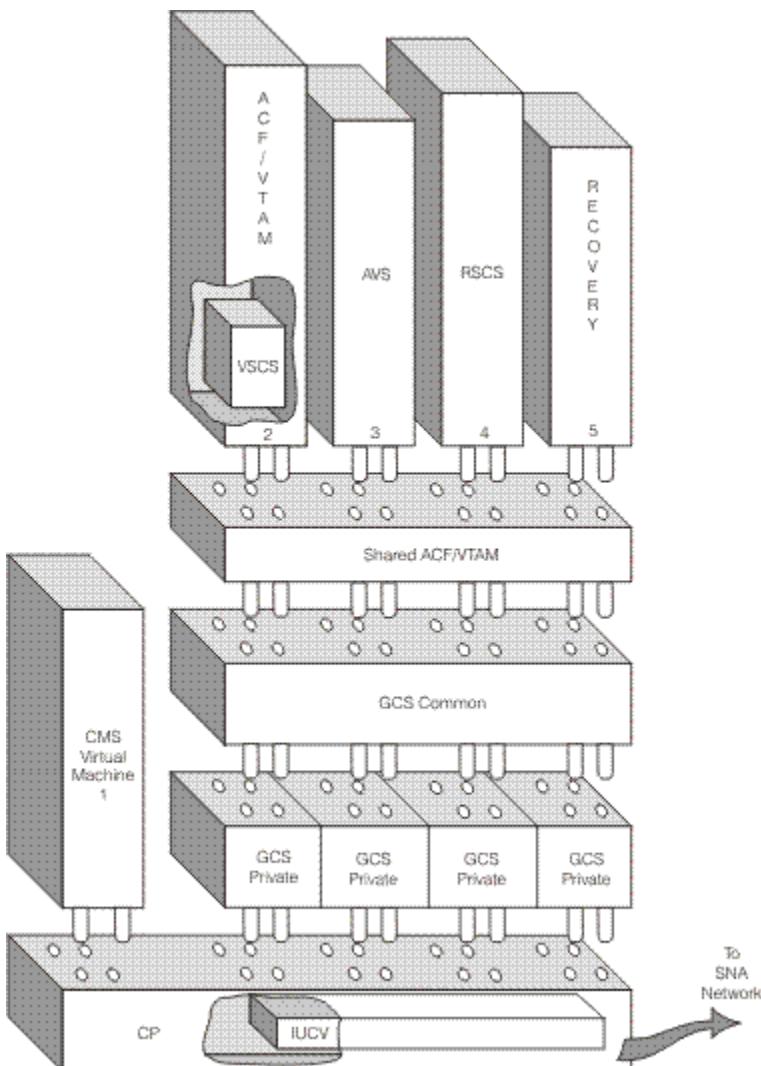


Figure 5. GCS in z/VM

This diagram shows only the conceptual relationships among the applications and saved segments in storage. Actual storage layout is different for every installation. The application space might even include two or more separate areas.

## GCS Scenario

The following scenario shows how GCS helps support native SNA communications.

Log on from a SNA terminal and IPL CMS. Neither you, as a user, nor CMS needs to know that it is a SNA terminal. CMS responds to your commands. Being an interactive system, it communicates back and forth with you through this terminal. The information exchange seems to happen easily enough. But because you have a SNA terminal, the path from your console to CMS is a complex one, involving GCS, ACF/VTAM, and SNA.

## Establish the Path Between System and Console

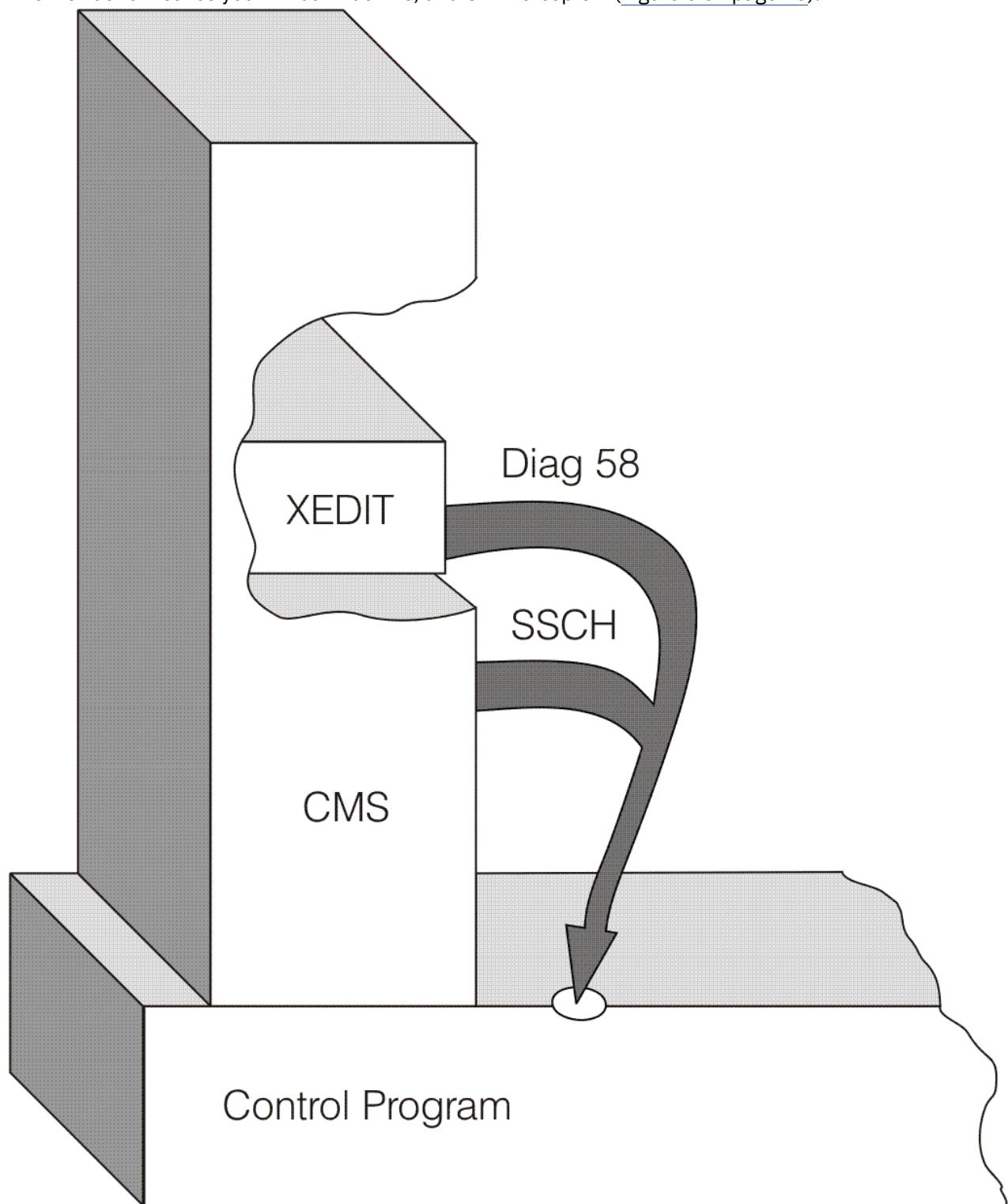
For example, CMS begins communicating with your console by issuing:

```
Start Subchannel (SSCH)
```

Or, a CMS application like XEDIT issues:

Diagnose code x'58'

The instruction leaves your virtual machine, and CP intercepts it ([Figure 6 on page 15](#)).

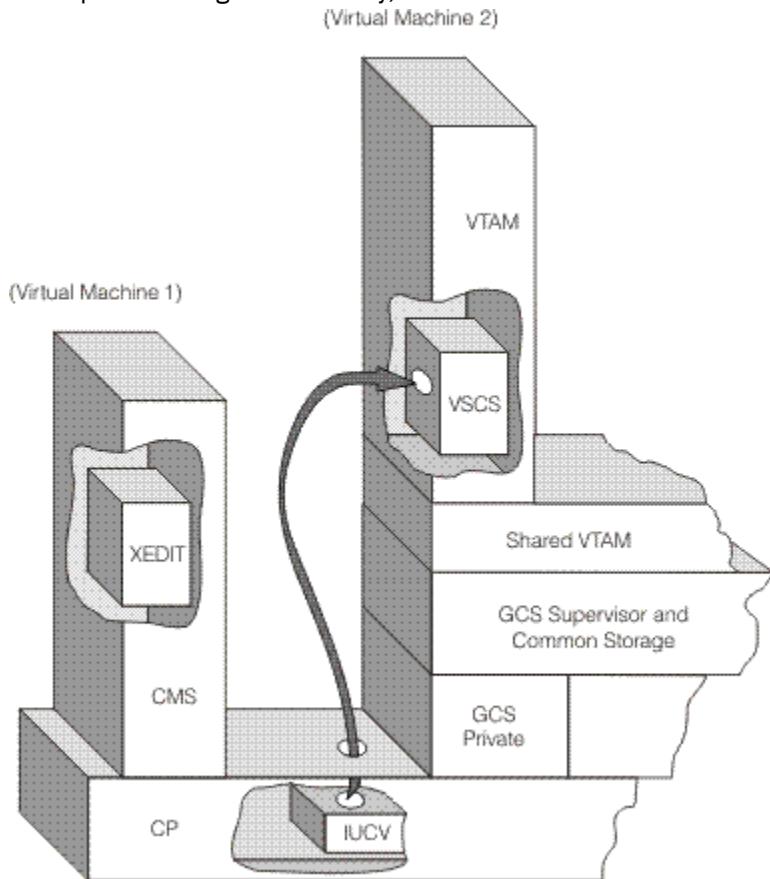


*Figure 6. CP Intercepts Instructions from the Virtual Machine*

After decoding and extracting the instruction's pertinent information, CP prepares to send data out on the network.

## Group Control System Overview

From CP, the data passes to a virtual machine running VSCS. (In the [Figure 7 on page 16](#) example, the VTAM machine runs VSCS. VSCS also may run in its own virtual machine.) The transfer from CP to VSCS takes place through a CP facility, Inter-User Communications Vehicle (IUCV).

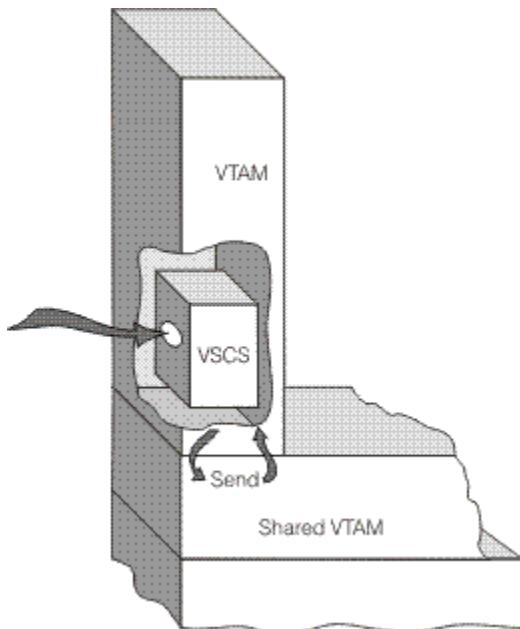


*Figure 7. Transferring Data to the Machine Running VSCS*

[Figure 7 on page 16](#) shows the VTAM virtual machine running on GCS. In a z/VM system with SNA terminals, this machine must be running ACF/VTAM Version 3 because:

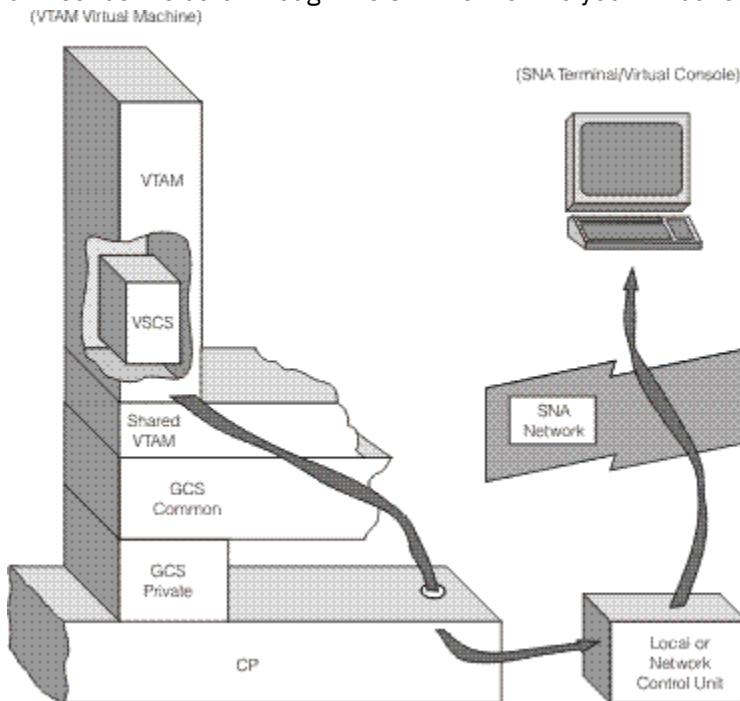
- ACF/VTAM allows a VSCS component to run in the VTAM virtual machine (as in this example).
- ACF/VTAM provides a SHARED VTAM interface that lets all other machines running in this GCS group communicate with ACF/VTAM and the rest of the network.

[Figure 8 on page 17](#) shows what happens after CP sends data to the VTAM machine. VSCS receives it, processes it into a physical screen image, and issues a SEND macro. The SEND macro finally gives control to VTAM.



*Figure 8. Path of Data Moving through the VTAM Machine*

From VTAM, the information travels toward your terminal (Figure 9 on page 17). Output instructions are relayed from VTAM to GCS, from GCS to CP, and from CP to the network or local control unit. The control unit sends the data through the SNA network to your virtual console.



*Figure 9. Data Traveling from VTAM to the Virtual Console*

## GCS Task Management

GCS provides multitasking services for multiple active tasks, as opposed to CMS which supports only one active task at a time.

- **What is a task?**

A task is a single piece of work to be done, usually an independent routine. A program running in a GCS machine can spawn a series of tasks, each with a specific job to do. Together, these tasks contribute to the program, letting it accomplish its overall assignment.

- **What is Multitasking?**

A program can have tasks that belong to it, and those tasks can have numerous subtasks. With GCS, a single program can have many tasks active at one time, although the processing unit can process only one task at a time. Multitasking is the act of managing system resources for all those tasks as they *line up* to run.

This multitasking capability provides for more programming flexibility and better system performance. GCS provides services that allow applications to control and manipulate tasks within the system. A GCS task represents a unit of execution and has associated with it a task identification number (id) from 1 to 65535 and a task dispatching priority from 0 to 255 with 255 being the highest priority. The order of execution of tasks is controlled by the GCS dispatcher and GCS tasking services.

## Adding and Discarding Tasks

A GCS program starts with one initial task. And that initial task can add on additional subtasks using the ATTACH macro. Those subtasks, in turn, can add more subtasks of their own. What results is a task hierarchy like that shown in [Figure 10 on page 18](#). All those tasks belong to one GCS application program. They compete with each other for an opportunity to run in that application's virtual machine.

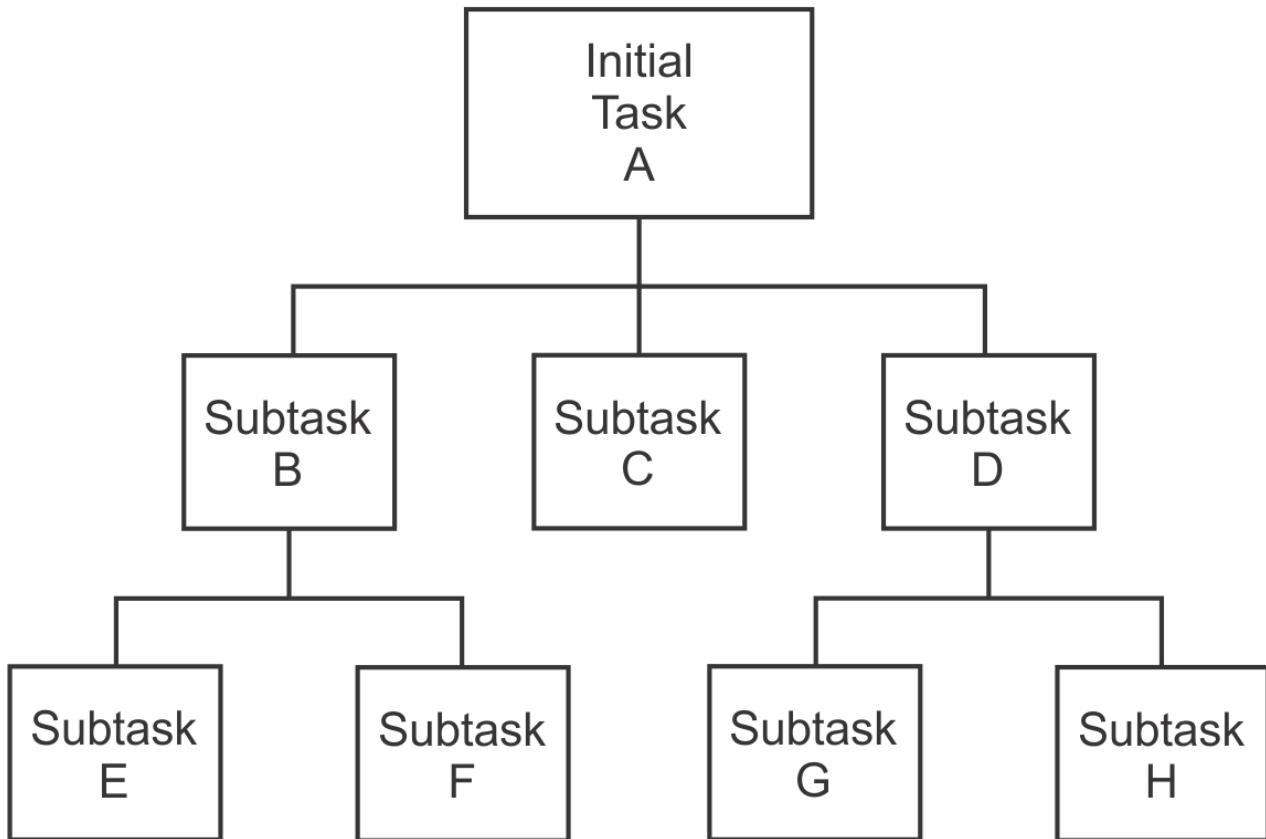
Tasks use the following two macros for adding and discarding subtasks:

**ATTACH**

To add on a subtask

**DETACH**

To get rid of a subtask.



*Figure 10. Diagram of a Task's Family Tree*

## Dispatching Tasks

To help GCS set up a task hierarchy, each task has a 2-byte task ID and a 1-byte dispatching priority number. Tasks that want to run first identify themselves with the task ID. And then, GCS sets the order of dispatching according to the 1-byte dispatching priority number.

Tasks themselves determine dispatching priority numbers. Parent tasks assign priority numbers to newly created subtasks. Subtasks' priorities can be the same, higher, or lower than their parents'. To change an existing priority assignment, tasks must call the CHAP macro. CHAP works only for a:

- Task that wants to change its own priority
- Parent task that wants to change the priority of one of its attached subtasks.

For more information, see [“CHAP” on page 187](#).

Tasks with the largest dispatching priority numbers have the highest priority. Usually, dispatching follows the simple rule:

- High priority before low.

But exceptions do occur:

- When tasks have equal priority, the task dispatcher will keep timing information about the running task. If the running task exceeds the time limits the task dispatcher will switch to a ready task of equal priority.
- When the highest priority task cannot run, GCS dispatches the next-highest, runnable task.

Otherwise, when a task does get dispatched, it maintains control:

- While disabled for interrupts
- Until a higher priority task becomes ready to run
- Until it terminates
- Until it issues a WAIT.

## GCS System Tasks

When a GCS segment is IPLed, it begins with two system tasks: console and commands. The function of the console task is to control the GCS console. It has a task ID of 1 and a dispatching priority of 255 (X'FF'). When a command is entered from the GCS console, the console task processes the attention interrupt and examines the data. If it is an immediate command (for example, HX), the command is executed immediately. If it is not an immediate command, the console task issues a POST macro for the commands task to process the input. The console task then issues the WAIT macro and waits for the next console input.

The GCS commands task has a task ID of 2 and a dispatch priority of 251 (X'FB'). All nonimmediate commands issued from the GCS console (including OSRUN) are run under the commands task. Resolution of the command names, such as commands and EXECs, are the same as in CMS. If the command was not found, it is passed to CP through a Diagnose X'08' for execution. If it is still not found, an error message is returned to the console.

## Task Dispatching and Multi-tasking Services

The GCS dispatcher is a priority-based dispatcher, that is, a higher priority ready task is executed before a task with a lower priority. The higher the dispatch priority number of the task, the higher its priority. When multiple tasks have equal dispatch priorities, they are dispatched in a *round-robin* format. The GCS dispatcher also has a timing facility, not related to the GCS timer facilities, such as STIMER, that will keep track of how long a task has been running. If a task has been running for more than 300 milliseconds and there are other tasks ready on that priority level, a task switch will occur when the dispatcher tries to redispatch the task. A task will not be interrupted by the dispatcher when its time is exceeded. A GCS task can give up control in one of the following ways:

- A higher priority task becomes ready

- The task issues the WAIT macro
- The task's time slice is exceeded when the dispatcher tries to redispatch the task.

If a GCS task is running disabled, it will always regain control even if a higher priority task is ready.

Each task in GCS is represented by a *task block*. When a dispatcher context switch occurs, the information about the active task (registers, PSW and so forth) is stored in the task block. The task block points to a stack of *state blocks* which contains information regarding the state of programs executing under that task.

Tasks can be controlled by using GCS task-based service macros. The ATTACH macro creates a task which can have an exit specified to be run at task termination (ETXR), have data passed to it (PARAM), and have a dispatch priority which can be different from the parent task (DPMOD). A task may be created as an independent application (JSTCB=YES). This type of task, and all of its subtasks, will remain active after a command has completed. This is used by applications, such as VTAM and Remote Spooling Communications Subsystem (RSCS) to be loaded only after into GCS. Only a task that is a direct subtask of the commands task can be attached as an independent application. Only a program running under the commands task may attach an independent application.

The DETACH macro cleans up tasks after they have terminated. The task id assigned by the ATTACH macro must be used to identify the task that is being detached. The WAIT and POST macros control task synchronization. A task that issues a WAIT, has its execution suspended until an event (signified by a corresponding POST) is completed. This is the primary method by which tasks control the flow of execution in GCS. Another way to control the execution of tasks is with the CHAP macro. This will change the dispatch priority of a task thus, altering its order of execution.

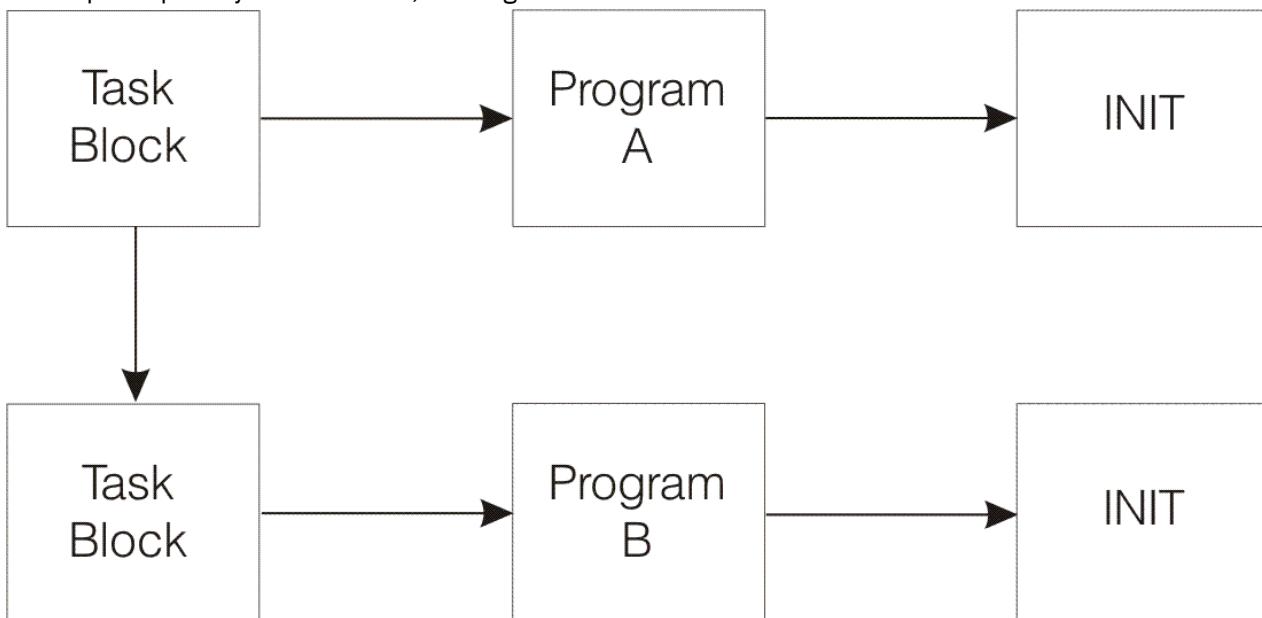


Figure 11. Task Block Dispatch Priority

Program A issues an ATTACH for Task B as depicted in Figure 11 on page 20. The resulting structure is that Task B is a subtask of Task A. When Task B is the highest priority ready task, the GCS dispatcher will give control to Program B. The INIT program in both tasks is generated by GCS for each task. It initializes the task and cleans it up when the task terminates.

## Coordinating Dependent Tasks

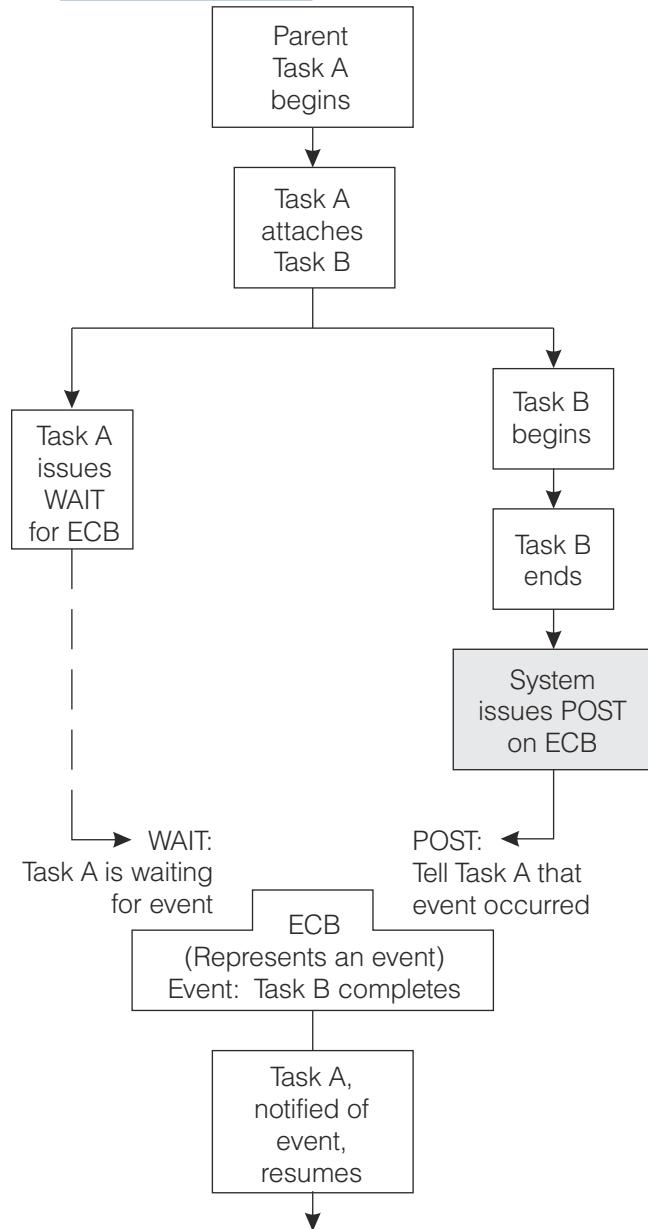
Often, tasks depend on each other to get work done. For instance, one task might have to stop running until a second task provides additional information or service. When that event occurs, and the first task resumes, the two tasks have synchronized.

Events are important reference points for coordinating or synchronizing tasks. Tasks plan their actions around events by using Event Control Blocks (ECBs). An ECB is a word of storage that represents some event.

The two task management macros that use ECBs are:

- **WAIT** - Suspends the task until some event occurs
- **POST** - Notifies the task that some event has completed.

For example, when a task has to wait for an ECB, it is suspended until a POST macro is issued for that same ECB. A task can wait for a whole list of ECBs. When any one of them gets posted, the task resumes. See [Figure 12 on page 21](#).



*Figure 12. How Tasks Can Use WAIT and POST Macros*

WAIT and POST work only among tasks in the same virtual machine. For more information on these macros, see [“WAIT” on page 365](#) and [“POST” on page 314](#).

### Coordinating Shared Resources

Sometimes tasks have to synchronize their use of a *resource*. A resource is something (perhaps a facility or service) that applications in a particular virtual machine need to use. Its assigned resource name has significance only within that virtual machine, and then only to the applications programmed to use it. When many tasks have to share such a resource, they coordinate their time using:

- **ENQ** - Enqueues a request for control of a resource
- **DEQ** - Releases previously requested resource.

With an ENQ request, a task provides a resource name, identifies the resource it wants to use, and specifies whether it can share that resource. If a task cannot share the resource, it enqueues in *exclusive mode*, requesting exclusive use of that resource. If it can share, it enqueues in *shared mode*. Sometimes tasks have to wait so they each can take separate turns using a particular resource. In other cases, many tasks share one resource at the same time.

If a task has enqueued a resource in exclusive mode, any other task that issues ENQ on that same resource must wait until the first task finishes. After the first task issues DEQ, the second can take its turn. In addition, if one or more tasks are already enqueued in shared mode, a new task cannot gain control in exclusive mode. It will be forced to wait until the others finish with the resource in shared mode.

ENQ and DEQ apply only to tasks running in the same virtual machine. For more information on ENQ and DEQ, see “[ENQ](#)” on page 213 and “[DEQ](#)” on page 204.

### Terminating Tasks

Task termination has two facets:

1. What makes tasks terminate:

NORMALLY:

A task ends normally for one reason:

- It finishes its work and returns control to the GCS supervisor. The supervisor or an exit routine (specified with the GCS TASKEXIT macro) cleans up any resources the task was using.

ABNORMALLY:

A task terminates abnormally (abends) because:

- It requests an abnormal termination with the GCS ABEND macro. When a task specifies abend with the DUMP option, it receives a dump of its virtual machine.
- A parent task above it terminates. (When a parent task terminates, its immediate subtasks and all their attached subtasks terminate too.)
- Its parent task orders it terminated with a DETACH macro.
- The virtual machine operator cancels the entire application program.
- The GCS supervisor cannot provide a requested service.

The supervisor or an exit routine (specified with the TASKEXIT or ESTAE macro) cleans up any resources the abended task was using.

2. What happens because tasks terminate:

a. Tasks call exit routines.

Programs running in authorized machines can set up termination routines with the TASKEXIT macro. These routines reside in shared storage so that they can serve any machine in the group. When any task terminates, normally or abnormally, the GCS supervisor calls these exit routines.

Not all terminations are final. GCS has procedures that permit tasks to appeal abnormal terminations. Tasks can set up exit routines that are local to their own virtual machine with the

ESTAE macro. These routines will clean up resources and decide whether to uphold the abnormal termination. ESTAE lets an exit routine, which you have written:

- Perform some predetermination processing
- Diagnose the cause of the abend
- Continue normal processing at some retry point
- Continue termination.

During the exit, an abended task can ask the GCS supervisor to let it recover control and continue executing. GCS will call this ESTAE exit for any abend, unless certain circumstances prevail. For more information see “ESTAE” on page 223.

b. GCS *cleans up* resources when tasks terminate:

- Closing any files the task opened
- Releasing any storage the task used
- Releasing any locks the task held
- Severing all IUCV paths the task established
- Canceling any timer intervals the task set
- Canceling resources the task requested through ENQ macro
- Closing General I/O devices the task opened and unlocking any locked pages of storage
- Canceling any replies from the operator that the task requested through the WTOR macro
- Subtracting the task's modules from running totals in storage (program load count and use count)
- *Undefining* any commands you defined with LOADCMD (only if you terminated the task with an HX command)
- Deleting any NAME/TOKEN pairs associated with the TASK

## Abend Processing

ABnormal END (ABEND) is the processing that occurs when an error, either from the system or an application, is detected. The task which the error occurred in must be terminated and *cleaned-up*. ABEND processing first checks for any *critical bits* being set on in GCS. This would indicate that the error occurred during a critical system path (for example, changing storage management pointers). If this occurs, a dump of the system is taken and the virtual machine is reset. GCS next checks to see if any Extended Specify Task Asynchronous Exit (ESTAEs) were declared for this task. If so, each exit is run in the reverse order in which it was declared (most recent run first). Upon return from the exit, a check is made to see if the exit specified a retry point. If so, ABEND will return control to the specified retry routine. If no retry routine was specified, processing is continued.

If the DUMP parameter was specified on the ABEND macro, it is taken now. If the virtual machine receiving the dump is authorized, the entire virtual machine (including common storage) is dumped. If the virtual machine was not authorized, only the storage pertaining to that task is dumped.

ABEND next tries to clean up all of the subtasks of the ABENDING task. Resource managers are run for each task to release system resources such as storage, locks, and timers. Each of these subtasks is detached by ABEND. When this is complete, the same resource managers are run for the ABENDING task. The ABEND message is then sent to the GCS console. If the ECB parameter was specified when the task was attached, the task that attached the abending task is posted. Finally, control is returned to the dispatcher. The task that ATTACHED the ABENDED task must still issue the DETACH macro to get rid of the ABENDED task.

## General I/O (GENIO) Facility

GCS General I/O is a function that allows a program to drive an I/O device that is defined to a virtual machine. GENIO then becomes an interface between the I/O system and applications. The program may run channel programs on the device, and process I/O interrupts through a user exit routine. All

device specific support, including device-level request queuing and error recovery, must be provided by the program. When a task terminates, GENIO will clean up outstanding I/O requests, unlock pages and close any open devices. An application can open a device and GCS will then notify the application of I/O operations completing or asynchronous interrupts. GENIO will then schedule an asynchronous exit on the task which issued the GENIO open. For example, the GCS console. The GCS console task, at initialization time, issues a GENIO open for the GCS console. All I/O interrupts from the console are reflected through an exit to the GCS console task.

GCS GENIO is unique in several significant ways. GCS provides a method of specifying I/O exits and will control the stacking of interrupts. User defined exits are executed to notify an application when an I/O operation is complete or an asynchronous interrupt has occurred.

## GCS Real I/O

Another feature of GENIO is the ability to process real channel programs on dedicated devices. This is done by using the STARTR (start real) function and may only be issued by authorized applications. This support uses the CP Diagnose X'98'. The application must provide a channel program which resides in real storage and uses real storage addresses in the Channel Control Words (CCWs). A real page address can be obtained by using the GCS page lock (PGLOCK) and page unlock (PGULOCK) macros. The PGLOCK macro will also use Diagnose X'98' to lock a real page of storage and return the address to the program. To use the STARTR functions the:

- Program must be running in supervisor state.
- Program must be executing with a PSW key other than 0.
- Virtual machine must be authorized in the CP directory to use Diagnose X'98'. This is accomplished by specifying OPTION DIAG98 on the CMS DIRECT command.

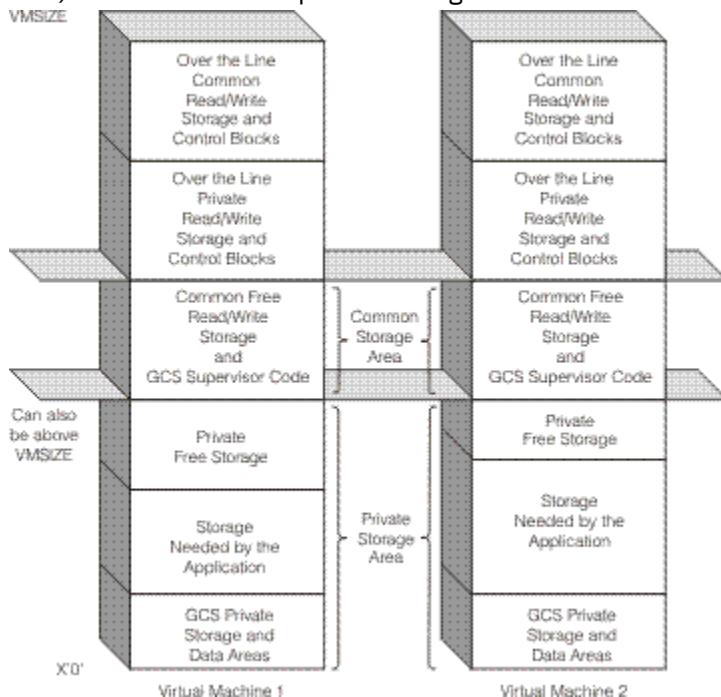
The benefit of GENIO STARTR is that an authorized application can lock pages in storage, build channel programs and receive a performance gain by avoiding the overhead of CP doing CCW translations and page locking for each I/O request.

# Chapter 2. Planning for GCS

## Planning GCS Storage Layout

Besides calculating how much storage you need (see “[Calculating Storage Requirements](#)” on page 25), you also have to decide where to locate low and high common storage in relation to low and high private storage.

Common storage must begin at the same address for each group machine, an address determined by the *largest* application storage area needed in the group. [Figure 13 on page 25](#) shows how the end of the larger application area in Virtual Machine No. 2 (and the private free storage that extends to a multiple of 4KB) determines where private storage ends and common storage begins.



*Figure 13. Ideal Locations of Common and Private Storage in Two Virtual Machine Group Members*

By locating common storage above the largest application storage area, just inside the virtual machine's highest address (VMSIZE), you avoid fragmenting private storage.

**Note:** GCS common storage must be located within the VMSIZE of both the VTAM machine and the recovery machine. That way, both machines will be the same size. And, in case of an abend during a real I/O operation, the recovery machine (with DIAG98 in its directory entry) will be able to unlock any pages of storage locked by VTAM.

GCS common storage may exist outside the VMSIZE of other machines in the group.

## Calculating Storage Requirements

The planning process involves calculating your GCS system's storage requirements. Later, you will use your findings to fill out the fields in the GROUP EXEC.

Here are guidelines to consider when reserving storage space for each of your GCS virtual machines:

### Reserve space for private storage:

1. How much for GCS private storage?

Eight pages of GCS private storage should be enough to hold your group configuration file along with certain control blocks and work areas that only the GCS supervisor has read/write access to. These pages are 0-7 in the DEFSYS command.

Suppose you have a configuration file with five authorized user IDs and two shared segments. Such a file, together with the supervisor's control blocks and work areas, would fit within the eight-page (32KB) estimate.

**Note:** The space needed by the configuration file is your only variable here. (most configuration files will take up less than 1KB of storage.) However, if you have an exceptionally large configuration file, with long lists of authorized user IDs and shared segments, you may need more than eight pages of GCS private storage.

If you need more pages, you must make sure the entry point GCTBUFND in the module GCTBUF stays within the pages that are saved.

### 2. How much private storage space for applications?

Each machine will have a different-sized application space because different applications have different requirements.

You have to reserve application private storage space in 4KB increments. However, most application sizes may not be even multiples of 4KB. If the largest application code is not a multiple of 4KB, you must round up to the next multiple of 4KB and use this as your private storage size. Therefore, remaining space, between application code's end and the last 4KB boundary is extra private free storage.

### 3. What size trace table do you need?

The GROUP EXEC, used at build time, places the internal trace table in private storage by default. If you use this default, you have to decide how much *history* you need in your storage dumps. The more applications you run and more activity you require of the GCS supervisor, the larger you need to make your trace table. The default size is 16KB.

GCS private storage (8 pages of 4KB each)	32768	bytes
+ Largest contiguous block of storage needed (determined by largest application run)	-----	bytes
+ Amount needed to round up to 4KB	-----	bytes
+ Size of the trace table if in common storage. (Default is 16KB) (The trace table can be placed in common storage by overriding the default in the GROUP EXEC.)	-----	bytes
<hr/> <b>Total</b> (Lowest possible beginning point for Common storage)	-----	bytes

Some considerations when planning private storage include:

- Largest contiguous block of storage needed for application code
- Total amount of storage needed
- Location of GCS
- Location of other shared segments and DCSSs
- Size of the configuration module.

### Reserve space for common storage:

Your common storage must be large enough to hold GCS supervisor code, common free storage space, a trace table if specified in common storage, and control blocks required for each group member. Specifically, here is what to consider:

1. *How much space does the GCS supervisor code take up?*

Approximately 312KB out of low common storage and 104KB out of high common storage. This amount should remain constant. Double check it, in your load map, after you have built your GCS system.

2. *How much common free storage do each of your GCS applications require?*

To exploit the ESA environment common free storage should be both above and below the 16M line. The minimum and default for high common is 1MB. (For the exact amounts of common free storage needed, see each application's associated planning book.)

3. *How many virtual machines will you have in your virtual machine group?*

Each group member takes up one 24-byte control block in common storage. So, the more members you have, the more blocks you will have in common storage.

4. *What size trace table do you need?*

The GROUP EXEC, used at build time, places the internal trace table in private storage by default. If you plan to override this default and place the internal trace table in common storage, you have to decide how much *history* you need in your storage dumps. Trace table storage will be needed for all virtual machines in the group. The more applications you run and more activity you require of the GCS supervisor, the larger you need to make your trace table. The default size is 16KB.

Use the following formula to calculate your common storage size:

Common free storage (Total needed by all group applications)	_____ bytes
+ Size of the supervisor (approximately 340KB)	340KB _____ bytes
+ Control blocks (VMCBs) (One 24-byte block for each group member)	_____ bytes
+ Size of the trace table (Default is 16KB) (The trace table is placed in private storage by the GROUP EXEC. If you want the trace table in common storage, you must save space for it and specify that the trace table is to go to common storage when the GROUP EXEC is run.)	_____ bytes
<hr/> Total (low common)	_____ bytes
High common	_____ bytes

## Preparing to Build Other Saved Segments

The process for setting up a shared segment for use with GCS involves making decisions like those you make when setting up and defining other shared segments or saved systems.

CP supports both regular segments and segment spaces. A segment space is a saved segment composed of up to 64 member saved segments referred to by a single name. A segment space occupies one or more segments. It begins and ends on megabyte boundaries. A user with access to a segment space has access to all its members. Loading more than one segment into a segment space, when segment size allows, makes better use of storage space. Unused storage in regular segments or in segment spaces is not available for other GCS use.

For more information about segments and segment spaces, see [z/VM: CP Planning and Administration](#).

Definition and use of segment space is controlled by the CP DEFSEG command. For more information about this command, see [z/VM: CP Commands and Utilities Reference](#).

## Shared Segments Recognized by GCS

For a user-defined shared segment to be usable by GCS the various entry points it contains must be defined in a directory. This directory contains the name of each entry point in the saved segment mapped to its address. Use the CONTENTS macro to create such a directory. See “[CONTENTS](#)” on page 197. To build a shared segment for GCS, you must:

1. Enter a DEFSEG command for the saved segment.
2. Create a directory for the segment with the CONTENTS macro. This directory will reside in its own module and will contain the name and entry point of every routine in the segment.
3. Load this directory module, followed by all the other modules you want into the segment, at the desired location in storage. Use the CMS LOAD command with the ORIGIN option.
4. Set the segment's storage key using the SETKEY command.
5. Enter the CP SAVESYS command to actually save the segment you have built.

## Private Segments for Applications

It is also possible to use a segment that does not contain an entry point directory created by the CONTENTS macro.

To be used, the segment must:

- Have a starting address larger than the virtual machine size
- Be manually loaded by the application
- Not be named in the GCS configuration file
- Not overlap any segment named (or defaulted) in the GCS configuration file.

To build these shared segment, you must:

1. Enter a DEFSEG command for the saved segment.
2. Load the modules you want into the segment, at the desired location in storage. Use the CMS LOAD command with the ORIGIN option.
3. Set the segment's storage key using the SETKEY command.
4. Enter the CP SAVESYS command to actually save the segment you have built.

## Making VSAM Available to GCS

---

If you want your GCS applications to use VSAM data sets, you must install the VSE/VSAM product. You add VSAM to your system configuration as another shared segment. After installation, both your CMS and GCS applications can access VSAM data sets and can share the same CMSBAM and CMSVSAM segments. However, GCS does not require the CMSDOS and CMSAMS segments for VSAM. For information on how to build and install VSAM, see the *VSE/VSAM Program Directory*, for your release.

## Authorizing Access to Supervisor State

---

You can control access to supervisor state by revising your list of authorized user IDs with the GROUP EXEC. Load the GROUP EXEC, and go to the screen marked *Authorized VM User IDs*. Follow the directions there for adding, changing, or deleting entries. By doing that, you can provide or deny user IDs access to supervisor state and authorized GCS functions. After making changes with the GROUP EXEC, generate an updated GCS nucleus.

## Authorizing Access to GCS

---

After you have installed your GCS segment, you can still control who has access to it. On one level, you decide which user IDs can IPL the GCS system; to be authorized to use the system, (when the segment in the DEFSYS command is defined with a RSTD parameter), the user must have a NAMESAVE control statement in the directory.

## Authorizing Commands for Virtual Machines

---

If you add or delete authorized user IDs with the GROUP EXEC, you will probably need to change their **privilege classes** too. For example, to protect GCS code, you have to limit what CP commands unauthorized user IDs have access to. On the other hand, you might want certain authorized user IDs to have access to all available CP commands. By changing the privilege class specified on user IDs' USER control statements in the directory, you affect which CP commands they can use.

You should redefine your system's privilege classes so that two like these are available:

### 1. A privilege class for authorized user IDs

This class should be for GCS user IDs that need to use the CP debugging commands BEGIN, DISPLAY, DUMP, STORE, TRACE, and VMDUMP.

### 2. A privilege class for unauthorized user IDs

This class should be for GCS user IDs that do not need to use debugging commands. It should give access to all current Class G commands except BEGIN, DISPLAY, DUMP, STORE, TRACE, and VMDUMP.

## Authorizing Machines for Real I/O

---

You choose whether your GCS machines will use real channel programs to drive real, attached I/O devices. The recovery machine, for instance, should be authorized to use real I/O. To authorize a virtual machine for real I/O, you have to change your CP directory and specify the parameter DIAG98 on the OPTION control statement. For more information on creating directories, see [z/VM: CP Planning and Administration](#).

## Using AUTOLOG Functions

---

To use the CP AUTOLOG function for GCS, you need to make a CP directory entry for each user ID you want logged on automatically.

The directory entry for GCS should look like this:

```
IPL GCS PARM AUTOLOG
```

where

```
GCS
```

is the name given to your GCS system.

**Note:** *PARM AUTOLOG* will not work properly if you try to enter it from an *IPL* instruction on your console's command line. Use *PARM AUTOLOG* only in directory entries.

If one user ID has this entry in its directory, a second user ID, having a privilege class of A or B, can log on the first one automatically with the CP AUTOLOG command. For more information about AUTOLOG, see [z/VM: CP Commands and Utilities Reference](#).

## Using a PROFILE GCS File

---

You can identify load libraries and initialize GCS applications automatically in the PROFILE GCS. When you IPL your GCS system:

1. Saved segments (specified in your GCS configuration file) are linked to your virtual machine.
2. Disks are accessed.
3. Disks are searched for a file of name and type PROFILE GCS, and if there is one, it executes. (PROFILE GCS must contain REXX code because the REXX/VM interpreter is the facility that processes it.)

By setting up enough PROFILEs, you can automate logging and initialization procedures for most of your virtual machine group. Because the recovery machine must be the first to IPL GCS, you could give it a PROFILE that would automatically log on all other group members that have IPL GCS PARM AUTOLOG specified in their CP directory entries. Be sure to assign the recovery machine a privilege class of either A or B so that it has authorization to issue the CP AUTOLOG command.

By defining an AUTOLOG1 user ID in your CP directory, you can have it automatically log on the recovery machine as well. For more information, see [z/VM: Running Guest Operating Systems](#).

If you set up a PROFILE GCS, you cannot prevent it from executing. But if you find a problem with it, you can interrupt and stop it with the:

- HX (halt execution) command
- BREAK or PA1 key.

After that, you can go back to CMS and change your PROFILE GCS file.

## Preparing CP Directory Entries

---

You may need to update your CP directory and prepare new entries there for user IDs that will use GCS. To see a working sample of a directory entry for a recovery machine user ID, see your System DDR. For more information on how to create directory entries with the Directory program, see [z/VM: CP Planning and Administration](#).

## Operation

---

*Operating* GCS involves initializing GCS, starting and stopping programs, replying to messages, and querying information. This section describes each activity. The commands are described elsewhere in this book.

### Initializing GCS (How to Join a Group)

*Initialization* is simply the act of loading (IPLing) your GCS system. It also means the same thing as *joining a virtual machine group* or *IPLing the GCS supervisor*.

To join a group, enter:

```
ipl gcs
```

To leave a group, do one of the following:

- Log off
- IPL another system
- Enter one of these CP commands:

```
SYSTEM RESET  
SYSTEM CLEAR  
DEFINE STORAGE  
SET MACHINE
```

You can enter the IPL command (with the name of your GCS saved system) from your virtual machine console, or you can automate the loading of your GCS system by using the CP AUTOLOG and auto-IPL

procedures described under “[Using AUTOLOG Functions](#)” on page 29. For example, if you named your GCS system GCS at build time, you would enter:

```
ipl gcs
```

The system will respond with a system ID message (if you specified one with the GROUP EXEC) and a generate message. For example:

```
GROUP CONTROL SYSTEM
Generated at mm/dd/yy hh:mm:ss
GCTACC423I A (0191) R/W
Ready;
```

For more information on how to specify a system ID in GCS, see the “[Changing GCS Default Definitions](#)” on page 512.

When you initialize GCS, any other shared segments you identified with the GROUP EXEC become linked to your virtual machine, and disks are accessed as shown by [Table 4 on page 31](#).

*Table 4. Automatic Disk Access at IPL*

Device Type	Virtual Device Address	Access Mode
Primary Disk read/write	191	A
User Disk read/write	192	D

After the disks are accessed, GCS searches them for a PROFILE GCS file and, if you have one, processes it. PROFILE GCS resembles the PROFILE EXEC in CMS and is described in “[Using a PROFILE GCS File](#)” on page 29.

After you have IPLed GCS and have the proper disks accessed, you can enter GCS commands to assign files and start applications. For example, these are the commands you would use to start RSCS operations:

```
global loadlib rscs
filedef config disk rscs config *
loadcmd rscs dmtman
```

You may enter these commands from your virtual machine console or place them in a PROFILE GCS to run automatically at IPL time.

## Starting and Stopping Programs

If you want a program to run on GCS, you have two choices:

1. Write your own.
  - a. Write and compile it or assemble it using CMS.
  - b. Put the resulting text files in a load library using the CMS LKED command.
  - c. IPL your GCS segment.
  - d. Use the GLOBAL command to identify the load library where the program resides.
  - e. Run and *debug* the program using GCS commands. (If you make any corrections to the program's source code, you have to do them using CMS and then reload the program in its load library.)
2. Identify one that already resides in a shared segment. The program should be listed in the segment's directory—a directory created by the CONTENTS macro when the segment was built.

You can start programs in your virtual machine by entering:

1. The OSRUN command (or the name of an exec that will issue OSRUN)

Use the OSRUN command to start programs that you want to load and give control to. When you enter OSRUN to start a program, GCS will not process any other commands (except immediate commands)

until the program ends. The system will not accept other commands because it allows only one active command at a time. So, OSRUN remains the active command while the program is running.

The program will stop automatically without prompting.

### 2. An application command (one you have defined with the LOADCMD command).

This lets you call an application that will start itself either with an OSRUN command or an ATTACH macro with the JSTCB=YES parameter. If the application's start-up module uses the ATTACH macro to start, your initial application command remains active, and you still can enter other application commands.

An application started with LOADCMD stops:

- Automatically (when it finishes its work)
- When prompted (you enter the command name you defined and include the necessary stop parameter).

If a program issued an ATTACH macro to start, it must issue the DETACH macro to stop the attached program.

You can stop programs during their execution with the HX command. HX also clears any commands defined with the LOADCMD command that are stacked and waiting to be processed.

## Replying to Messages

When a program needs to communicate with you, it can send a message to your console and request your reply. For this, the program uses a WTOR macro (Write To Operator with Reply). It may ask you, as a GCS virtual machine operator, to set up certain devices, provide data, or do some other request.

To respond to messages sent by WTOR, you enter the REPLY command. Each message you respond to will have an id number associated with it. You use this id number to route your response.

Unlike CMS, GCS lets programs continue running even when you owe them many replies. If you want to check for messages that require replies, you can enter a QUERY REPLY command. This will display the id numbers and text of all messages waiting for replies.

## Querying Information

Sometimes you need information about the status of your virtual machine. For example, you might want to see the search order of your accessed disks or to see if external tracing is active. You can find this information using the QUERY command. QUERY can report on:

- Whether internal recording of user trace events is enabled (QUERY ITRACE)
- Trace events that are enabled for recording in a spool file (QUERY ETRACE)
- User IDs of virtual machines in your GCS group (QUERY GROUP)
- The common lock's status—whether the lock is held and what user ID is holding it (QUERY LOCK)
- The id number and text of all messages waiting for a reply (QUERY REPLY)
- Any file definitions in effect (QUERY FILEDEF)
- The status and search order of accessed disks (QUERY SEARCH)
- The load libraries GCS will search for load modules (QUERY LOADLIB)
- Names of attached saved systems and saved segments (QUERY SYSNAMES)
- The current DBLs in effect (QUERY DBL)
- Information about accessed disks (QUERY DISK)
- All the entry points that were loaded by the LOADCMD command (QUERY LOADCMD)
- All entry point names and corresponding addresses that were loaded into this virtual machine (QUERY LOADALL)
- The status of the DUMP : ON, OFF or DEFAULT (QUERY DUMP)

- The status of the DUMPLOCK : ON or OFF (QUERY DUMPLOCK)
- The level of z/VM (QUERY GCSLEVEL), and service level
- The virtual machine IPOLL setting
- GCS module address
- GCS module compilation date



# Chapter 3. GCS Programming and Command Processing

## Linkage Registers

The general registers 0, 1, 13, 14, and 15 are also known as linkage registers. By convention, each register has a specific purpose:

Register	Conventional Purpose
0 and 1	Used to pass parameters to the supervisor or to a called program. Some system macros expand to include instructions that load a value into one or both of these registers. Others load the address of a parameter list into register 1. At times, the supervisor will load a parameter value into register 1 and pass it to a program that you have called.
13	Used to hold the address of the register save area provided by the calling program.
14	Used to hold the return address within the calling program. That is, the address of the executable statement just after the instruction that passed control to another program. After the calling program regains control, it is at this point that execution resumes.
15	Used to hold the entry point address of the called program. Some macros expand to include instructions that load a parameter list address into register 15, which is then passed to the supervisor. Programs also use register 15 to pass return codes to the programs that called them.

## Establishing a Base Register

In z/VM, addresses are resolved by adding a displacement to a base address. Therefore, you must establish a base register using one of the registers 2 through 12 or register 15. If your program does not use GCS macros and does not pass control to another program, then you can establish a base register using the entry point address contained in register 15. Otherwise, because both the supervisor and your program may use register 15 for other purposes, you must establish a base using one of the registers 2 through 12. This should be done immediately after saving the calling program's registers.

**Note:** Choose your base register carefully. Remember that some instructions (GCS macros included) change the contents of some registers.

## Providing a Save Area

If one of your programs passes control to another, then the former must provide a save area where the contents of its registers are saved by the program it calls. A register save area is 18 fullwords long, beginning on a fullword boundary. The following table describes the save area's structure and content.

Word	Contents
0	Used by PL/I, if applicable. Otherwise, unused.
1	If applicable, the address of the calling program's register save area.
2	The address of the current program's next register save area.
3	The contents of register 14 (the return address within the calling program).
4	The contents of register 15 (the address of the called program).

<b>Word</b>	<b>Contents</b>
5	The contents of register 0.
6	The contents of register 1.
7	The contents of register 2.
8	The contents of register 3.
9	The contents of register 4.
10	The contents of register 5.
11	The contents of register 6.
12	The contents of register 7.
13	The contents of register 8.
14	The contents of register 9.
15	The contents of register 10.
16	The contents of register 11.
17	The contents of register 12.

A called program can save the registers belonging to the program that called it by issuing either the STM (STORE MULTIPLE) assembler instruction or the SAVE macro. The

```
STM 14,12,12(13)
```

assembler instruction places the contents of all registers, except register 13, in the proper words of the save area. The SAVE macro is described in detail in the entry titled “[SAVE](#)” on page 324.

## **Example of Chaining Save Areas in a Nonreenterable Program**

```
PROGRAM1 CSECT
    STM 14,12,12(13)
    LR 12,15
    USING PROGRAM1,12
    ST 13,SAVEAREA+4
    LR 2,13
    LA 13,SAVEAREA
    ST 13,8(2)
    .
    .
    .
    SAVEAREA DC 18F'0'
```

The program uses the STM instruction to store the contents of the registers in the save area provided by the calling program. Then, the program establishes register 12 as its base register. The program goes on to save the address of the calling program's save area in the second word of another save area that it established through the DC instruction. Then, the program loads the address of the calling program's save area into register 2. Finally, it loads the address of the new save area into register 13, then stores the same address in the third word of the calling program's save area.

## **Example of Chaining Save Areas in a Reenterable Program**

```
PROGRAM2 CSECT
    SAVE (14,12)
    LR 12,15
    USING PROGRAM2,12
    GETMAIN R,LV=72
    ST 13,4(1)
    ST 1,8(13)
    LR 13,1
```

:  
:

This program uses the GCS SAVE macro to save the contents of its registers. (It could also have used an STM instruction.) The program loads the entry point address into register 12, establishing it as the base register. It then issues an unconditional GCS GETMAIN macro, requesting the supervisor to allocate 72 bytes of virtual storage for the save area from outside the program. The supervisor returns the address of this 72-byte area in register 1. The program stores the address of the old and new save areas in the customary locations and loads the address of the new save area into register 13.

## Summary of Conventions for Passing and Receiving Control

---

### **Before it passes control (return required), a calling program should**

- Place the address of its register save area in register 13.
- Place its return address in register 14.
- Place the entry point address of the program it wishes to call in register 15.
- If applicable, place the address of a parameter list in register 1.

### **Before it passes control (return not required), a calling program should**

- Restore to registers 2 through 12 and register 14 the values that were present when it received control.
- Place the address of the save area provided by the program that called it in register 13.
- Place the entry point address of the program it wants to call in register 15.
- As applicable, place the addresses of parameter lists in registers 0 and 1.

### **Immediately after receiving control, a called program should**

- Save the contents of registers 0 through 12 and registers 14 and 15 in the save area, whose address is in register 13.
- Establish a base register.
- Provide a save area of its own, unless of course it plans to call no other program.

If it is a reentrant program, then it must obtain storage for its save area outside its own storage through the GETMAIN macro. If it is a nonreentrant program, then its save area can be located with the rest of its storage.

- Store the save area addresses in the assigned locations.

### **Just before returning control, a called program should**

- Restore to registers 0 through 12 and register 14 the values that were present when it received control originally.
- Place in register 13 the address of the save area belonging to the program to which it is returning control.
- If required, place the appropriate return code in register 15. Otherwise, restore to register 15 its original value.
- If it is a reenterable program that obtained storage for its save area through the GETMAIN macro, then it must release that storage through the FREEMAIN macro.

## GCS Program Exits

---

GCS provides the ability for programs to have program exits run either on another task or when an event occurs. Asynchronous exits are programs which run on tasks, usually because of some event. These exits are scheduled by the GCS dispatcher for execution and run the next time the task receives control, possibly interrupting the usual execution of the task. Some functions which use the GCS exit facility are GENIO, IUCV support, TIMER and SCHEDEX (schedule exit). For example, if Task A in virtual machine GCSVMM1 issues a SCHEDEX macro for an exit to run on Task B in virtual machine GCSVMM2, GCS will

cause an interrupt on GCSV2 and the GCS dispatcher will schedule the exit to run for Task *B*. This exit will preempt the program currently running on Task *B*. When the exit completes, the program that was previously running on Task *B* will resume where it left off. Task *A* regains control when the notification of this exit is sent to GCSV2. Task *A* does not wait until the exit is actually scheduled to run on Task *B*.

One type of asynchronous exit that works differently is an IUCV exit for an authorized path. This path is established with the PRIV=YES parameter on the IUCVINI macro. This authorized exit is not scheduled to run on a task. Instead, for performance reasons, this exit is given control directly by the GCS APPC/VM interrupt handler. It runs as an extension of the interrupt handler, disabled and key of the caller.

Restrictions for this type of exit are that it cannot issue an SVC instruction, enable itself, or generate any kind of machine interrupt. The GCS POST service is the only GCS service supported for use within this exit. Because it is usual for this kind of routine to POST a task that the event has occurred, and use of the POST macro would cause an SVC interrupt, a special branch entry to the GCS POST routine is provided for use by these routines. For more information on the POST interface, see [“POST” on page 314](#).

For asynchronous exits that are scheduled to run on a task by the GCS dispatcher, they receive control enabled and in the same storage key as the program that established the exit. Thus an exit may be interrupted by another exit. If a program (both application program and exit) does not want to be interrupted by an asynchronous exit, the program must disable for interrupts. Only authorized (supervisor state) applications can disable for interrupts. When this happens, the disabled task will be given control by the GCS dispatcher on supervisor calls until it terminates, enables for interrupts or issues a WAIT macro.

Another type of exit is one that is authorized and shared among many tasks or virtual machines in the group. The TASKEXIT macro is an authorized function that identifies an exit in common storage and is executed anytime a task terminates in that virtual machine group. The MACHEXIT macro does the same function whenever a virtual machine leaves the group. These exits are not scheduled and dispatched but are run immediately when the event occurs. The exits run in supervisor state and in the storage key of the task that established them. These exits are deleted when the task that declared them terminates.

## GCS Commands Operation

---

GCS commands may be entered from the GCS console, an exec, or from a program. As was discussed earlier, GCS has a console and commands task to handle GCS commands. A command entered from the console is examined to see if it is an immediate command, if so it is executed from the console task. If not, the GCS commands task is POSTed and will process the command.

GCS, like CMS, allows the user to extend the number of commands by being able to define their own commands. The way that GCS supports them is through the LOADCMD command. Using LOADCMD, the user may extend the number of GCS commands with commands of their own. This is particularly useful for applications which are established as independent applications (see [“GCS Task Management” on page 17](#) for more details). When a program is established with LOADCMD, the specified program is loaded into storage and the command is added to the GCS command set. LOADCMD is similar in operation to the CMS NUCXLOAD command. The program will remain in storage until it is deleted or until the HX command is executed. Programs that are defined using the LOADCMD are only valid for the virtual machine that defined them. They cannot be used by other members of the group.

GCS commands may also be executed from programs. This can be done by using the CMDSI macro. The CMDSI macro takes the information provided by the program, translates it into the same parameter list format as the commands processing does, and executes a SVC 202 to run the command. The search order for GCS commands is the same as it is for CMS commands. This also includes the execution of GCS execs as part of GCS search order.

## Example of an Application Program in GCS

The following is an example of an application that can be set up as a GCS user command. The intention is to give the programmer an idea of how applications can be written in GCS. See the specific macro descriptions in this book for the exact formats of GCS macros and for further information regarding each of the macros used in this example.

This example shows how to define a user command LISTCMD that will be used to maintain a list in GCS storage. The LIST command options are:

**INIT**

Initializes the application environment.

**READ**

Reads data from the command into the list.

**WRITE**

Writes data to a file.

**END**

Terminates the application.

All of these options will be processed by a single module, LSTMOD. No formal programming language is used.

```
LSTMOD:
  ENTRY
  Register5 = Register1          Entry Point for module.
                                Save parameter list pointer
                                supplied in register 1.
  IF option = INIT THEN          Option in parameter list=INIT?
    IDENTIFY EP=LSTREAD          Identify entry points to GCS
    IDENTIFY EP=LSTWRITE
    LOAD    EP=LSTDATA           Load common data area (list)
    DATAPTR=Register5            Save pointer to input data
    ATTACH   EP=LSTREAD,          Attach read as a separate task,
                                DPMOD=200,
                                ECB=MAINECB,
                                JSTCB=YES
    ATTACH   EP=LSTWRITE,          Attach write as a separate task,
                                DPMOD=200,
                                ECB=MAINECB,
                                JSTCB=YES
    WAIT     ECB=MAINECB          Let LSTREAD Start.
    WAIT     ECB=MAINECB          LET LSTWRITE Start.
    RETURN
                                End initialization

  IF option = READ THEN          Option in parameter list=READ?
    DATAPTR=Register5            Save pointer to input data
    POST    ECB=READECB           Enable LSTREAD task to run.
    WAIT     ECB=MAINECB          Give control to LSTREAD.
    RETURN
                                End READ.

  IF option = WRITE THEN         Option in parameter list=WRITE?
    DATAPTR=Register5            Save pointer to input data
    POST    ECB=WRITECB           Enable LSTWRITE task to run.
    WAIT     ECB=MAINECB          Give control to LSTWRITE.
    RETURN
                                End WRITE.

  IF option = END THEN          Option in parameter list=END?
    POST    ECB=READECB           Enable LSTREAD task to run.
    WAIT     ECB=MAINECB          Give control to LSTREAD.
    POST    ECB=WRITECB           Enable LSTWRITE task to run.
    WAIT     ECB=MAINECB          Give control to LSTWRITE.
    DETACH  EP=LSTREAD           Clean-up LSTREAD task.
    DETACH  EP=LSTWRITE           Clean-up LSTWRITE task.
    DELETE   EP=LSTDATA           Release List from storage.
    RETURN
                                End END.

LSTREAD:
  ENTRY
  WORK = TRUE                   Entry point for LSTREAD.
  DO UNTIL WORK = FALSE
    IF option=INIT THEN          Set Boolean variable
      LOAD    EP=LSTDATA          option=initialization.
      LSTADDR=Register0+DATA     Get address of data area.
                                Address of data in Register 0,
                                and skip over ECBs.
      POST    ECB=MAINECB          Resume main task.
      WAIT     ECB=READECB          Wait for work.

    IF option=READ THEN          option=READ
      LSTADDR->LIST =           Move into the data area pointed
                                to by LSTADDR, the data in the
                                parameter list pointed to by
                                DATAPTR.
      DATAPTR->DATA             Increment LSTADDR pointer.
      LSTADDR=LSTADDR+NEXT        Resume main task.
      POST    ECB=MAINECB
```

```

WAIT ECB=READECB      Wait for work

IF option=END THEN    option=END
  DELETE EP=LSTDATA   Reduce use count
  WORK = FALSE        Indicate task done
END                  End Do Loop
RETURN               Terminate task,
                     MAINECB is automatically
                     POSTED by GCS.

LSTWRITE:
  ENTRY                Entry point for LSTWRITE.
  WORK = TRUE          Set Boolean variable.

  DO UNTIL WORK = FALSE
    IF option=INIT THEN
      LOAD EP=LSTDATA   option=initialization.
      POST ECB=MAINECB   Get address of data area
      WAIT ECB=READECB  Resume main task.

    IF option=WRITE THEN
      OPEN LSTDCB,OUTPUT
      WRITE LSTDATA,LSTDCB,
                 LSTDATA
      CLOSE LSTDCB       Close the file
      POST ECB=MAINECB   Resume main task
      WAIT ECB=READECB  Wait for work

    IF option=END THEN
      DELETE EP=LSTDATA
      WORK = FALSE
    END
    RETURN               Terminate task,
                         MAINECB is automatically
                         POSTED by GCS.

  END LSTMOD           End of module

```

The LSTMOD module has three sections:

- Main code. This code always gets control when the user command LSTCMD is entered. The code runs under the GCS Commands Task.
- Read code. This entry is created as a separate task under the GCS Commands Task. It is established using the ATTACH macro, with the JSTCB parameter to make it an independent application. This must be done so the task can last more than one command in the system.
- Write code. This entry is created the same as the read code.

These tasks all share a common data area (LSTDATA). This data area is a separate member of a loadlib and can be declared as:

```

LSTDATA  CSECT
MAINECB  DS   F           ECB used by main task
READECB  DS   F           ECB used by READ task
WRITECB  DS   F           ECB used by WRITE task
DATAPTR  DS   F           pointer to input data
DATA     DS   CL4080
END

```

This defines a 4KB data area that will be shared among the tasks. To get this application ready for GCS, you have to create a LOADLIB using CMS. Assuming the TEXT decks have been created for LSTCMD and LSTDATA, you can enter the following commands to create the LOADLIB.

```

LKED LSTMOD  (LIBE LSTLIB
LKED LSTDATA (LIBE LSTLIB REUS

```

The second LKED command marks LSTDATA as reusable so it will only be loaded once into storage. Every subsequent LOAD will return the storage address of LSTDATA.

Next, you would IPL your GCS segment. You could place the following commands in your PROFILE GCS file to be executed at initialization time:

```
GLOBAL LOADLIB LSTLIB
LOADCMD LSTCMD LSTMOD
LSTCMD INIT
```

These three commands define the LOADLIB to GCS, establish LSTCMD as a user command and begin the tasks with the LSTCMD INIT command. The LSTREAD and LSTWRITE tasks are initialized and waiting for work. If you enter the command:

```
LSTCMD READ This is a line of data for the list
```

The commands task gives control to the LSTMOD code, this code POSTs the LSTREAD task. The LSTREAD task reads the data into the list. LSTREAD then POSTs the commands task and the commands task completes.

To expand on this example, suppose that the virtual machine containing LSTMOD is an authorized virtual machine. This means that the LSTMOD ENTRY receives control in supervisor state when it is given control under the GCS commands task. However, this program (task) authorization is not passed along to LSTREAD and LSTWRITE tasks that are attached as subtasks of the GCS commands task. LSTREAD and LSTWRITE receive control in problem state because the parameter SM=SUPV was not specified on the ATTACH macro. Therefore, LSTMOD may use GCS authorized functions (and receives no GCS validation) although LSTREAD and LSTWRITE may not use GCS authorized functions and receive GCS address validation when requesting GCS supervisor services.

An alternative to having the common data area (LSTDATA) as a separately compiled, linked, and loaded module would be to INCLUDE LSTDATA in the LSTMOD load module and then to reference the ECBs with V-type address constants. This would eliminate the need for the GCS *LOAD EP=LSTDATA* requests and the *DELETES*. Of course, it would eliminate the capability to independently replace or update LSTDATA.

An alternative would be for LSTCMD to have its own profile EXEC. Instead of this PROFILE GCS:

```
GLOBAL LOADLIB LSTLIB
LOADCMD LSTCMD LSTMOD
LSTCMD INIT
```

it could be:

```
GLOBAL LOADLIB LSTLIB
LOADCMD LSTCMD LSTMOD
LSTPROF GCS
```

where LSTPROF GCS is the LSTCMD profile and would contain any initialization commands necessary such as LSTCMD INIT.

In this example, the command processing is serialized through the GCS commands task. The GCS commands task is put into a wait state pending completion of the execution of each subtask to complete the command. No other nonimmediate GCS commands can be processed while the commands task is in the wait state. Another method that could be used would be to pass the command information to LSTREAD and LSTWRITE so that LSTMOD could return to the GCS commands task and allow other commands to be processed. This would, of course, involve a more sophisticated way of *stacking* commands to the LSTREAD and LSTWRITE functions.

## Console and Command Support

### Communicating through the Console

Any z/VM supported terminal can be a GCS console. GCS virtual machine operators use their consoles to communicate with:

- The GCS supervisor (through GCS commands)
- Applications running in the machine (through application commands defined with the LOADCMD command.)

If the GCS supervisor or GCS applications want to communicate with a GCS virtual machine operator, they send messages to that operator's console using the WTO (Write To Operator) and WTOR (Write To Operator with Reply) macros:

### **WTO**

Writes a message to the console

### **WTOR**

Writes a message and adds a reply ID so the GCS virtual machine operator can respond. Unlike CMS, GCS lets programs keep running although you might owe them many replies.

## **Entering Commands to GCS**

You can enter commands three ways:

- Directly from the console
- From a program, using the CMDSI macro
- From a command file (exec).

A command file contains a series of GCS commands and resides on a disk. You start it with a console command, with GCS's CMDSI macro, or from another command file. PROFILE GCS (if you have one) is a particular type of command file that will run automatically when you IPL your GCS system.

Besides GCS commands, a GCS command file can contain REXX statements and functions. REXX processes these statements and, in fact, the entire GCS command file. Therefore, most REXX capabilities you are familiar with in CMS also apply with GCS. The differences with GCS are:

- REXX programs (execs) have a file type of GCS.
- Each task has its own program stack. With GCS, the program stack's primary use is for communication between execs. Execs belonging to the same program share data on the program stack. Execs belonging to different programs cannot. Moreover, because GCS console management routines bypass the program stack, you cannot stack commands there for execution.
- GCS has no external event queue (also called *terminal input buffer*). If you enter PULL, and a task's program stack is empty, you receive a message at the console asking for the necessary input.
- ADDRESS GCS replaces ADDRESS CMS. (REXX's default is ADDRESS GCS.) It acts the same as ADDRESS CMS, providing full command resolution, including execution of command files and implied CP commands.

The ADDRESS COMMAND environment acts much as it does on CMS: it executes host commands, but not command files or implied CP commands.

- You can cancel command files using HX. The commands TS, TE, and HI, which worked with REXX in CMS, have no support on GCS.
- You can call REXX programs from assembler language programs with the CMDSI macro. FILEBLK, a parameter on CMDSI, contains the address of the file block. FILEBLK is useful for executing in-storage command files, executing command files with file types other than GCS, and establishing an initial subcommand environment.
- Non-REXX programs can share variables with REXX programs through the EXECCOMM macro. GCS's EXECCOMM macro has the same capabilities as CMS's EXECCOMM service.
- GCS supports external function calls if they are written in REXX. It does not support external function libraries, like RXSYSFN, RXLOCFN, and RXUSERFN.
- GCS supports subcommand environments (ADDRESS nnnn) set up using LOADCMD. However, there is no facility like the *non-SVC fast path* for issuing subcommands.

**Note:** Execs cannot have the same names as the GCS *immediate commands* ETRACE, ITRACE, HX, QUERY, REPLY, and GDUMP. Immediate commands always execute first; therefore, an exec of the same name would never run.

For more REXX information, see [z/VM: REXX/VM Reference](#).

## Processing GCS Commands

GCS processes commands much the same way as CMS does. Some commands get:

- **Processed immediately**

If you enter commands with the CMDSI macro or any one of these *immediate* commands:

CLEAR  
ETRACE

GDUMP  
HX

ITRACE  
QUERY  
REPLY

they do not get stacked, and GCS processes them right away, even if you enter them while another command is being executed.

- **Stacked and wait their turn** (regular procedure)

All commands you define with LOADCMD and all nonimmediate commands you enter get processed serially (see “[LOADCMD](#)” on page 112). When the current command finishes executing, GCS processes the next command on the stack. The first command entered is the first command executed.

### **Commands That GCS Supports**

GCS supports commands that let you define, start, terminate, and control an application. Some commands are unique to GCS; others are existing or modified CMS commands:

*Table 5. Supported Commands*

Unique GCS Commands	GCS/CMS Commands
CLEAR ETRACE GDUMP ITRACE LOADCMD REPLY	ACCESS CONFIG DBL ERASE ESTATE EXECIO FILEDEF GLOBAL HX OSRUN QUERY RELEASE SET

In addition, you can define your own *application* commands with the LOADCMD command.

## OS Management Services

The OS *management* services described in this section are GCS services that resemble (but do not duplicate) MVS/XA functions.

## Storage Management

Each GCS machine in a virtual machine group has two storage areas: private and common. Private storage is local to an individual machine and not shared with other group members. This means that a program running in a neighboring machine cannot use or change another's private storage area. Common storage, however, is shared in a read/write fashion with all other machines in the group. Any program can use or look at nonfetch-protected information in common storage. But only authorized programs can obtain or otherwise modify storage space there.

GCS uses storage keys to prevent unauthorized storage allocation. Any program that wants to obtain storage must have a PSW key (bits 8 through 11 in the PSW) that matches the storage key of the address range in question. Unauthorized programs, for example, have PSW keys of 14. Therefore, they cannot obtain GCS common storage that has a storage key of 0 (zero).

### ***Obtaining Storage***

A program or task that runs in a GCS virtual machine can obtain or release storage space as the need arises. It does this using GCS's GETMAIN and FREEMAIN macros. With GETMAIN, the task requests a certain-sized block of storage. GCS allocates the space and passes the block's address along to the task. Later, when the task no longer needs that space, it issues the FREEMAIN macro and tells what block it wants freed.

When a task requests a certain size of storage with GETMAIN, it also can request other storage characteristics by specifying a subpool. A subpool is a number between 0 and 255. This number characterizes storage as:

- Private or common
- Fetch-protected or nonfetch-protected
- Task-related (automatically released when the task ends) or persistent (retained after the task ends).

### ***Assigning Storage Keys***

When allocating storage, the GCS supervisor assigns the address range a storage key that matches the requesting task's PSW key. There are 16 possible storage keys for different types of code. A storage area's key depends on what type of code it contains:

<b>Key</b>	<b>Type of Code</b>
<b>0</b>	Saved segments and reentrant code (including GCS common storage and other shared code)
<b>1-13</b>	Authorized (privileged) applications
<b>14</b>	Unauthorized (nonprivileged) applications (also the starting key for authorized applications)
<b>15</b>	VSAM and BAM shared segments

### ***Switching Keys***

A program can obtain storage only in the key of the PSW that it is running in. Authorized and unauthorized GCS programs both start out with the same PSW Key 14. Thus, unauthorized programs can secure only fetch-protected storage in Key 14. Authorized programs, on the other hand, can allocate storage in any key, including both fetch-protected and nonfetch-protected common storage.

An authorized program, running in supervisor state, can obtain storage in a new key by changing its PSW key. This involves:

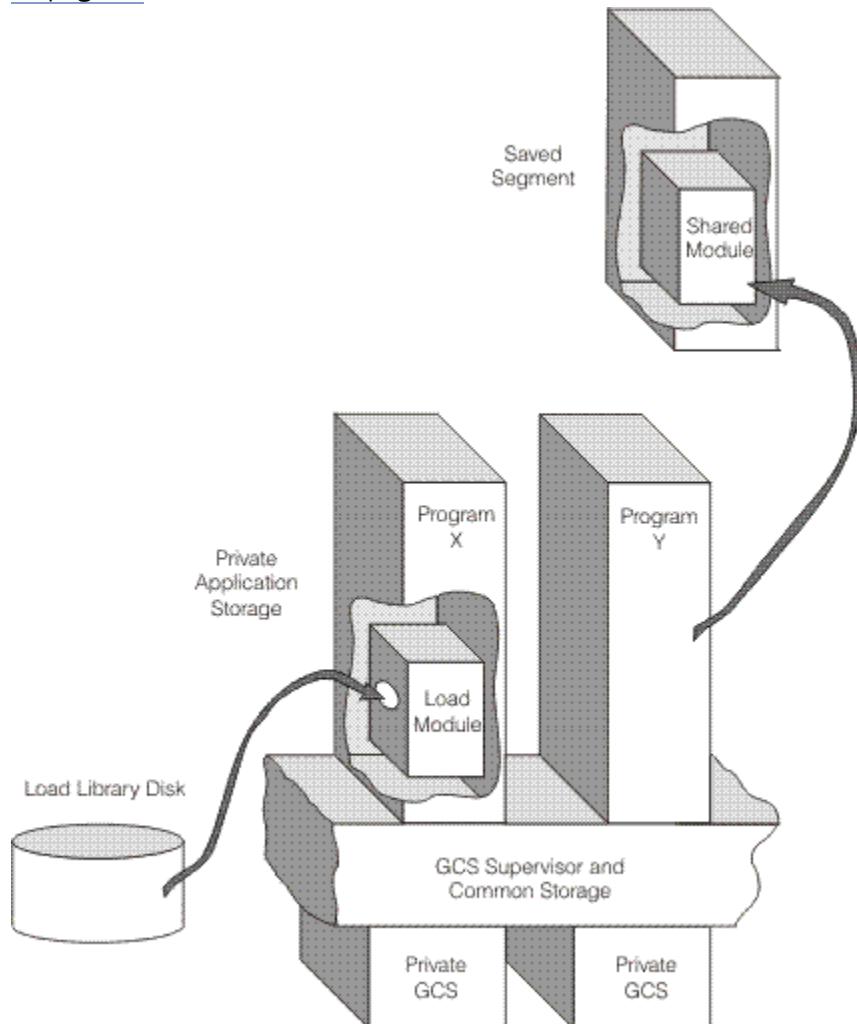
- Specifying a new PSW key with the SPKA instruction
- Allocating storage in the new key with the GETMAIN macro.

### ***Program Management***

Programs running on GCS can load and run modules of code that were assembled and link-edited under CMS. Some of these modules reside on a disk in a load library. Others reside in saved segments that get linked automatically when you IPL your GCS segment.

When a GCS program requests a module, the GCS supervisor first tries to find one that was previously loaded in that program's virtual machine. If no usable copy is available, the supervisor tries to locate the module in one of your system's saved segments. Each saved segment has a directory created with

the CONTENTS macro. The GCS supervisor searches these directories when looking for a particular module. (In either case, the supervisor will use a copy where it locates one.) Failing to find it in a saved segment, the supervisor searches the load libraries specified by GCS's GLOBAL LOADLIB command. If the supervisor finds the module there, it loads a copy into the program's private storage area. See [Figure 14 on page 45](#).



*Figure 14. Obtaining Modules Requested by a GCS Program*

To load a module, a program can issue any of the following macros in [Table 6 on page 45](#).

*Table 6. Loading Functions*

Macro	Action 1	Action 2	Action on Return
LINK	Finds and loads a module (if it was not already in storage) containing a specified entry point.	Passes control to the loaded module at the specified entry point.	After the LINKed module runs, control returns to the program that issued LINK. And if no other program is using that copy of the module, GCS deletes it from storage.
LOAD	Locates and loads a module (if it was not already in storage).	Returns the address of an entry point, plus the AMODE bit, associated with the loaded module, to the program that issued LOAD.	LOAD returns control to the program that issued it. The supervisor keeps track of the module's whereabouts until the program issues DELETE.

*Table 6. Loading Functions (continued)*

<b>Macro</b>	<b>Action 1</b>	<b>Action 2</b>	<b>Action on Return</b>
XCTL	Finds and loads a module (if it was not already in storage) containing a specified entry point.	Passes control to the loaded module at a specified entry point.	After the XCTLed module runs, control does not return to the program that issued XCTL, but to the program before that.

Macros associated with these loading functions include:

#### **BLDL**

Creates a directory entry list that contains information about modules you expect to invoke. (It includes their names, what load libraries they reside in, their disk addresses, and other facts).

#### **CALL**

Passes control to an entry point in the same or different control section (csect).

#### **DELETE**

Releases a module from its caller's control (and removes it from storage if no other programs want to use it).

#### **IDENTIFY**

Defines an entry point within a load module.

#### **RETURN**

Returns control to the calling program.

#### **SAVE**

Saves the contents of registers belonging to a program that is calling another program.

#### **SYNCH**

Passes control to a program, in the same or different state, at a specified entry point.

The macros are described later in this book.

Here are examples of how you might use the loading macros:

#### **LOAD**

1. Program 1 LOADs module A.
2. Program 1 gives control to module A with LINK or SYNCH.
3. Module A executes.
4. Program 1 regains control when module A finishes.
5. Program 1 DELETEs module A.

#### **LINK and XCTL**

1. Program 2 LINKs to module B.
2. Module B executes and XCTLs to module C.
3. Module C executes.
4. Program 2 regains control when module C finishes.

## **Timer Management**

Programs or tasks that run under GCS sometimes need the services of a timer. A task, for example, may want to set a timer for a certain interval and, when that interval is up, transfer control to an exit routine. Another task might want to set a timer for a certain interval and then stop executing until that interval expires.

GCS has three macros that let tasks define and manage time limits:

#### **STIMER**

Lets you set an interval by specifying:

**a time length**

For the next 10 seconds, do this ...

**a time-of-day**

At 09:30, do this ...

**TIME**

Asks the GCS supervisor to provide the current time-of-day and date. In effect, it asks the system, *What time is it right now?*

**TTIMER**

Cancels any remaining interval (and exit routine) that was set with the STIMER macro.

## Native GCS Services

*Authorization* provides the basis for native GCS services. Some functions serve unauthorized programs running in problem state machines; other functions serve only authorized programs running in supervisor state machines.

### Calling Authorized Programs

An unauthorized GCS program in problem state can transfer control to an authorized program in supervisor state. When called, the authorized program executes, beginning at an identified entry point in shared storage. Upon finishing, it returns control to the unauthorized program.

This transfer of control involves two macros:

**AUTHNAME**

The authorized program has to provide an authorized entry point, identified with the AUTHNAME macro.

**AUTHCALL**

The unauthorized program *calls* and passes control to the authorized one by issuing the AUTHCALL macro.

Table 7. The AUTHCALL Macro

AUTHCALL does	AUTHCALL does not
Cause an authorized program to start executing at an entry point identified with AUTHNAME. The entry point always receives control in supervisor state and Key 0.	Cause a task switch to occur. (The same task is still running.)
Return control to the calling program in its original state and key, when the authorized program finishes.	Let an unauthorized program run its own code in supervisor state or Key 0.

### Communicating through IUCV

GCS supports communication within a virtual machine, or between any two virtual machines, at a routine-to-routine level. Task-users (routines running within a task) communicate through IUCV with:

- Other task-users in the same machine
- Task-users in other virtual machines on the same system
- CP.

GCS also supports communication between virtual machines within a group of VM systems, or cluster. Each of these VM systems must have the Transparent Services Access Facility (TSAF) virtual machine component installed and running. For more information on installing TSAF, see [z/VM: Installation Guide](#). For more information on running TSAF, see [z/VM: Connectivity](#). Task-users (routines running within a task) communicate through APPC/VM with resource task-users in the:

- Same machine

- Other virtual machines in the same or different systems.

The target of an APPC/VM connection must be established as a resource. For more information, see [z/VM: CP Programming Services](#)

Task-users rely on two macros for IUCV and APPC/VM communications:

### **IUCVINI**

Initializes or terminates a task-user's IUCV environment

### **IUCVCOM**

Sets up, carries out, and terminates communications between two IUCV users.

To allow IUCV and APPC/VM communication at the task-user level, GCS provides:

1. A *nonprivileged* IUCV interface for both authorized and unauthorized task-users. This nonprivileged interface provides the following support:

#### **Functions provided:**

#### **Functions not provided:**

#### **ACCEPT**

DCLBFR (Declare Buffer)

#### **CONNECT**

RTRVBFR (Retrieve Buffer)

#### **PURGE (IUCV only)**

DESCRIBE (Describe)

#### **QUERY**

SETMASK (Set Mask)

#### **QUIESCE (IUCV only)**

SETCMASK (Set Control Mask)

#### **RECEIVE**

TESTCMPL (Test Completion)

#### **REJECT (IUCV only)**

TESTMSG (Test Message)

#### **REPLY (IUCV only)**

#### **RESUME (IUCV only)**

#### **SEND**

#### **SEVER**

**Note:** The SEND function issues all of the APPC/VM “SEND” functions:

- SENDCNF
- SENDCNFD
- SENDDATA
- SENDERR
- SENDREQ.

2. A *privileged* interface only for authorized task-users that specify PRIV=YES with the IUCVINI SET function. With the privileged interface, a task-user:

- Cannot issue IUCVINI REP to change its general exit
- Cannot issue IUCVCOM REP to change a path-specific exit
- Must use the IUCVCOM functions CONNECT, ACCEPT, and SEVER to establish or terminate IUCV and APPC/VM paths
- Can issue the following functions directly (without going through the IUCVCOM macro):

#### **IUCV PURGE**

IUCV REJECT

<b>IUCV QUERY</b>	IUCV REPLY
<b>IUCV QUIESCE</b>	IUCV RESUME
<b>IUCV RECEIVE</b>	IUCV SEND
<b>APPCVM QUERY</b>	APPCVM RECEIVE
<b>APPCVM SENDCNF</b>	APPCVM SENDCNFD
<b>APPCVM SENDDATA</b>	APPCVM SENDERR
<b>APPCVM SENDREQ</b>	

## Performing I/O

When a GCS program needs an I/O operation performed, it uses a function called General I/O. The related macro, GENIO, provides six different functions that an unauthorized application can use to process virtual channel programs on any real or virtual I/O device except DASD and the virtual machine console:

- Open Device (OPEN)

This function identifies a task as owner of a particular I/O device. OPEN also requires the task to specify an exit. Whenever the task receives an I/O interrupt from the device, this specified exit gets control.

- Close Device (CLOSE)

This function ends a task's *ownership* of a specified device. Once closed, the device stops passing I/O interrupts to the specified exit.

- Modify (MODIFY)

This function requests that an active channel program be modified. An application first must modify the virtual channel program and then issue MODIFY.

- Obtain Device Characteristics (CHAR)

This parameter returns information about an I/O device's type, class, model, and features.

- Start I/O (START)

This function starts a virtual channel program on an open device. (The device may be either virtual or real.)

- Halt I/O (HALT)

This halts an operation on a given device, terminating any active I/O.

The GENIO macro also provides a function for authorized programs that want to process real channel programs on real devices:

- Start real I/O (STARTR)

This starts a real channel program on an open real device. (The device must be real.)

## Executing Real Channel I/O Programs

Authorized GCS programs can use real channel programs to move data between main storage and real I/O devices (except DASDs). Real channel programs run directly on the real channel, without CP first translating them. Before you can run real channel programs, you need an authorized user ID and a special entry in your z/VM directory. You make this entry by specifying the DIAG98 parameter on the OPTION directory control statement.

To process real I/O, authorized programs use GENIO's STARTR (start real) function. STARTR works much like the ordinary START function for virtual I/O. However, with STARTR:

- CP does not translate the channel program before starting it.
- GCS issues a DIAGNOSE code X'98' instead of an SSCH instruction (or an SIOF instruction for 370 accommodation).

## Securing Pages of Storage

An authorized program intending to perform real I/O using STARTR must first build a channel program in real storage. In the process of building a real channel program, the program must lock pages of virtual storage into real storage. Later, it needs a way to unlock those pages.

The two macros that do this are:

### **PGLOCK**

Locks given pages of virtual storage into real storage

### **PGULOCK**

Unlocks pages that were fixed through the PGLOCK macro.

## Manipulating Locks

Locks are controls that help authorized programs share resources. They serve as warning signs that a particular resource is *in use*. There are two kinds of locks:

### **Local**

Helps synchronize the use of resources within a virtual machine

### **Common**

Helps synchronize common storage among many virtual machines.

The GCS supervisor uses the LOCKWD macro to manipulate these locks and regulate access to local resources or common storage. The LOCKWD macro has parameters that:

- Identify a lock as local or common
- Test the common lock (to see whether it is *on* or *off*)
- Specify whether the lock is to be acquired or released.

When a program or task wants to use a resource within its own virtual machine, it uses the LOCKWD macro to acquire the local lock for that machine. That action prevents all other tasks in the virtual machine from running until the lock is released.

When a task wants exclusive use of common storage, it can acquire the common lock for its virtual machine. First, a task has to acquire the local lock before it tries to acquire the common lock. Next, the program should use the LOCKWD macro to test the common lock's availability. If another virtual machine already has acquired it, the lock will be *on*. Until that machine releases the lock, no other machine will be able to acquire it. In the meantime, if a program tries to acquire the common lock when it is already *on*, the GCS supervisor will suspend the requesting program until the lock gets turned off. When it is *off*, LOCKWD informs the waiting machine that the common lock is available. This serializes (or synchronizes) group use of common storage.

## Validating Requests for Storage Access

An authorized program can validate another program's request for storage access. The authorized program uses the VALIDATE macro to check input (a parameter list, for example) from the other program. VALIDATE compares the other program's PSW key with the storage key of the storage area to be accessed. If those two keys match, the authorized program will honor the storage access request for both read and write access. If the keys are different and the storage is nonfetch protected, the authorized program will allow read access only. The authorized program's key does not need to match either the unauthorized program or storage. As an authorized program, it can switch itself to key 0 and transfer data across the different key boundaries.

## Scheduling Exits in Other Tasks

An authorized program can schedule an exit for any task in any group machine. With the SCHEDEX macro, the program can preempt a specific task and arrange for a designated exit routine to assume control. Instead of the task getting dispatched (if it is not disabled), the exit routine gets control in supervisor state and with a PSW key of 0.

After scheduling the exit, the authorized program continues executing. And after the exit routine finishes, GCS lets the interrupted task continue executing.

## Establishing Exits for Group Members

Authorized programs can establish exits for the entire virtual machine group. These exit routines must reside in storage that all machines in the group can share.

- Machine exits

Authorized programs can use the MACHEXIT macro to set up exit routines that will get control when any machine terminates or leaves the group. These routines will process in the group's recovery machine.

- Task exit routines

Authorized programs define task exit routines for programs in the same virtual machine group. Whenever a task in one of the group's virtual machines terminates, a specified exit routine gains control. An authorized program uses the TASKEXIT macro to identify the address where that exit routine begins.

- Exits to authorized entry points

Defining an entry point does not define an *exit*, in the true sense of the word. However, when an authorized program identifies an entry point with the AUTHNAME macro (see “[Calling Authorized Programs](#)” on page 47), it resembles the act of identifying an exit routine's address. For more information on transferring control to authorized entry points, see “[AUTHNAME](#)” on page 174 and “[AUTHUSER](#)” on page 179.

## Data Management Services

GCS applications can process CMS files that reside on minidisks, VSAM files, and CP spool files. GCS applications cannot process CMS files that reside in a Shared File System (SFS) file pool. With GCS's data management services, applications can perform input, output, or update operations on a file, depending on whether it is a CMS, VSAM, or CP spool file. Two types of data management services:

1. One type (resembling, but not duplicating, MVS/BSAM and MVS/QSAM services) that allows processing of CMS disk files and CP spool files
2. Another type (resembling, but not duplicating, MVS/VSAM services) that allows processing of VSAM files.

## Processing CMS Minidisk Files

A GCS program processes CMS minidisk files using BSAM or QSAM macros. For GCS, these macros have unique constraints. In particular, GCS's data management service supports only the *extended file system* format.

GCS's QSAM/BSAM data management service supports the following command:

### **FILEDEF**

Defines CMS minidisk files and CP spool files.

GCS data management supports the following set of macros, at the MVS/SP 1.3.1 level:

### **CHECK**

Wait for and test completion of a read or write operation (BSAM).

### **CLOSE**

Logically disconnect a file (BSAM and QSAM).

**DCB**

Construct a data control block (BSAM and QSAM).

**DCBD**

Provide symbolic reference to data control blocks (BSAM and QSAM).

**GET**

Obtain next logical record (QSAM).

**NOTE**

Determine relative position (BSAM).

**OPEN**

Logically connect a file (BSAM and QSAM).

**POINT**

Point to the relative position of a specific block (BSAM).

**PUT**

Write next logical record (QSAM).

**READ**

Read a block (BSAM).

**SYNADAF**

Perform SYNAD analysis function (BSAM and QSAM).

**SYNADRLS**

Release SYNADAF buffer and save areas (BSAM and QSAM).

**WRITE**

Write a block (BSAM).

Unlike CMS's data management service, it does not allow:

- Files that reside in a CMS Shared File System file pool
- OS formatted files
- File mode 4 (treated instead like file mode 1)
- Spanned records
- Console or tape I/O
- Utility functions (like formatting disks, loading files from tape, editing files, and others).

GCS's data management does follow the same rules as CMS's when two or more virtual machines want to share the same disk. Read/write privileges go to only one virtual machine at a time, while multiple disk and minidisk users must share in read-only mode. For more information on disk sharing, see [z/VM: CMS User's Guide](#).

Sometimes two or more tasks within the same machine need to share a single file. They can do this under two conditions:

1. If they concurrently open and use multiple Data Control Blocks (DCBs) that refer to the same, shared file.

When many DCBs refer to a single file, the type of processing (input, output, or update) decides what programming procedures you should use. [Table 8 on page 52](#) shows you the different types of processing and the requirements that go along with each.

*Table 8. Opening Multiple DCBs*

Type of processing:	Programming required:
INPUT	Each task should issue READ and GET requests as if no file sharing were taking place. GCS keeps track of the read pointers.
OUTPUT	This sort of sharing is not supported for multiple DCBs. Unpredictable results will occur if you attempt it.

Table 8. Opening Multiple DCBs (continued)

Type of processing:	Programming required:
UPDATING (in BSAM)	Each task should issue ENQ before the READ macro. This helps serialize the processing of each block of records. Macros issued to complete the update are WRITE, CHECK, and DEQ, in that order. For more information on these macros, see “ <a href="#">WRITE (BSAM)</a> ” on page 413, “ <a href="#">CHECK (BSAM)</a> ” on page 380, and “ <a href="#">DEQ</a> ” on page 204.
UPDATING (in QSAM)	When updating a file, a task must avoid altering blocks containing records that other tasks are updating. GCS has no way of knowing whether different tasks are processing discrete blocks.

**Note:** When you share a file with multiple DCBs, be sure you enter the FILEDEF command only once for each ddname. If you need to enter FILEDEF for the same ddname and same file later in the program, make sure you specify the NOCHANGE option. See “[FILEDEF](#)” on page 94.

2. If they concurrently open and use only one shared DCB.

When tasks share a single DCB, GCS permits all three types of processing (inputting, outputting, and updating). However, tasks have to use the ENQ and DEQ macros to coordinate their activities. (See “[Coordinating Shared Resources](#)” on page 22.) Because only one of them can have control at a time, the tasks must issue the ENQ macro first (to take turns at getting control) and end with the DEQ macro (to release control).

## Data Compression

You can save data in a compressed format to conserve storage media and network transmission line costs. The CSRCMPSC macro provides services that compress and expand data. These services are available when the CVTCMPSC bit is on in the communication vector table (CVT). Data Compression Services will use the S/390® hardware compression feature, if available. Otherwise, a software compression program will simulate the hardware instruction. If the CVTCMPSH flag is on in the CVT, the hardware feature will be used for the compression.

Compression takes an input string of data and, using a data area called a *dictionary*, produces an output string of compression symbols. Each symbol represents a string of one or more characters from the input.

Expansion takes an input string of compression symbols and, using a *dictionary*, produces an output string of the characters represented by those compression symbols. Dictionary items are mapped by various forms of the CSRYCMPD macro.

Parameters for the CSRCMPSC macro are in an area mapped by CMPSC DSECT (CSRYCMPS macro) and specified by the CBLOCK parameter of the CSRCMPSC macro. These parameters contain the following information:

- The address, ALET, and length of a source area. The source area contains the data to be compressed for a compression operation, or to be expanded for an expansion operation.
- The address, ALET, and length of a target area. After the macro runs, the target area contains the compressed data for a compression operation, or the expanded data for an expansion operation.
- An indication of whether to perform compression or expansion.
- The address and format of a dictionary to be used to perform the compression or expansion. The dictionary must be in the same address space as the source area.

For more information on how to use Data Compression Services with GCS, see [Appendix D, “Data Compression Services,”](#) on page 539.

### Processing CP Spool Files

BSAM and QSAM functions let GCS programs process virtual reader, printer, and punch files. Existing CP facilities, like CP Directory, DEFINE, DETACH, SPOOL, TAG, and so on, define and manipulate the various unit record devices.

**Note:** GCS programs cannot write to virtual readers, nor can they read from virtual printers and punches.

### Processing VSAM Files

GCS programs use VSAM macros supported at the MVS/VSAM Release 3.8 level, the same level as CMS. In fact, you will find them in a CMS macro library named OSVSAM MACLIB. When you request a service with one of these macros, it gets mapped to VSE/VSAM format and executed using VSE/VSAM code.

GCS's VSAM data management service supports the following command:

#### **DLBL**

Identifies VSAM files for I/O

GCS data management supports the following macros:

#### **ACB**

Generates an access method control block at assembly time

#### **BLDVRP**

Builds a resource pool for Local Shared Resources

#### **CHECK**

Suspends processing and waits for a request to complete

#### **CLOSE**

Disconnects a program and data

#### **DLVRP**

Deletes a resource pool

#### **ENDREQ**

Terminates a request

#### **ERASE**

Deletes a record

#### **EXLST**

Generates an exit list

#### **GENCB**

Generates an access method control block, exit list, or request parameter list at execution time

#### **GET**

Retrieves a record

#### **MODCB**

Modifies an access method control block, exit list, or request parameter list dynamically

#### **OPEN**

Connects a program and data

#### **POINT**

Points VSAM to a specific record to be accessed

#### **PUT**

Stores a record

#### **RPL**

Generates a request parameter list

#### **SHOWCAT**

Retrieves information from the VSAM catalog

#### **SHOWCB**

Displays fields of a control block or list

**TESTCB**

Tests values in a control block or list

**WRTBFR**

Writes buffers that contain Deferred Writes

**Note:**

1. The control blocks generated by the OS ACB, RPL, and EXLST macros are converted from OS format to VSE format the first time that these control blocks are used by GCS. Because of this, the TESTCB, SHOWCB, and MODCB macros, rather than the OS mapping macros from the OSVSAM macro library, should get or modify data in these control blocks.
2. VSAM data management services support the CHECK macro and RPL's ASY option, but no asynchronous activity is performed.
3. GCS does not support utility functions. You have to perform disk initialization, catalog definition, and file definition (AMS functions) under CMS.
4. VSE/VSAM governs the sharing of VSAM data within a GCS virtual machine. The way you define a VSAM file and the way you use it determines how VSE/VSAM handles shared data. For more information, see the *VSE/VSAM User's Guide*.
5. When a task terminates, GCS attempts to close all open ACBs that the task opened.

Planning for GCS involves:

- Being familiar with the current procedures that tell how to plan for shared segments
- Knowing the requirements of all products you plan to run on GCS
- Making entries in your z/VM directory
- Reserving enough pages in storage to hold your GCS shared segment
- Defining your GCS configuration file with the GROUP EXEC.



# Chapter 4. GCS Commands

The GCS commands are:

Command	Function	page
ACCESS	Identifies CMS or VSAM disks that an application uses.	<a href="#">“ACCESS” on page 59</a>
CLEAR	Clears the virtual console.	<a href="#">“CLEAR” on page 61</a>
CONFIG	Lets the owner of the recovery machine change some of the configuration information supplied during system build.	<a href="#">“CONFIG” on page 62</a>
DLBL	Defines VSAM files used for program I/O.	<a href="#">“DLBL” on page 64</a>
ERASE	Removes one or more files from a read/write disk.	<a href="#">“ERASE” on page 70</a>
ESTATE/ESTATEW	Verifies the existence of a file on an accessed disk. ESTATEW verifies the existence of a file residing on a read/write minidisk.	<a href="#">“ESTATE/ESTATEW” on page 71</a>
ETRACE	Starts or stops external trace processing.	<a href="#">“ETRACE” on page 73</a>
EXECIO	Does I/O operations on a program stack or a variable, file, or executes CP commands.	<a href="#">“EXECIO” on page 76</a>
FILEDEF	Defines CMS format files and spool files.	<a href="#">“FILEDEF” on page 94</a>
GDUMP	Produces a copy of the contents of your virtual machine's storage.	<a href="#">“GDUMP” on page 98</a>
GLOBAL	Defines the CMS load libraries you want searched for modules.	<a href="#">“GLOBAL” on page 101</a>
HX	Stops execution of all programs and commands active in a virtual machine.	<a href="#">“HX” on page 107</a>
ITRACE	Enables or disables recording of internal trace events within a virtual machine or virtual machine group.	<a href="#">“ITRACE” on page 108</a>
LOADCMD	Defines a program that runs as a command.	<a href="#">“LOADCMD” on page 112</a>
OSRUN	Starts a GCS application program.	<a href="#">“OSRUN” on page 116</a>
QUERY	Request information about your GCS virtual machine.	<a href="#">“QUERY” on page 117</a>
RELEASE	Release a disk.	<a href="#">“RELEASE” on page 146</a>
REPLY	Replies to a message sent to the GCS operator.	<a href="#">“REPLY” on page 147</a>
SET	Replaces a saved system name entry in the SYSNAMES table for VSAM or to set or reset a particular function in your GCS machine.	<a href="#">“SET” on page 149</a>

## Immediate Commands

---

An immediate command is one that gets executed when you enter it. It does not get stacked, nor does it have to wait for the current command to finish. The immediate GCS commands are:

ETRACE (see page “[ETRACE](#)” on page 73)  
GDUMP (see page “[GDUMP](#)” on page 98)  
HX (see page “[HX](#)” on page 107)  
ITRACE (see page “[ITRACE](#)” on page 108)  
QUERY (see page “[QUERY](#)” on page 117)  
REPLY (see page “[REPLY](#)” on page 147).

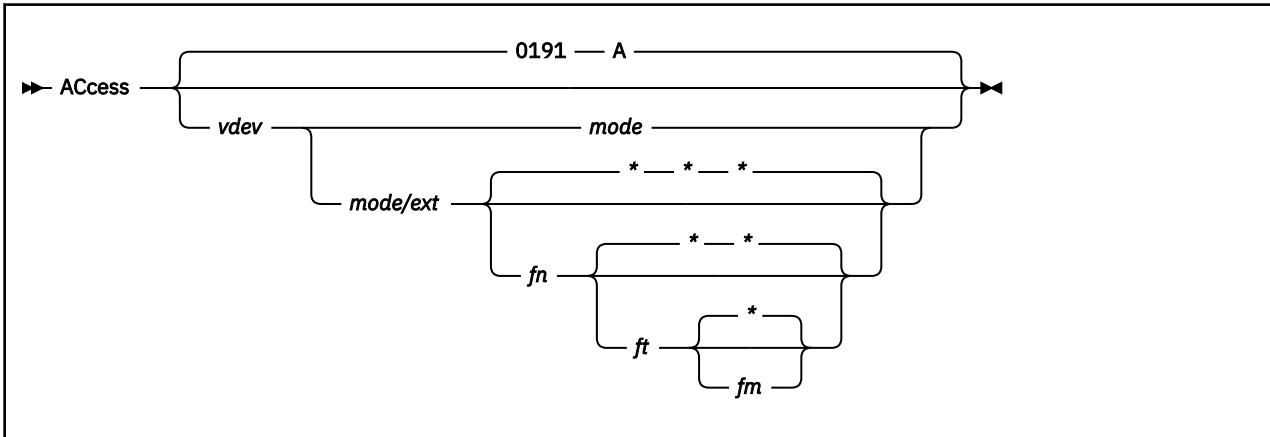
**Note:** If you enter several commands on the command line and separate them with # characters:

```
cmd1#cmd2#immed cmd#cmd3
```

your system will process any immediate commands first. You would receive results from *immed cmd* before the results from *cmd1*. If an exec or routine is named the same name as an immediate command, the immediate command is executed. This differs from the way CMS processes commands.

# ACCESS

## Format



## Purpose

Use the ACCESS command to identify the CMS or VSAM Disks that an application will use.

Applications that use files on CMS or VSAM disks must first identify those disks with the ACCESS command. The disk you identify must be either a:

- VSAM disk. (Make sure you enter ACCESS before entering the DLBL command.)
- CMS disk formatted with a block size of 512, 1KB, 2KB, or 4KB bytes. (You cannot have an 800-byte block size.)

## Operands

### **vdev**

Makes the disk available at the specified virtual address. Valid addresses are X'0001' through X'FFFF' (X'0001' through X'1FFF' for 370 accommodation). The default value is 0191.

### **mode**

Assigns a one-character file mode letter to all files on the disk being accessed. You must specify this field if you specified the **vdev** parameter. The default value is A.

### **ext**

Indicates the mode of the parent disk. Files on the disk being accessed (**vdev**) are logically associated with files on the parent disk; the disk at **vdev** is a read-only extension. A parent disk must be accessed in the search order before its extension gets accessed. Do not put a blank space before or after the slash (/).

### **fn ft fm**

Defines a subset of files residing on the disk to be accessed. These are the only files that will go into your user file directory, and these are the only files you will be able to read. Entering an asterisk (\*) in any one of these fields indicates that you want all file names or file types or file mode numbers (except 0) to be in your user file directory. You can specify file name, file type, and file mode fields only for CMS-formatted disks that you have accessed as read-only extensions. For example, to specify a file mode, use a letter and a number:

```
access 333 b/a * gcs b1
```

**Note:** You should enter the RELEASE command when your application no longer needs access to the disk.

## Messages

- **GCTACC005S** Virtual storage capacity exceeded RC=104
- **GCTACC006E** Invalid parameter '*parameter*' RC=24
- **GCTACC012E** No options allowed RC=24
- **GCTACC021E** Invalid mode '*mode*' RC=24
- **GCTACC414E** Disk *vdev* not properly formatted for ACCESS RC=16
- **GCTACC415E** Invalid device address '*vdev*' RC=24
- **GCTACC422E** *vdev* already accessed as Read/Write '*mode*' disk RC=36
- **GCTACC423I** *mode (vdev) { R/O | R/W}*
- **GCTACC424I** *vdev mode* released
- **GCTACC425I** *vdev replaces mode (vdev)*
- **GCTACC426I** *vdev also = mode disk*
- **GCTACC427S** "*mode (vdev)*" device error RC=100
- **GCTACC428S** "*mode (vdev)*" not attached RC=100
- **GCTACC429E** File not found. Disk *mode (vdev)* will not be accessed RC=28
- **GCTACC430W** OS disk - Fileid specified is ignored RC=4
- **GCTROS005S** Virtual storage capacity exceeded
- **GCTROS423I** *mode (vdev) { R/O | R/W} { -OS | -DOS}*
- **GCTROS426I** *vdev also = mode { -OS | -DOS} disk*

For more information on messages, see [z/VM: Other Components Messages and Codes](#).

# CLEAR

---

## Format

```
▶▶ CLEAR ▶▶
```

## Purpose

Use the CLEAR command to clear the virtual console.

## Operands

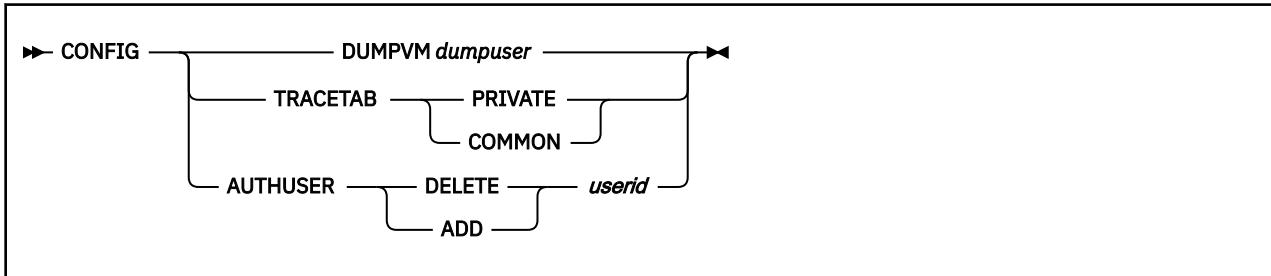
### CLEAR

Specifies that you want to clear the virtual console. Invariably, after the command completes control is returned to the user. The CLEAR command is processed immediately. It does not clear any lines that are queued and waiting to be displayed.

# CONFIG

---

## Format



## Purpose

The CONFIG command allows the owner of the recovery machine to change some of the configuration information provided during system build.

## Operands

### DUMPVM *dumpuser*

Alters the name of the virtual machine to receive dumps to the name specified in *dumpuser*.

### TRACETAB PRIVATE

Designates that the internal trace table is to be in private storage.

### TRACETAB COMMON

Designates that the internal trace table is to be in common storage.

### AUTHUSER DELETE *userid*

Deletes the user identified by *userid* from the authorized list.

### AUTHUSER ADD *userid*

Adds the user identified by *userid* to the authorized list.

## Usage

1. When the DUMPVM or TRACETAB options are specified, the configuration change is immediate.
2. When the AUTHUSER option is specified in a multiple user group environment, the change of the user authorization is not effective until the user re-IPLs.
3. When the AUTHUSER option is specified in a single user group environment, the user authorization change is immediate. Other user IDs may be specified with the AUTHUSER option, but they will have no effect on the operation of the single user environment.
4. The DUMPVM option is not valid in a single user environment.

## Messages

All CONFIG messages are issued without message numbers.

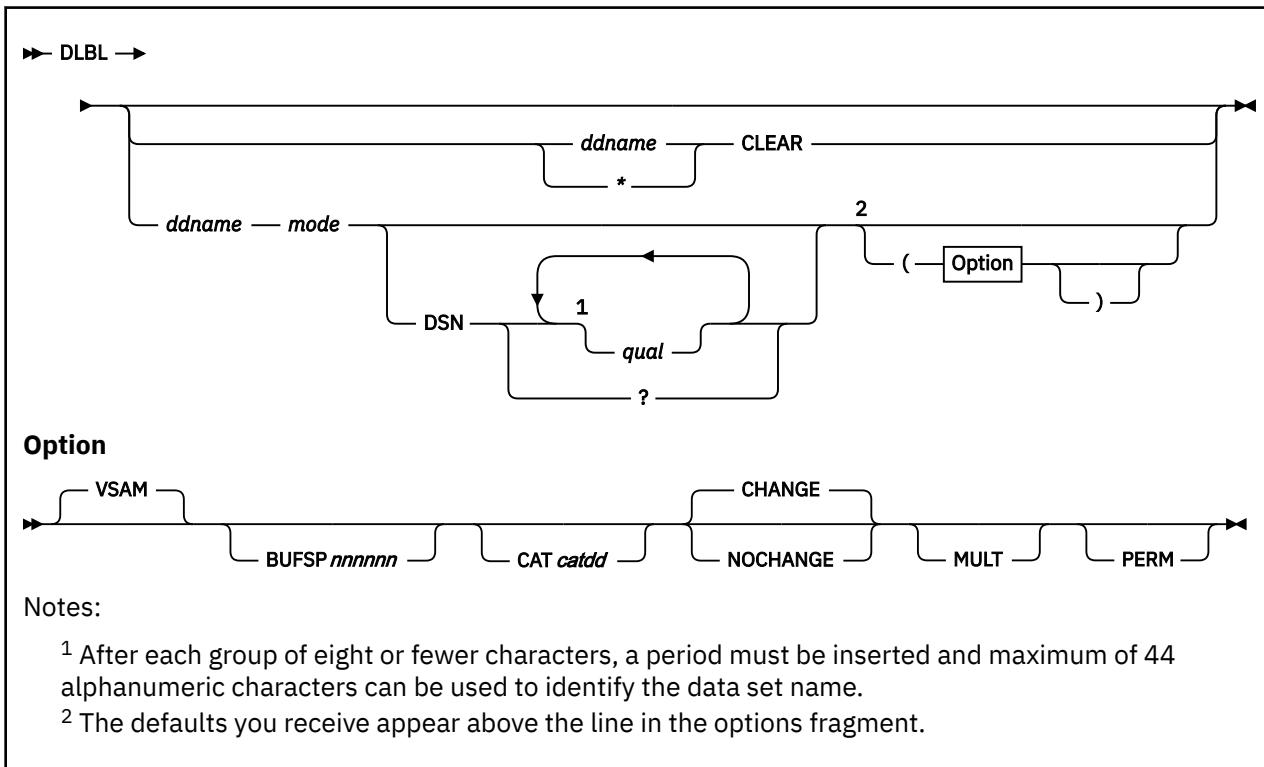
- **GCTCFG009E** Operand is missing or invalid
- **GCTGFI244I** '*userid*' is now the virtual machine receiving dumps
- **GCTGFI245I** '*userid*' can now IPL as an authorized virtual machine
- **GCTGFI246I** '*userid*' can no longer IPL as an authorized virtual machine
- **GCTGFI247I** The trace table is now being maintained in '*location*' storage
- **GCTGFI248I** No users are currently authorized

- **GCTGFI249E** The recovery machine '*userid*' must be authorized

For more information on messages, see [z/VM: Other Components Messages and Codes](#).

## DLBL

### Format



### Purpose

Use the DLBL command to define VSAM files used for program I/O.

Application programs usually require some *setting up* before you try to start and run them. The DLBL command is one of the preliminary commands usually issued to prepare a program for execution. You enter the DLBL command to define VSAM I/O files needed by the program. Be sure you enter the ACCESS command for the disk containing your VSAM files before entering DLBL.

**Note:** For non-VSAM file definitions, use the “FILEDEF” on page 94. VSAM itself does not always require file definition statements. For more information on file definitions, see the *VSE/VSAM User’s Guide*.

### Operands

#### *ddname*

A one- to seven-character program *ddname*. If you have *ddnames* over seven characters long, be aware that only the first seven characters get processed. If you have two different files with the same first seven letters and try to process them both, you will receive an error message when GCS opens the second file.

This *ddname* must be the same as the ACB DDNAME parameter (or the ACB name if DDNAME is omitted). An asterisk (\*) entered, along with the CLEAR operand, indicates that all DLBL definitions, except those that are entered with the PERM option, are to be cleared.

#### \* (asterisk)

Indicates that you want all ddnames.

**CLEAR**

Removes any conditions for the specified *ddname*. Clearing a *ddname* before defining it ensures that a file definition does not exist and that any options previously defined for that *ddname* no longer have any effect.

If you release a disk that has a DLBL definition in effect, you should clear that DLBL before executing a VSAM program. If a disk has a DLBL in effect, but the disk is not accessed, GCS will issue the message:

```
Disk { mode/vdev/volumeid} not accessed
```

**mode**

A letter representing the file mode of a VSAM disk and, optionally, a file mode number. You must specify a letter, and it must refer to a disk that is already accessed. The file mode number, however, is optional. If you do not provide one, the default is 1. VSAM disks do not require this number anyway, but GCS will accept one without error.

If a mode is specified, the associated disk must already be accessed.

**DSN**

Specifies that this is a VSAM file.

**qual**

A unique name associated with the file on the volume. It can be from one to 44 characters of alphanumeric data. If fewer than 44 characters are used, the field is left-justified and padded with blanks.

For VSAM, DSN must be specified when an existing (input) file is being processed. The name (qual) is identical with the name of the file, specified in the DEFINE command and listed in the VSAM catalog. For VSAM, the name (qual) must be coded according to the following rules:

- One to 44 alphanumeric (A-Z, 0-9, @, \$, or #) characters or hyphen (-) or plus zero (+0).
- After each group of eight or fewer characters, a period (.) must be inserted.
- Embedded blanks are not allowed.
- The first character of the name (qual) and the first character following a period must be alphabetic or national (A-Z, @, \$, #).

If this operand is omitted, *ddname* is used.

**? (question mark)**

Indicates that you will enter the *ddname* interactively. GCS will prompt you with the message:

```
Enter data set name:
```

When prompted, you must enter the data set name in its exact form, including embedded blanks, hyphens, or periods. If you enter it as a command at the console or from a REXX command file, you may use its exact form. DLBL will replace any blanks between qualifiers with periods.

**VSAM**

Indicates that the file is a VSAM data set. If not specified, VSAM is assumed.

**BUFSP *nnnnnn***

Specifies the number of bytes (in decimal) to be used for I/O buffers by VSAM data management during program execution, overriding the BUFSP value in the ACB for the file. The maximum value for *nnnnnn* is 999999; embedded commas are not permitted.

For more information, see the *Usage Notes* under the DLBL command in the [z/VM: CMS Commands and Utilities Reference](#).

**CAT *catdd***

Identifies the VSAM catalog (defined by a previous DLBL command) containing the entry for this data set. You must use the CAT option when the VSAM data set you are creating or identifying is not cataloged in the current job catalog.

*catdd* is the *ddname* in the DLBL definition for the catalog.

To identify a VSAM master catalog and job catalog, you have to use two special *ddnames*:

**IJSYSCT**

identifies the master catalog when you begin a terminal session. You should use the PERM option when you define it.

**IJSYSUC**

identifies a job catalog to be used for subsequent VSAM programs.

**Note:** VSAM programs search only one catalog when performing a function. If you defined an IJSYSUC job catalog, but want VSAM to use a different catalog, you have to indicate that other catalog with the CAT option. (See “[Examples](#)” on page 68.) [Figure 15 on page 67](#) shows how VSAM programs running on GCS go about selecting a VSAM catalog.

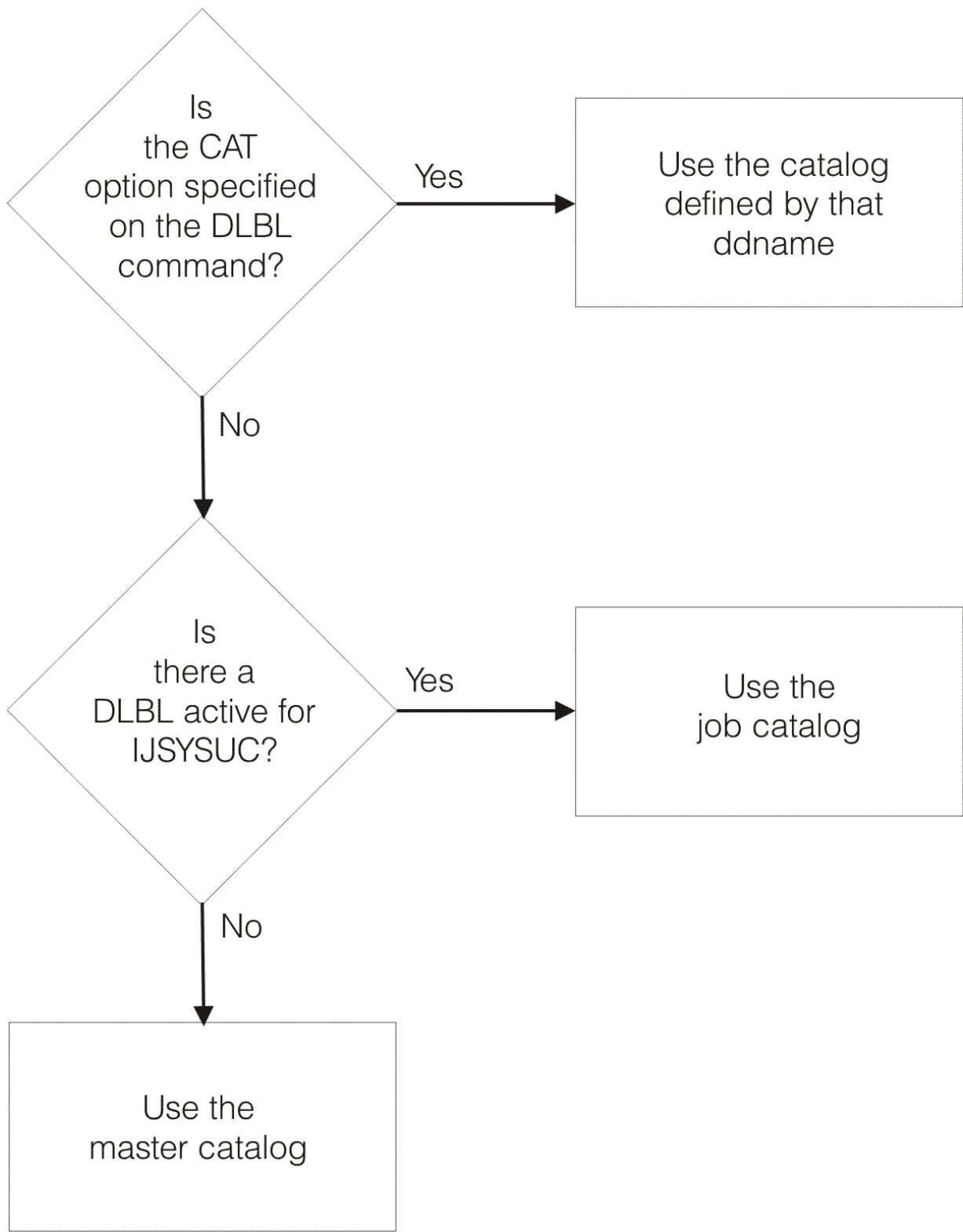


Figure 15. Determining Which VSAM Catalog to Use

#### CHANGE

Specifies that any existing definition for this *ddname* is not to be canceled, but conflicting options are to be overridden and new options merged into the existing definition. Both the *ddname* and the DSN file identifier must be the same for the definitions to be merged.

**NOCHANGE**

Indicates that a new definition for the specified *ddname* is to be created if none exists, but if a definition already exists, it is not to be changed.

**PERM**

Specifies that this DLBL definition can be cleared only by an explicit CLEAR request. It cannot be cleared when `dlbl * clear` is entered.

**MULT**

Indicates that you want to enter volume specifications that refer to an existing multivolume VSAM data set. For more information on the requirements for VSE/VSAM, see the *VSE/VSAM User's Guide*.

When you specify MULT, the GCS supervisor sends a message asking you for additional disk mode letters. You provide the mode letters using the REPLY command (["REPLY" on page 147](#)) and these rules apply:

- All the disks you refer to must be mounted and accessed when you enter the DLBL command.
- Do not repeat the mode letter that you entered on the command line.
- If you enter all the letters on the same line, separate them with commas. (GCS ignores any trailing commas at the end of the line.)
- You can specify a maximum of 25 disks, using any letter except S. However, you do not need to specify them in alphabetic order.

**Examples**1. =====> `dlbl`

Displays all file definitions in effect for your disks. GCS responds with:

```
'ddname' DISK 'fn' 'ft'  
      .     .     .  
      :     :     :  
      :     .     .
```

If you have no DLBL definitions in effect, GCS sends the following message:

```
No user defined DLBL'S in effect
```

2. =====> `dlbl infile c (mult`

Identifies a file named *infile* on your C mode disk and, because you specified the MULT option, prompts you to enter additional disk mode letters. You receive the following message:

```
nn GCTDLB312R Enter volume specifications:
```

where nn is a reply ID number. You enter the requested disk mode letters using the REPLY command (["REPLY" on page 147](#)) and this reply ID. For example, you may want to refer to disks accessed at modes D, E, F, and G.

Enter:

```
reply nn D, E, F, G  
reply nn
```

The second `reply nn` is a null line to terminate the command. If you do not enter this null line, you may get an error message and have to reenter the entire sequence of commands.

## 3. The following sequence of DLBL commands shows how you can use catalogs.

```
=====> dlbl ijsysct c dsn mastcat (perm
```

Identifies a VSAM master catalog, named MASTCAT, for the terminal session.

```
=====> dlbl ijsysuc d dsn mycat (perm
```

Identifies a VSAM job catalog, named MYCAT, for the terminal session.

```
=====> dlbl intest1 e dsn test.case
```

Identifies a VSAM file *intest1* that is cataloged in the job catalog MYCAT as *test.case*.

```
=====> dlbl cat3 dsn testcat (cat ijsysct
```

Identifies an additional catalog *testcat* which has an entry in the master catalog. Because you specified a job catalog (MYCAT) earlier, you must use the CAT option to make sure that the master catalog IJSYSCT gets used instead.

```
=====> dlbl infile e dsn test.input (cat cat3
```

Identifies an input file *infile* cataloged in your catalog TESTCAT, which was identified with a *ddname* of CAT3 on the previous DLBL command.

## Messages

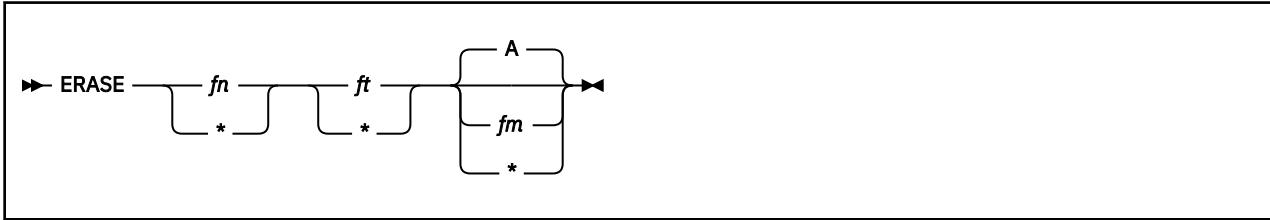
- **GCTDLB001E** Invalid option '*option*' RC=24
- **GCTDLB002E** Invalid parameter '*parameter*' in the option '*option*' field RC=24
- **GCTDLB003E** '*option*' option specified twice RC=24
- **GCTDLB004E** '*option1*' and '*option2*' are conflicting options RC=24
- **GCTDLB005S** Virtual storage capacity exceeded RC=104
- **GCTDLB006E** Invalid parameter '*parameter*' RC=24
- **GCTDLB009E** Operand is missing or invalid
- **GCTDLB017E** Disk {*mode/vdev/volumeid*} not accessed RC=36
- **GCTDLB021E** Invalid mode '*mode*' RC=24
- **GCTDLB302E** Parameter missing after DDNAME RC=24
- **GCTDLB303I** No user defined DLBL's in effect
- **GCTDLB305I** DDNAME '*ddname*' not found. CLEAR not executed
- **GCTDLB310R** Enter data set name:
- **GCTDLB311E** Invalid data set name RC=24
- **GCTDLB312R** Enter volume specifications:
- **GCTDLB313E** Invalid DDNAME '*ddname*' RC=24
- **GCTDLB314I** Maximum number of disk entries recorded
- **GCTDLB315E** Catalog DDNAME '*ddname*' not found
- **GCTDLB316E** *mode* disk is in CMS format; invalid for VSAM data set
- **GCTDLB317E** Job catalog DLBL cleared
- **GCTDLB318I** Master catalog DLBL cleared
- **GCTDLB345I** No *option* specified RC=24

For more information on messages, see [z/VM: Other Components Messages and Codes](#).

# ERASE

---

## Format



## Purpose

Use the ERASE command to delete one or more files from a read/write disk.

## Operands

### **fn**

is the name of the files to be erased. An asterisk (\*) coded in this position indicates that all names are to be used.

### **ft**

is the file type of the files to be erased. An asterisk (\*) coded in this position indicates that all file types are to be used.

### **fm**

is the file mode of the files to be erased. If this field is omitted, only the disk accessed as A is searched. An asterisk (\*) coded in this position indicates that files with the specified name and file type are to be erased from all read/write disks.

## Usage

1. If you specify an asterisk for both file name and file type, you must specify a file mode letter and number; for example, `erase * * a5`
2. If you enter an asterisk for the file mode, either the file name or the file type or both must be specified.
3. GCS supports passing of a parameter list that has an address above the 16MB line.
4. Control is passed back to the caller in the amode/rmode of the caller.
5. Authorized or unauthorized calls are supported.

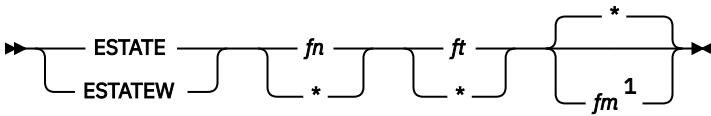
## Messages

- **GCTERS005S** Virtual storage capacity exceeded RC=25
- **GCTERS006E** Invalid parameter '*parameter*' RC=24
- **GCTERS019E** No read/write '*mode*' disk accessed RC=36
- **GCTERS021E** Invalid mode '*mode*' RC=24
- **GCTERS053E** File '*fn ft fm*' not found RC=28
- **GCTERS054E** Incomplete fileid specified RC=24
- **GCTERS062E** Invalid character '*char*' in fileid '*fn ft*' RC=20

For more information on messages, see [z/VM: Other Components Messages and Codes](#).

# ESTATE/ESTATEW

## Format



### Notes:

<sup>1</sup> File mode number specified is ignored when both the file name and file type are specified; it is used when both file name and type are specified as asterisks.

## Purpose

Use the ESTATE command to verify the existence of a file on any accessed disk. Use the ESTATEW command to verify the existence of a file on any accessed read/write disk.

## Operands

### **fn**

is the file name of the file whose existence is to be verified. If an asterisk is specified, the first file found satisfying the rest of the file ID is used.

### **ft**

is the file type of the file whose existence is to be verified. If an asterisk is specified, the first file found satisfying the rest of the file ID is used.

### **fm**

is the file mode of the file whose existence is to be verified. If an asterisk is specified, the first file found satisfying the rest of the file ID is used. If file mode is omitted or specified as \*, all your accessed disks (A-Z) are searched.

## Usage

1. If you enter the ESTATEW command specifying a file that exists on a read-only disk, you will receive an error message indicating that the file was not found.
2. ESTATE and ESTATEW can be used to verify only on CMS formatted disk.
3. Files larger than 65533 records are supported.
4. You can start the ESTATE or ESTATEW command from the terminal, from an exec file, or as a function from a program using CMDSI.
5. GCS supports passing of a parameter list that has an address above the 16MB line.
6. Control is passed back to the caller in the amode/rmode of the caller.
7. Authorized or unauthorized calls are supported.

## Messages

- **GCTSTT006E** Invalid parameter '*parameter*' RC=24
- **GCTSTT017E** Disk '*mode*' not accessed RC=36
- **GCTSTT021E** Invalid mode '*mode*' RC=24
- **GCTSTT053E** File '*fn ft fm*' not found RC=28
- **GCTSTT054E** Incomplete fileid specified RC=24

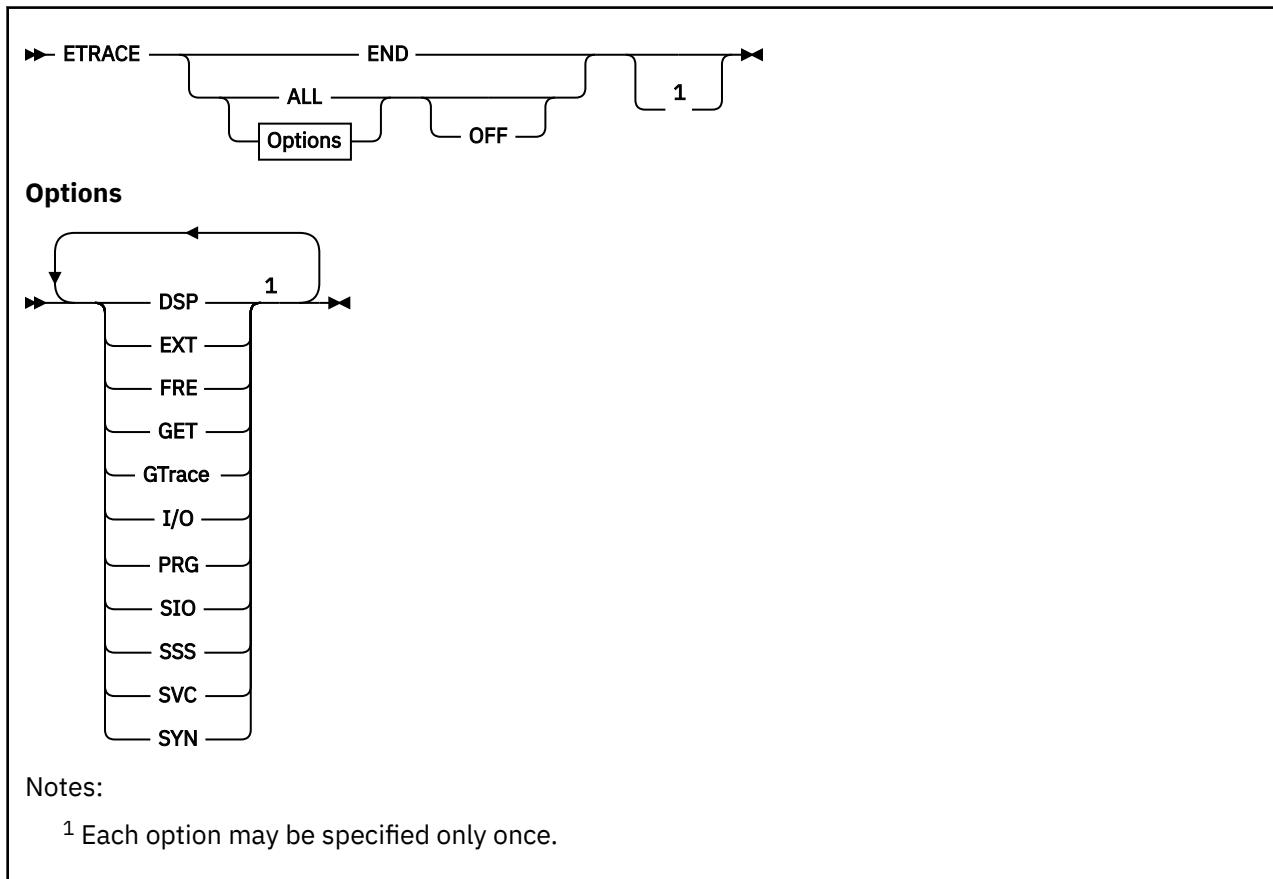
- **GCTSTT062E** Invalid character 'char' in fileid 'fn ft' RC=20

For more information on messages, see [z/VM: Other Components Messages and Codes](#).

# ETRACE

---

## Format



## Purpose

Use the ETRACE command to enable or disable the recording of events in a spool file for a virtual machine or virtual machine group.

GCS supports external tracing — the recording of events in a spool file. For more information on External Tracing Facility, see the *z/VM: Diagnosis Guide*. You control when this external tracing is active by using the ETRACE command.

You can enable (activate) or disable (deactivate) external tracing for a particular virtual machine or an entire virtual machine group. You can specify a certain list of events for one virtual machine and a totally different set of events for all other virtual machines in the group.

Before any external tracing actually takes place, though, a class C user must enter the TRSOURCE command for the virtual machines to be traced.

## Parameters

### END

Disables external tracing of all events.

### ALL

Enables or disables external tracing of all events.

**DSP**

Enables or disables external tracing of each task switch (dispatch of a different task).

**EXT**

Enables or disables external tracing of each external interrupt.

**FRE**

Enables or disables external tracing of FREEMAIN events started through SVC and Branch Entry calls.

**GET**

Enables or disables external tracing of GETMAIN events started through SVC and Branch Entry calls.

**GTrace**

Specifies that you want data, passed from the GTRACE macro, to be recorded in a spool file.

**I/O**

Enables or disables external tracing of each I/O interrupt.

**PRG**

Enables or disables external tracing of each program interrupt.

**SIO**

Enables or disables external tracing of each request by the supervisor for I/O. This includes execution of the following instructions: SIO, DIAGNOSE I/O, TIO, CLRIO, HIO, HDV, SIOF, and TCH.

**Note:** The event is not recorded when the instruction is executed by an application program.

**SSS**

Enables or disables detailed external tracing of IUCV interrupts on the Signal System Service path.

**SVC**

Enables or disables external tracing of each SVC interrupt.

**SYN**

Enables external tracing of APPC/VM synchronous events.

**OFF**

Disables external tracing of events for the specified types.

Omitting the OFF operand enables external tracing of events for the specified type.

**GRoup**

Specifies that this command is to affect the virtual machine group, of which the issuer of the command is a member. If this operand is omitted, the command is applied only to the issuer's virtual machine.

If external tracing of certain types of events is enabled for the group, then they are automatically enabled for any virtual machine that may join the group later.

The GROUP operand can be used only by an authorized member of a virtual machine group. That is, by a member of the group placed on the list of authorized users in the GCS configuration file.

An unauthorized group member cannot deactivate tracing enabled by the GROUP operand. However, an authorized virtual machine can disable external tracing for itself although ETRACE with the GROUP operand was specified by another authorized virtual machine.

**Usage**

1. To enable external tracing, you must enter TRSOURCE in either BLOCK or EVENT mode and then enter ETRACE.
2. To disable external tracing in an orderly sequence, you should enter ETRACE END and then disable TRSOURCE, or trace records can be lost.
3. When ETRACE END is issued and EXTERNAL tracing is in BLOCK mode, the buffer is sent to CP.
4. When GCS loses control from a CP perspective (such as a system reset), the data in the buffer cannot be sent to CP and will not appear in the external trace records.

5. When running in EVENT mode, trace records do not get lost, but the performance gain of the BLOCK support is not realized.

### Examples

```
etrace all
.
.
.
etrace i/o prg off
```

Requests that all types of events for the issuer's virtual machine be recorded in a spool file. Later, a second ETRACE command, was issued to disable external tracing of I/O and program interrupts for the issuer's virtual machine.

```
etrace dsp i/o sio group
```

Requests that the following types of events be recorded externally for the virtual machine group: task dispatches, I/O interrupts, and GCS supervisor I/O requests. The individual who issues this command must be an authorized user, because the request is for the group.

```
etrace end
```

Requests that external tracing of events in a spool file for the issuer's virtual machine be disabled. This request will not be honored for an unauthorized user if the ETRACE events were started by the GROUP operand.

### Messages

- **GCTYTE001E** Invalid option '*option*' RC=4
- **GCTYTE009E** Operand missing or invalid RC=4
- **GCTYTE509I** ETRACE set ON for *event-types*
- **GCTYTE510I** ETRACE set ON for *event-types* for GROUP
- **GCTYTE511I** ETRACE set OFF for *event-types*
- **GCTYTE512I** ETRACE set OFF for *event-types* for GROUP
- **GCTYTE513E** ETRACE GROUP option is in effect for *event-types* RC=8

For more information on messages, see [z/VM: Other Components Messages and Codes](#).

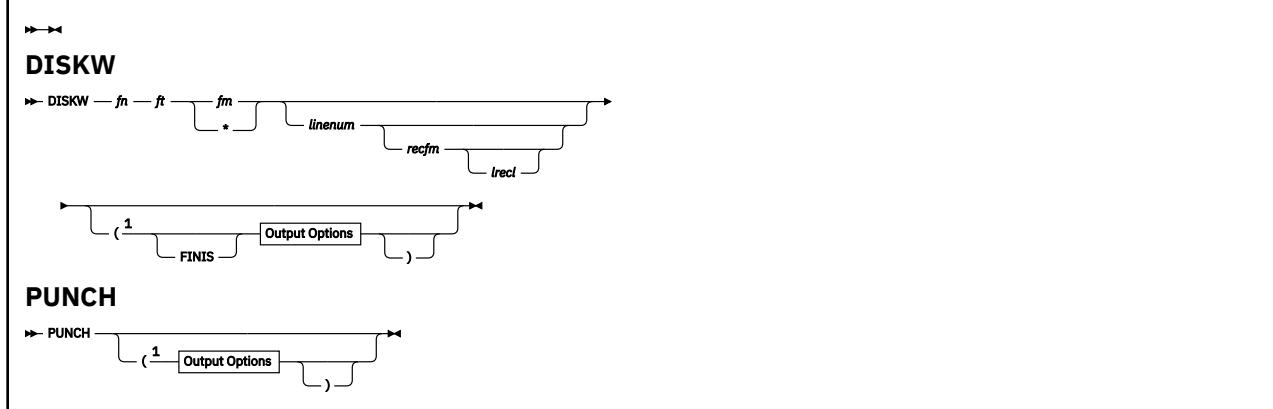
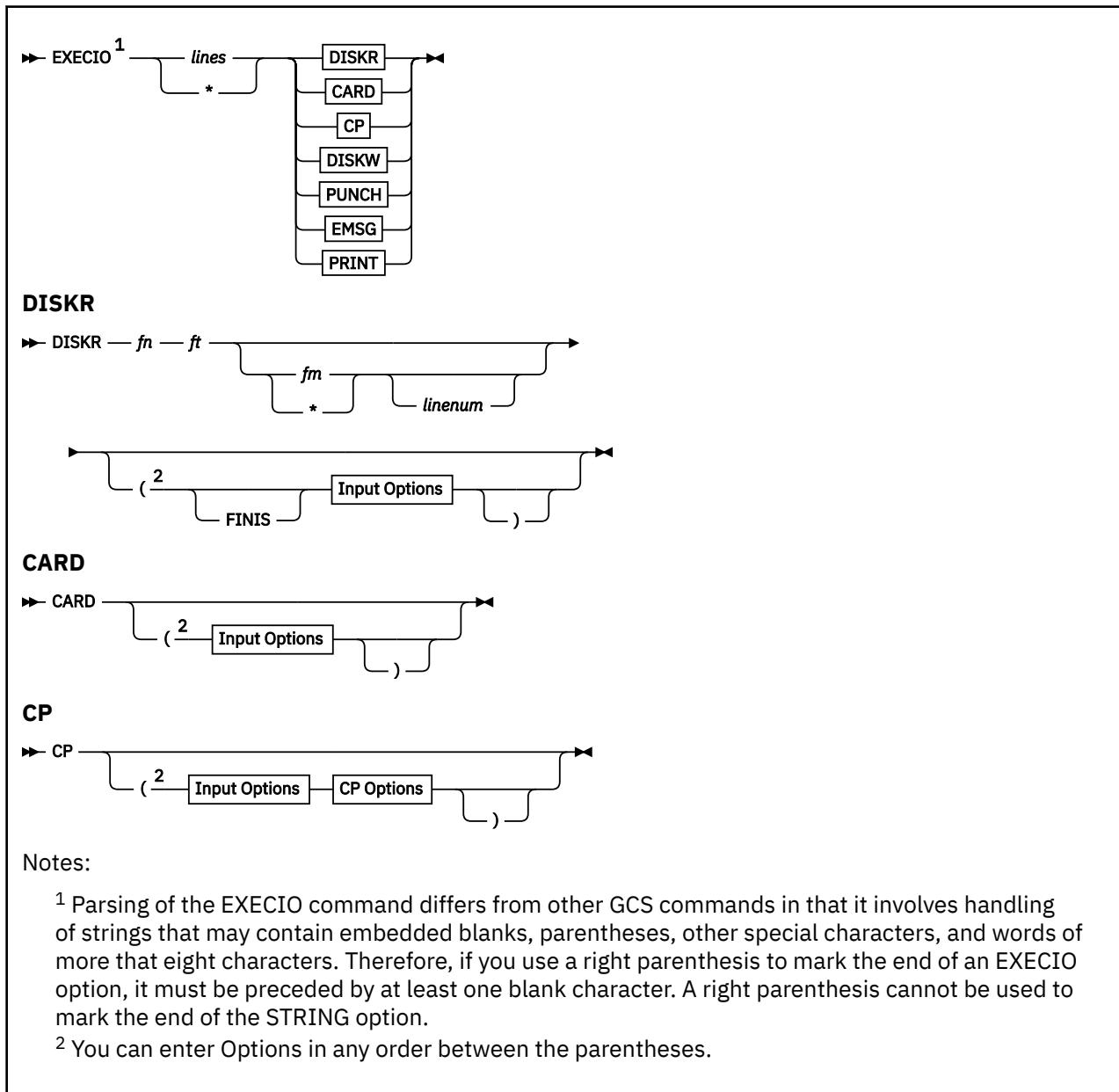
### Return Codes

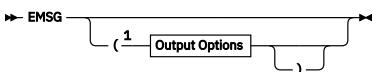
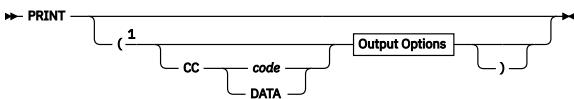
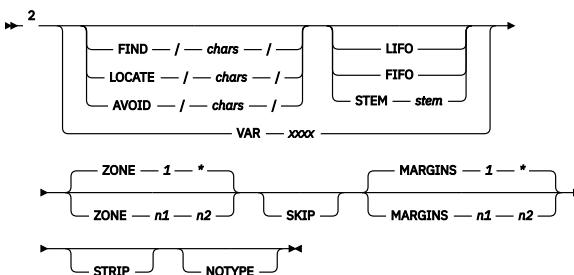
The meanings of return codes for these messages are:

Hex Code	Decimal Code	Meaning
X'00'	0	The specified ETRACE events have been successfully enabled or disabled.
X'04'	4	An invalid operand was specified, an unauthorized user specified the GROUP operand, the ETRACE command was issued before TRSOURCE was enabled. Your request was ignored.
X'08'	8	An authorized virtual machine had enabled external tracing using the GROUP operand. An unauthorized virtual machine then attempted to disable external tracing. The request was ignored.
X'10'	16	The trace buffer was not processed.

# **EXECIO**

## **Format**



**EMSG****PRINT****Input Options****CP Options**

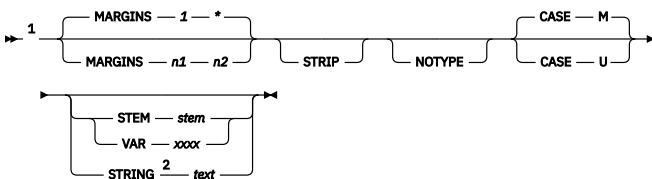
## Notes:

<sup>1</sup> You can enter Options in any order between the parentheses.

<sup>2</sup> Input options control how data is passed into your exec from EXECIO.

<sup>3</sup> If the STRING option is used, it must be the last option specified.

&gt;&gt;&gt;

**Output Options**

## Notes:

<sup>1</sup> Output options control how data is passed from your exec to EXECIO.

<sup>2</sup> If the STRING option is used, it must be the last option specified.

**Purpose**

Use the EXECIO command to:

- Read lines from a disk or virtual reader to the program stack or a variable.
- Write lines from the program stack or a variable to a CMS file or virtual spool device (punch or printer).
- Cause execution of CP commands and recover resulting output.

At times output data to be written may be supplied directly on the EXECIO command line.

The information immediately following is reference level information about EXECIO format and operands. Following this reference information you can find extended descriptive and use information. If you are not familiar with EXECIO, you should review the complete command description before attempting to use it. Also, to get full benefit from EXECIO, you should be familiar with use of execs under REXX. (See

the Appendix in [z/VM: REXX/VM Reference](#) for specific GCS REXX capabilities and for general information about REXX see [z/VM: REXX/VM Reference](#).)

In the following descriptions, "relative line number" means the number of lines processed to satisfy an EXECIO operation; "absolute line number" means the number of the line relative to the top of the file.

## Parameters

### **lines**

is the number of source lines processed. This can be any nonnegative integer. With the VAR option, the number of lines must be 1. An asterisk (\*) indicates that the operation is to terminate when:

- A null (0-length) line is read during an *output* operation
- An end-of-file condition is detected during an *input* operation.

Specification of \*, together with the STRING option, is valid only with the CP operand. Using the \* and STRING combination with any other operand causes an error message to be issued. Also the combination of the \* and the VAR option is not allowed. If *lines* is specified as zero (0), no I/O operation is performed other than FINIS, when it is specified as an option.

### **DISKR**

reads a specified number of lines from the CMS file *fn ft fm* to the program stack FIFO (first-in first-out) or to a REXX variable if the STEM or VAR options are specified.

#### **fn**

is the file name of the file.

#### **ft**

is the file type of the file.

#### **fm**

is the file mode of the file. When file mode is specified, that disk and its extensions are searched. If file mode is optional and is not specified, or is specified as an asterisk (\*), all accessed disks are searched for the specified file. If file mode is required and an asterisk (\*) is specified, the file with the specified file name and file type on the first accessed mode (in alphabetic order) will be opened. If no file is found that matches, EXECIO will be unsuccessful and an error message will be issued.

Because GCS checks for open files first, if you specify an asterisk for *fm*, you may get unexpected results if there are open files matching the file name and file type specified.

### **linenum**

is the absolute line number within the specified file where a DISKR or DISKW operation is to begin. If *linenum* has a value of zero or is not specified, reading begins at the first line and writing begins at the last line. For other files, reading or writing begins at the line following the one at which the previous operation ended. Because EXEC processors manipulate execs that are currently executing, a read or write to a currently running exec should explicitly specify the *linenum* operand. By not specifying this you may cause the first line to be read or written each time. If recfm or lrecl is specified for the DISKW operation, then a *linenum* value must be specified explicitly.

### **CARD**

reads a specified number of lines from the virtual reader to the program stack (FIFO) or to REXX variable if the STEM or VAR options are specified.

### **CP**

causes output resulting from a CP command to be placed on the program stack (FIFO) or to REXX variable if the STEM or VAR options are specified. To obtain the reply from a CP command, specify the *lines* operand as an asterisk (\*). If you want to enter a command to CP, suppressing messages and obtaining only the return code, specify the *lines* operand as zero (0). You may specify which CP command is to be issued using the:

- STRING option on the EXECIO command line

- Next line from the program stack.

Remember that all characters of CP commands must be in upper case.

### **DISKW**

writes a specified number of lines from the program stack or from REXX variable if the STEM or VAR options are specified to a new or existing CMS file *fn ft fm*.

Inserting a line into a variable length CMS file can cause truncation of the portion of the file following the inserted line. See the extended DISKW operand description.

#### ***fn***

is as described under the DISKR operand.

#### ***ft***

is as described under the DISKR operand.

#### ***fm***

is as described under the DISKR operand.

#### ***linenum***

is as described under the DISKR operand.

#### ***recfm***

#### ***lrecl***

define the record format and record length for any *new* file created by a DISKW operation. The default value for recfm is V (variable), in which case "lrecl" has no meaning. If you specify F (fixed) for recfm, the default lrecl value is 80. The maximum lrecl value that may be specified is 255 unless the VAR or STEM option is used to bypass the use of the program stack, then the maximum is the limit defined by the GCS file system which is the amount of storage available in your virtual machine below the 16MB line. If recfm or lrecl is specified for the DISKW operation, then a *linenum* value must be specified explicitly.

### **PUNCH**

transfers a specified number of lines from the program stack or from a REXX variable if the STEM or VAR options are specified to the virtual punch.

### **EMSG**

causes a message to be displayed. The text of the message may be the:

- Character string specified on the STRING option
- Next available line from the program stack
- Information from a STEM or VAR variable.

Messages are edited according to the current CP EMSG setting.

### **PRINT**

transfers a specified number of lines from the program stack or from a REXX variable if the STEM or VAR options are specified, to the virtual printer.

### **CC**

is used with the PRINT operand to specify carriage control for each line transferred to the virtual printer. Using the CC operand, you can supply carriage control code explicitly, or by specifying DATA, indicate that the carriage control character is the first byte of each line. If you omit the CC operand, a blank code serves as the default carriage control character. If CC is the last option specified and it is not followed by a code or DATA, then DATA is the default.

### **code**

is the character (ANSI or machine code) that defines carriage control. A blank code (the default value) cannot be specified on the command line.

### **DATA**

specifies that the first byte of each line sent to the virtual printer is a carriage control character.

**Defaults****AVOID**

is like the LOCATE option, except that the search is for a line (or zone portion of that line) that *does not* contain the specified characters.

**BUFFER *length***

specifies the length, in characters (bytes), of the CP command response expected from a CP operation. The limits of values that may be specified for *length* are 1 through the amount of storage available in your virtual machine below the 16MB line. If this option is not specified, up to 8192 characters of the response are returned.

**CASE**

causes data read from the program stack, from a STEM, or from a variable to be:

1. Translated to uppercase if U is specified
2. Not translated (mixed) if M is specified.

M (mixed) is the default value.

**FIND**

writes the following, LIFO (last-in first-out) to the program stack or FIFO (first-in first-out) to a REXX array:

1. The contents of the line that *begins with* the characters specified between delimiters
2. The line number of the first occurrence of that line (or zone portion of that line). For DISKR operations, both the relative and absolute line numbers are written. Otherwise, only the relative line number is written. FIND is case-sensitive.

If you wish to *search* only a portion of each line, use the ZONE option. If you wish to *write* only a portion of any line matching the search argument to the program stack or a variable, you can also use the MARGINS option.

**FINIS**

causes the specified file to be closed following completion of a DISKR or DISKW operation.

**LIFO****FIFO**

defines the order in which lines are written to the program stack. Generally, the default order is FIFO (first-in first-out). The exceptions are operations that put line numbers on the program stack because of a search operation (FIND, LOCATE, or AVOID). These operations default to LIFO (last-in first-out).

**LOCATE**

is like the FIND option explained previously, except that the object characters may occur any place within a line (or zone portion of that line), as opposed to only at the beginning of the line, as with the FIND option.

**MARGINS**

specifies that only a portion (columns n1 through n2 inclusive) of affected lines are to be processed (from the stack or a variable). The default values are column 1 through the end of each line (\*). The limits of values that may be specified for n1 or n2 are 1 through the maximum line length allowed by the operation being performed.

**NOTYPE**

suppresses the display of message GCTEIO632E at the virtual console.

**SKIP**

allows a read function (DISKR, CARD, CP) to occur without writing any information to the program stack.

**STEM *stem***

indicates that the specified *stem* defines variables used either to supply input data for EXECIO output-type operations (PUNCH, PRINT, EMSG, and DISKW), or as the destination for output of EXECIO input-type operations (CARD, DISKR, and CP).

*stem* is a regular REXX stem of the form *xxx..* For example, if the REXX stem is *xxx..*, then the variable *xxx.0* will represent the number of lines returned for input-type operations; *xxx.1* will contain the first line returned, *xxx.2* will contain the second, and so on.

STEM can be used with the STRING option only with the CP operation. If the STEM option is used on a print operation, then channel 9 or channel 12 indications returned from the hardware will be ignored. The maximum length variable name with the STEM option is 240 bytes.

#### **STRING *text***

supplies up to 255 characters of output data or a CP command explicitly on the EXECIO command line. Any characters following the STRING keyword are treated as string data, not additional EXECIO operands. Therefore, STRING, if specified, must be the final option on the command line.

#### **STRIP**

specifies that trailing blank characters are to be removed from any output lines or lines returned.

#### **VAR *xxxx***

indicates that the variable *xxxx* is to be used to supply input data for output-type operations (PUNCH, PRINT, EMSG, and DISKW) or that variable *xxxx* is the destination for output for the input-type operations (CARD, DISKR, and CP). If VAR is specified, then the number of lines must be 1. The maximum length variable name is 250 characters.

#### **ZONE**

restricts the portion of input lines searched by the FIND, LOCATE, or AVOID options. The search is between columns *n1* and *n2* (inclusive), if specified. The default values are column 1 through the end of the line (\*). The limits of values that may be specified for *n1* or *n2* are 1 through the maximum line length allowed by the operation being performed.

### **Messages**

- **GCTEIO621E** Bad Plist: Device and lines arguments are required RC=24
- **GCTEIO621E** Bad Plist: Disk '*argument*' argument is missing RC=24
- **GCTEIO621E** Bad Plist: Disk filemode required for DISKW RC=24
- **GCTEIO621E** Bad Plist: File format specified ('*recfm*') does not agree with existing file format ('*recfm*') RC=24
- **GCTEIO621E** Bad Plist: File lrecl specified ('*lrecl*') does not agree with existing file lrecl ('*lrecl*') RC=24
- **GCTEIO621E** Bad Plist: Input file '*fileid*' does not exist RC=24
- **GCTEIO621E** Bad Plist: Invalid character in file identifier RC=24
- **GCTEIO621E** Bad Plist: Invalid DEVICE argument ('*argument*') RC=24
- **GCTEIO621E** Bad Plist: Invalid EXEC variable name RC=24
- **GCTEIO621E** Bad Plist: Invalid mode '*mode*' RC=24
- **GCTEIO621E** Bad Plist: Invalid positional argument ('*argument*') RC=24
- **GCTEIO621E** Bad Plist: Invalid record format ('*recfm*') -- Must be either F or V RC=24
- **GCTEIO621E** Bad Plist: Invalid record length argument ('*lrecl*') RC=24
- **GCTEIO621E** Bad Plist: Invalid value ('*value*') for number of lines RC=24
- **GCTEIO621E** Bad Plist: Invalid value ('*value*') for disk file line number RC=24
- **GCTEIO621E** Bad Plist: Missing DEVICE argument RC=24
- **GCTEIO621E** Bad Plist: Option '*option*' can only be executed from a REXX EXEC RC=24
- **GCTEIO621E** Bad Plist: '*option*' option is not valid with '*option*' option RC=24
- **GCTEIO621E** Bad Plist: '*option*' option not valid with '*operation*' operation RC=24
- **GCTEIO621E** Bad Plist: STRING option with LINES=\* is valid only for CP operation RC=24
- **GCTEIO621E** Bad Plist: Unknown option name ('*name*') RC=24
- **GCTEIO621E** Bad Plist: Value ('*value*') not valid for '*option*' option RC=24

- **GCTEIO621E** Bad Plist: Value missing after ('*option*') option RC=24
- **GCTEIO621E** Bad Plist: VAR option with LINES > 1 is invalid
- **GCTEIO622E** Insufficient free storage for EXECIO RC=*rc*
- **GCTEIO632E** I/O error in EXECIO: rc=*nnnn* from *operation* operation RC=*rc*  
(See [“Explanation of Message GCTEIO632E” on page 91](#) for explanation of *nnnn* and *operation*.)

For more information, see:

- [“Extended Descriptions and Use Information” on page 83](#)
- [“EXECIO Return Codes” on page 90](#)
- [“EXECIO Abend Codes” on page 92](#)
- [“Explanation of Message GCTEIO632E” on page 91](#)

## Extended Descriptions and Use Information

EXECIO commands are usually issued as statements from REXX EXECs. Under GCS EXECIO can be executed from a REXX EXEC, through the CMDSI macro or by entering the command at the virtual console.

Remember that when a GCS task completes and the READY message (Ready;) displays, GCS closes all files used by EXECIO. Any subsequent EXECIO read operation will begin at file line one unless a "linenum" value is specified. Any subsequent EXECIO write operation will begin at the end of the file unless a "linenum" value is specified. Therefore, when possible, it is a good idea to specify a "linenum" value on the EXECIO command line.

For write operations, data to be written is usually taken from the program stack. However, data to be written may be supplied by the STRING option or by the VAR or STEM options (always the exec in question must be a REXX exec).

If the STEM option is used for a print operation, any channel 9 or channel 12 indications returned by the hardware will be ignored. When channel 9 or channel 12 is detected by the hardware, it will inform the issuer by a return code. A return code of 102 indicates that channel 12 was sensed; a return code of 103 indicates that channel 9 was sensed.

If the issuer of the EXECIO print operation needs to be able to handle channel 9 or 12 indications, the issuer should use the VAR, STRING option, or the program stack rather than STEM.

### **Program Stack:**

The program stack is a buffer area, expanded as necessary from available free storage. Data flow into and out of the program stack is:

1. Usually FIFO (first-in first-out) for read or write operations
2. LIFO (last-in first-out) for read options, such as FIND or LOCATE, that result in a line number being stacked.

A successful search (LOCATE, FIND, and so forth) operation results in two lines being written (LIFO) to the program stack:

1. The contents of the line that satisfied the search argument
2. The relative line number (number of lines read to obtain a match for the search argument), and for a DISKR operation only, the absolute line number (position from the top of the file).

Stacked line number values may be used on subsequent EXECIO operations for *lines* or *linenum* operands.

The QUEUED() built-in function can be used to return the number of lines in the program stack. See [z/VM: REXX/VM Reference](#) for information on the QUEUED() built-in function.

**Note:** When the stack is empty, EXECIO will request input data from you by using a WTOR.

For example:

```
Ready;
execio 1 diskw test file a
01 Enter input for command EXECIO
reply 1 This is a line for the test file.
Ready;
Ready;
```

In this example because the stack was empty, EXECIO prompts for the data by issuing a WTOR. Then a reply to the WTOR is returned to supply data for the EXECIO command. As a result, the test file contains the input entered in response to the WTOR. See ["REPLY" on page 147](#) for information on entering a reply to WTOR.

### **Setting Variables Directly:**

The VAR and STEM options allow REXX variable(s) to be set directly, bypassing the use of the stack. Data flow in and out of the variable(s) is:

- Always FIFO (first-in first-out) for read or write operations using STEM variables
- FIFO for read options, such as FIND or LOCATE, that result in a line number being returned.

A successful search (LOCATE, FIND, and so forth) operation results in two lines being assigned (FIFO) to EXEC variables if the STEM option has been used:

1. The contents of the line that satisfies the search argument
2. The relative line number (number of lines read to obtain a match for the search argument), and for a DISKR operation only, the absolute line number (position from the top of the file).

Returned line number values may be used on subsequent EXECIO operations for "lines" or "linenum" operands.

If the STEM option was specified, then variable xxxx.0 contains the number of lines of data returned.

If the command returns a nonzero return code, the variables specified by the command will be undefined and cannot be used.

#### ***Closing Files and Virtual Devices:***

EXECIO (DISKR or DISKW) operations from within a REXX EXEC or issued by the CMDSSI macro do not close referenced files when the operation terminates unless the FINIS operand is specified on the EXECIO command line.

There is considerable system overhead associated with the execution of FINIS. Therefore, if multiple references are to be made to a given file, it should be closed only when necessary.

If successive EXECIO commands reference a particular internal area of a CMS file, it is probably more efficient to let the file remain open until the last of these commands is issued. If this is done, each operation begins at the file line following the last line processed. This eliminates much of the need for calculating the "linenum" value.

When multiple tasks access the same file, they must be cooperating tasks. When a DETACH of a subtask is issued, all files used by that subtask through EXECIO are closed.

EXECIO does not close virtual spool devices. Therefore, to cause any spooled EXECIO output to be processed you must close the corresponding device. For example:

```
CP CLOSE PRINTER 00E
```

or:

```
CP SPOOL CLOSE 00E CLOSE
```

can be used to close the virtual printer after using the EXECIO PRINT function.

If an input spool file is read with the EXECIO CARD operation and the read is not completed (that is, the virtual machine does not get a last-card indication), you must enter a CP CLOSE READER command to be able to read that file again (or to read any other file). The file is purged unless you specify HOLD when you close a reader file. See the CP CLOSE command in the [z/VM: CP Commands and Utilities Reference](#).

If you have specified the PRINT operand and you try to write a line that is longer than the virtual device (PRINTER) allows, you will get error message GCTEIO632E.

#### ***lines Operand:***

For a DISKW, PUNCH, PRINT, or EMSG operation (if the STEM option had not been specified), if the *lines* operand exceeds the number of lines on the program stack, a WTOR is issued to the terminal. At that point, you must enter the balance of the lines (the number specified in the *lines* operand) from the terminal. Entering a blank character (null line) does not terminate the EXECIO operation; it writes a blank character to the output device. When the *lines* operand has been satisfied, the exec from which EXECIO was issued continues to execute.

If \* (to end of file) is specified for *lines* on an output operation, and you want the operation to terminate at any line in the program stack or a STEM array, you must make sure that line is null. Reading a null line terminates any of the four output operations if \* is specified for the *lines* operand.

For input operations (DISKR, CARD, and CP), the number of lines written to the program stack or the STEM array does not necessarily equal the number specified by *lines*. For example, an end-of-file or a satisfied search condition terminates a read operation, even if the specified number of lines has not been written to the program stack or the STEM array. When a search argument (FIND, LOCATE, AVOID option) is satisfied, and no SKIP option is specified, and the default stacking order (LIFO) is used, the line at the top (first line out) of the stack or the STEM array contains the number of operations required to satisfy the search. The next line contains the line that satisfied the search.

If the search argument (FIND, LOCATE, AVOID option) is not satisfied, a return code of 3 is given, even if EOF occurs before the specified number of lines has been read. A return code of 3 is also given if \* is specified for *lines* on a read operation, and the search argument is not satisfied.

When a number greater than 0 is specified for *lines* with output operation CP, and the number of lines written to the program stack, stem array, or variable name is not equal to the number specified by *lines*, a return code of 2 is given.

When \* is specified for *lines* on a read operation, the operation is terminated at end-of-file. A return code of 0 is given because the \* is an explicit request to read to end-of-file.

When a search argument (FIND, LOCATE, and AVOID options) is not satisfied and an end-of-file situation occurs for the EXECIO CARD operation, the reader file is purged unless a CP SPOOL READER HOLD was previously specified. For more information on how spool files are processed, see the CP CLOSE and CP SPOOL commands in the [z/VM: CP Commands and Utilities Reference](#).

#### ***DISKR Operation:***

The first line read on a DISKR operation may be:

- The first line of the specified file
- Specified using the "linenum" operand
- Determined by the results of a previous operation.

The DISKR operation may be used to simply read a specified number of lines from a specified file and write them to the program stack or a variable. For example, suppose file MYFILE DATA contains:

```
The number one color is red
The number two color is yellow
The number three color is green
The number four color is blue
The number five color is black
```

The command:

```
EXECIO 2 DISKR MYFILE DATA * 1
```

writes to the program stack (FIFO) two lines beginning with line one, like this:

```
|. The number one color is red | .<-next line read
|. The number two color is yellow | .
|.
|.
|.
```

However, a little more complex version of this command:

```
EXECIO 2 DISKR MYFILE DATA * 3 (LIFO MARGINS 5 14
```

would have resulted in this program stack:

```
|. number fou | .<-next line read
|. number thr | .
|.
|.
|.
```

Notice in the preceding example the use of \* as a file mode operand on the command lines to serve as a place holder. The command:

```
EXECIO 2 DISKR MYFILE DATA * 1 (STEM X.
```

assigns to variables X.1 and X.2 one line each, beginning with line one, like this:

```
|. The number one color is red |. X.1  
|. The number two color is yellow |. X.2  
|.  
|.  
|.  
|.
```

The X.0 variable contains the number of lines (in this example, 2).

When a line satisfies the LOCATE, FIND, or AVOID option for a *DISKR* operation, EXECIO writes that line to the program stack (LIFO) or a variable (FIFO) and in an additional stack line or variable, writes the relative (number of lines read to satisfy the search) and absolute (position from the top of the file) line numbers.

#### ***CP Operand:***

When a search argument is required, the CP operand uses the FIND, LOCATE, and AVOID options to process output resulting from the associated CP command. Each line that satisfies the search criteria is written to the program stack or a variable. When data exceeds 8192 characters, it is truncated on a line basis and an error code is returned. If you specify the BUFFER option, data is truncated on a line basis after the number of characters specified in *length* or 8192 is reached, whichever is greater. Each line returned ends with a X'15' character. This must be allowed for when calculating the buffer size needed. The number of read operations required to match the search argument is written to the next stack line.

If you do not supply the CP command to be issued by the STRING option, the next line in the program stack is treated as that command. If there are no lines in the program stack, a WTOR is issued to the terminal. The reply to the WTOR is treated as the CP command. If the reply consists of a null line, the operation terminates. See "["REPLY" on page 147](#)" for information on entering a reply to WTOR.

The responses from the CP command are treated as input. If CP SET IMSG is set OFF, no response is issued by some CP commands. This may result in a return code of 2, if a number other than zero (0) is specified for *lines*. The return code of 2 indicates that the end of the input file was reached before the specified number of lines could be read. This will not occur if you specify the *lines* operand as an asterisk (\*). For more information regarding which CP commands are affected by the setting of IMSG see the CP SET command in the [z/VM: CP Commands and Utilities Reference](#).

Remember that all characters of CP commands must be uppercase.

ZONE and MARGINS options do not affect the reading of the CP command; however, they do affect the portions of the lines processed as a result of the command execution.

#### ***DISKW Operand:***

The DISKW operand causes the next lines from the program stack to be written to a CMS file. The point at which writing begins in an existing file on a DISKW operation may:

1. Follow the last file line (default "linenum" when writing to a newly opened file, for example)
2. Be specified using the "linenum" operand
3. Be determined by the results of a previous operation.

For example, suppose you want to write 10 lines from the program stack to the end of an existing file, BUCKET STACK A, on your disk accessed as A. Your exec file statement to do this would be:

```
EXECIO 10 DISKW BUCKET STACK A
```

Now, take a slightly more complex requirement. Using stack lines down to the first null line, create a new file, BASKET STAX A, then close the file after it is written. Also, make the file fixed length format with a record length of 60. The EXECIO command to do this is:

```
EXECIO * DISKW BASKET STAX A 1 F 60 (FINIS
```

A word of caution about using the *linenum* operand to insert lines in the middle of CMS variable length files. Because of the way GCS handles these files, any variable length line inserted must be equal in length to the line it displaces. Otherwise, all lines following the one inserted are truncated.

For example, assume the variable format file WORDS LEARNING A is:

```
A is for apple
C is for cake
C is for candy
D is for dog
```

execution of:

```
EXECIO 1 DISKW WORDS LEARNING A 2 (STRING B is for butterfly
```

produces a file that contains only:

```
A is for apple
B is for butterfly
```

Because "B is for butterfly" contains more characters than the line it writes over, "C is for cake", all lines following it are truncated. However, slightly modifying the command to:

```
EXECIO 1 DISKW WORDS LEARNING A 2 (STRING B is for baby
```

results in:

```
A is for apple
B is for baby
C is for candy
D is for dog
```

To prevent truncation when inserting records in a variable-length file, you can use fixed-format files.

#### ***recfm lrecl operands:***

The default value for *recfm* is V (variable), in which case "lrecl" has no meaning. If you specify F (fixed) for *recfm*, the default *lrecl* value is 80. The maximum *lrecl* value that you may specify is 255 unless the VAR or STEM option is used to bypass the use of the program stack, in which case the maximum is the limit determined by the amount of storage available in your virtual machine below the 16MB line.

When lines are written to an existing file, the record format and record length of that file apply. Specifying *recfm* or *lrecl* values on the EXECIO command line that conflict with those of the existing file causes an error message to be issued.

#### ***CC Operand:***

When you specify CC together with the DATA operand, be sure the first character of each line to be sent to the virtual printer may be removed and interpreted as carriage control for that line.

You may use ANSI or machine code characters with the CC operand to specify carriage control as shown by [Table 9 on page 87](#). In this table the first character of the line is interpreted as a carriage control character.

*Table 9. Valid ANSI Control Characters for Carriage Control*

<b>Character</b>	<b>Hex Code</b>	<b>Meaning</b>
blank	40	Space 1 line before printing
0	F0	Space 2 lines before printing
-	60	Space 3 lines before printing
+	4E	Suppress space before printing
1	F1	Skip to channel 1

Table 9. Valid ANSI Control Characters for Carriage Control (continued)

Character	Hex Code	Meaning
2	F2	Skip to channel 2
3	F3	Skip to channel 3
4	F4	Skip to channel 4
5	F5	Skip to channel 5
6	F6	Skip to channel 6
7	F7	Skip to channel 7
8	F8	Skip to channel 8
9	F9	Skip to channel 9
A	C1	Skip to channel 10
B	C2	Skip to channel 11
C	C3	Skip to channel 12

For example, CC 0 causes space two lines before printing.

If you are using EXECIO with either the VAR, STRING option, or the program stack and the virtual FCB defines channel 9 or channel 12, it may be necessary to reset the carriage control. When channel 9 (return code 103) or channel 12 (return code 102) is sensed, the write operation terminates after carriage spacing, but before writing the line. If you are printing from the program stack, then the line will remain in the stack. The carriage control character should be modified to take the appropriate action (for example, skip to channel, or print with no space).

If your virtual printer is a device type which reflects channel code sensing back to you, the sensing of channel code 12 or 9 results in a return code of 2 or 3 from PRINT, which EXECIO reflects as return code 102 or 103. For these type conditions, the following options are available for you to handle recovery:

- Code the application to examine the return code from EXECIO and retry the print operation if a channel code 12 or 9 has been detected.
- Redefine the virtual printer to a device type that does not reflect channel 12 or 9 sensing.
- Redefine the FCB for the printer to eliminate the channel code 12 or 9.

#### **EMSG Operand:**

Lines to be displayed by EMSG should have the format:

xxxxmmmnns

where:

**xxxxmm**

is the issuing module name

**nnn**

is the message number

**s**

indicates the message type (E - error, I - informational, W - warning, and so forth)

The current settings of the CP SET EMSG command control the displayed lines. These settings, combined with message length, can cause messages to be abbreviated or not displayed at all.

#### **linenum Operand:**

When a *linenum* value (default 0) is not specified on the EXECIO command line, the number of the next file line available for reading or writing depends on results of previous operations that referenced that file.

For example, consider the two EXECIO DISKR operations just following. By looking at the first of these commands you can see:

- Four lines are to be read from MYFILE DATA, starting at line 1
- Because FINIS is not specified on the command line, MYFILE DATA remains open after the first read operation. Because the first command reads 4 lines, the subsequent read operation will begin at line 5.

```
EXECIO 4 DISKR MYFILE DATA * 1
.
.
.
EXECIO 3 DISKR MYFILE DATA (FINIS)
```

Because the second EXECIO command specifies no *linenum* operand, reading of the specified 3 lines begins at line 5.

Two situations that would cause the second EXECIO command to not begin execution at line 5 are:

- A program other than EXECIO accessing MYFILE DATA after the first and before the second EXECIO command is executed.
- A GCS operation except WTOR completing so the GCS READY message (Ready;) is displayed. In that case GCS closes associated files. Therefore, subsequent operations using these files would begin at line 1.

The FINIS operand causes MYFILE DATA to close. Therefore, any subsequent DISKR operation using a default *linenum* value would begin reading at line 1.

#### **FIND, LOCATE, AVOID options:**

The delimiter pair for the specified character string need not be //. They may be any character not included in the string. For example:

```
EXECIO * DISKR MYFILE DATES (LOCATE $12/25/81$
```

#### **FIFO, LIFO options:**

Most EXECIO operations that write to the program stack default to FIFO, first line written to the stack will be the first read out. The exceptions (LIFO) are operations involving a search (LOCATE, FIND, and AVOID options). These operations result in the relative line number (number of lines read to satisfy the search) being stacked. For DISKR operations the absolute line number (position from the top of file) is also stacked on the same line. It is necessary to have these numbers at the top of the stack so that they are immediately accessible to a subsequent EXECIO command.

#### **SKIP Option:**

On EXECIO read operations the SKIP operand prevents input lines from being written to the program stack or a variable. For example, you might want to put on the program stack all lines of MYFILE DATA that follow the line containing "4120 Rock Road". First, to search through the file for the line after which reading to the program stack is to begin, enter:

```
EXECIO * DISKR MYFILE DATA * 1 (LOCATE /4120 Rock Road/ SKIP
```

The SKIP option prevents the line being searched for, together with the line number, from being written to the program stack. Then, to write to the program stack the next line through the end of file, issue:

```
EXECIO * DISKR MYFILE DATA
```

Remember that accessing MYFILE DATA by another program or causing a GCS READY message to be displayed, except from WTOR, before issuing the second EXECIO command would change the point at which the second command begins reading. When possible, you should specify the *linenum* operand explicitly.

## EXECIO

Another use of the SKIP option might be the execution of a CP command using the CP operand to obtain a return code without displaying the resulting messages or writing them to the program stack or a variable. For example:

```
EXECIO * CP (SKIP STRING Q userid
```

The user ID must be uppercase.

As an alternative, specifying 0 for the *lines* operand value with the CP operand also causes results not to be displayed or written to the program stack.

### **STEM Option:**

The STEM option lets an array of REXX variables be set directly, bypassing the stack. For example, if you want the first 3 lines of MYFILE DATA to be assigned to REXX variables X.1, X.2, and X.3, enter:

```
'EXECIO 3 DISKR MYFILE DATA * 1 (STEM X.'
```

Variable X.1 now contains the first line of MYFILE DATA, variable X.2 contains the second line, and variable X.3 contains the third line. Variable X.0 contains the number of lines (in this example, 3). For REXX variables, the variable name should be enclosed in quotation marks and must be in uppercase.

For the STEM option, the maximum length variable name is 240 bytes.

### **VAR Option:**

On EXECIO operations, the VAR option allows a REXX variable to be set directly, bypassing the use of the stack. For example, if you want the second line of MYFILE DATA to be assigned to a variable named X, issue:

```
'EXECIO 1 DISKR MYFILE DATA * 2 (VAR X'
```

Variable X now contains the second line of MYFILE DATA.

For REXX variables, the variable name should be enclosed in quotation marks and must be in uppercase.

For the VAR option, the maximum length variable name is 250 bytes.

## EXECIO Return Codes

Return Code	Meaning
0	Finished correctly
1	Truncated
2	EOF before specified number of lines were read
3	Count ran out without successful pattern match
24	Bad Parameter List (PLIST)
41	Insufficient free storage for EXECIO
nnn	100 + return code from I/O operation (if nonzero)
2008	Variable name supplied on STEM or VAR option was invalid.
x1nnn	1000 + return code from CP command (if nonzero), where x is 0, 1, 2, or 3, as previously described
1xnnnn	100000 + return code from CP command (if nonzero), where x is 0, 1, 2, or 3, as previously described

## Explanation of Message GCTEIO632E

I/O error in EXECIO: rc=nnnn from *operation* operation

When the *operation* is:

### CARD

nnnn =

**2**

unit check, intervention required

**3**

I/O error (GCTCIO will issue a message)

**108**

Reader not attached at 00C

**112**

Reader is busy

### CP

nnnn =

See z/VM: CP Commands and Utilities Reference for more information on return codes.

### DISKR

nnnn =

**3**

Permanent I/O error

**8**

Truncation occurred

**12**

End of file, or record greater than the number of records in data set

**25**

Insufficient free storage available for file management control areas

**26**

Requested item number is negative or item number plus number of items exceeds file system capacity

### DISKW

nnnn =

**3**

Permanent I/O error

**4**

Invalid mode specified or disk is not accessed

**7**

Attempt to skip over unwritten variable-length item

**12**

Attempt to write to read-only disk or disk is not accessed

**13**

Disk is full

**15**

Length of fixed length item not the same as previous item

**17**

Variable length item greater than 65535 bytes

**22**

Virtual storage capacity exceeded

## **EXECIO**

**25**

Insufficient free storage available for file directory buffers

**26**

Requested item number is negative or item number plus number of items exceeds file system capacity

**27**

Attempt to update variable length item with one of different length

### **EMSG**

nnnn = No return code possible from EMSG routine

### **EXECCOMM**

nnnn =

**-2**

Insufficient storage is available to process your request

**8**

Invalid variable name

### **FINIS**

nnnn =

**6**

File not open (or no read or write was issued to the file), or invalid file ID (fn ft fm) specified.

### **PRINT**

nnnn =

**1**

line too long

**2**

Channel 12 punch detected

**3**

Channel 9 punch detected

**4**

Intervention required

**5**

I/O operation was unsuccessful (GCTPIO issues message)

**108**

Printer not attached at 00E

**112**

Printer is busy

### **PUNCH**

nnnn =

**2**

unit check, intervention required

**3**

I/O error (GCTCIO will issue a message)

**108**

Punch not attached at 00D

**112**

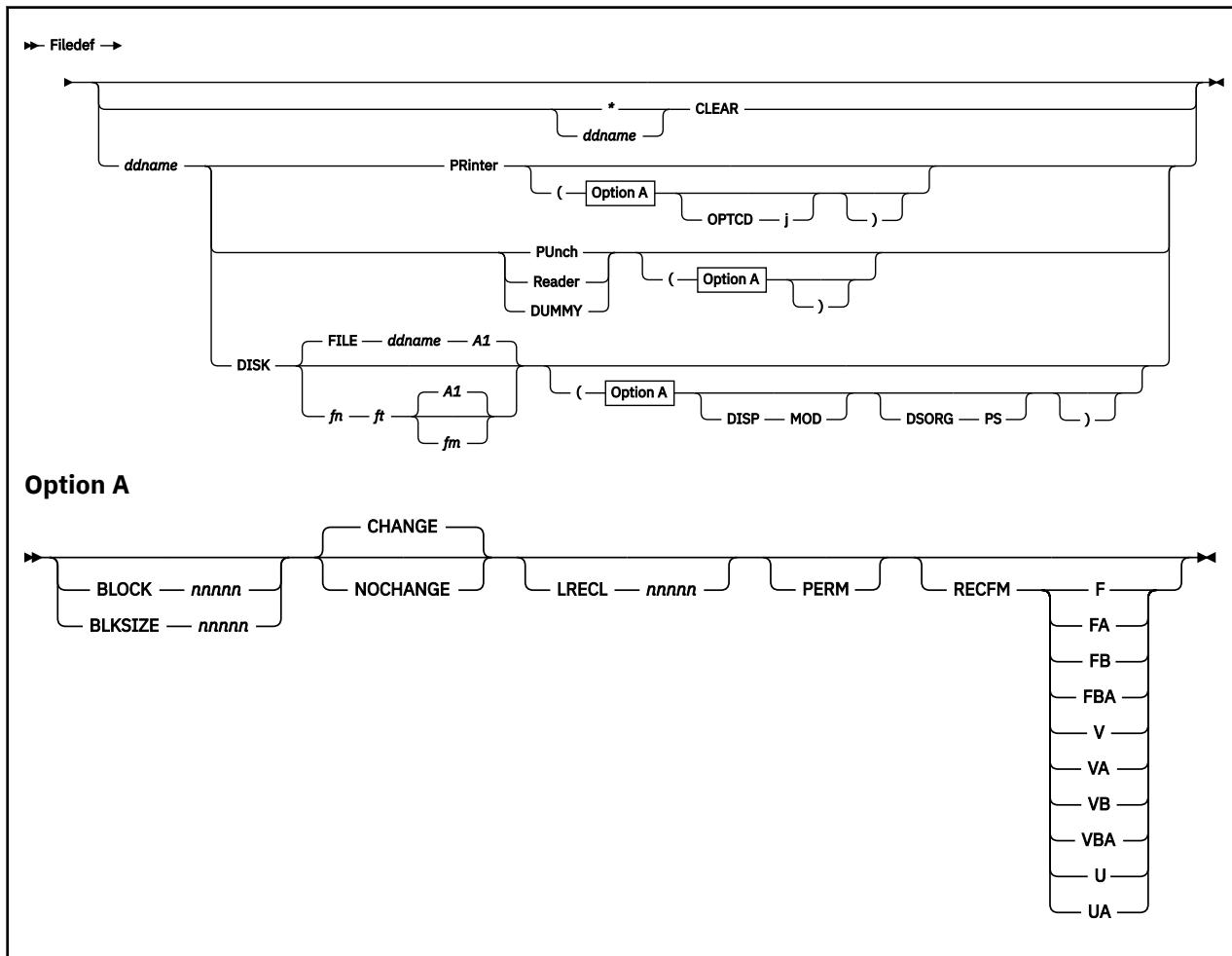
Punch is busy

## **EXECIO Abend Codes**

<b>ABEND Code</b>	<b>Reason Code</b>	<b>Meaning</b>
FCA	0C00	You are not authorized to access the storage specified in the parameter list.

# FILEDEF

## Format



## Purpose

Use the FILEDEF command to define CMS format files and spool files.

Application programs usually require some *setting up* before you can start and run them. The FILEDEF command is one of the preliminary commands issued to prepare a program for execution. You enter it to define CMS format files and spool files used by the program.

## Operands

### *ddname*

The name of the file, as referred to in your program. The *ddname* can contain one to eight alphanumeric characters. However, the first one must be alphabetic or national.

\*

If you specify an asterisk (\*) instead of a *ddname* and follow it with the CLEAR operand, all file definitions that you did not enter with the PERM option will be cleared.

**CLEAR**

Removes any existing definition for the specified ddname that is owned by the current task. You should clear ddnames before defining them to make sure the ddname does not already exist. Doing that cancels any operations previously defined with the ddname.

**PRINTER**

Represents the spooled printer, which you must have defined at virtual address 00E.

**OPTCD j**

When the virtual printer is a 3800, j indicates to QSAM and BSAM that the output data line's first byte will contain a table reference character (TRC). This TRC selects a character arrangement table to use in printing the data line. The TRC can be alone or with other ANSI control characters.

**PUNCH**

Represents the spooled punch, which you must have defined at virtual address 00D.

**READER**

Represents the spooled card reader, which you must have defined at virtual address 00C. (I/O to the card reader must not be *blocked*.)

**DUMMY**

Indicates that no real I/O takes place for the data set.

**DISK**

Specifies that the virtual I/O device is a disk. *fn* and *ft* are CMS fields. If you omit DISK *fn ft*, the default is FILE *ddname* A1.

**DISP MOD**

Positions the read/write pointer after the last record in a disk file. Use this option only when you are adding records to the end of a file. That file must be on a disk accessed as read/write. The disk cannot be an extension of another disk. If so, it would be read/only, and you could not write to it.

**DSORG PS**

Specifies that the data set has a physical, sequential (PS) organization.

## Options

**BLOCK nnnnn****BLKSIZE nnnnn**

Specifies the maximum block length in bytes. For fixed length records, this is the record length. For variable length records, this gives the maximum logical record length (up to 32756 bytes, plus 4 bytes for a block descriptor word). For undefined length records, this value can be altered by the problem program. It can be inserted directly into the data control block or specified in the length operand of a READ/WRITE macro.

**CHANGE**

Combines definitions for an existing ddname with new ones when you enter a new FILEDEF for that same ddname. All options from both definitions are merged. A new definition for a particular option replaces the original definition.

The CHANGE option is not valid when the new FILEDEF is issued for a ddname that another task has already issued a FILEDEF for, or when the associated file has already been opened.

**NOCHANGE**

Retains the current file definition, if one exists, for a specified *ddname*. With this option, the system stops further processing (error checking, scanning, and similar functions) for new FILEDEF commands with the same *ddname*.

**LRECL nnnnn**

Specifies the length, in bytes, of each fixed length logical record or the maximum length, in bytes, for variable length logical records. This value should not exceed 32760 bytes for fixed length records or 32756 (including four bytes for a record descriptor word) for variable length records.

**PERM**

Retains the current file definition until it either is explicitly cleared or is changed by a new FILEDEF command with the CHANGE option. If you do not specify PERM, the definition is cleared when you enter FILEDEF \* CLEAR.

**RECFM**

Represents the record format of the file, where *a* can be one of these:

**Type is:****The file contains:****F**

Fixed length records

**FA**

Fixed length records with American National Standards Institute (ANSI) characters

**FB**

Fixed length, blocked records (not for use with READER devices)

**FBA**

Fixed length, blocked records with ANSI characters

**V**

Variable length records

**VA**

Variable length records with ANSI characters

**VB**

Variable length, blocked records (not for use with READER devices)

**VBA**

Variable length, blocked records with ANSI characters

**U**

Records of an undefined length

**UA**

Records of an undefined length with ANSI characters.

For more information on using this command and its operands and options, see [z/VM: CMS Commands and Utilities Reference](#)

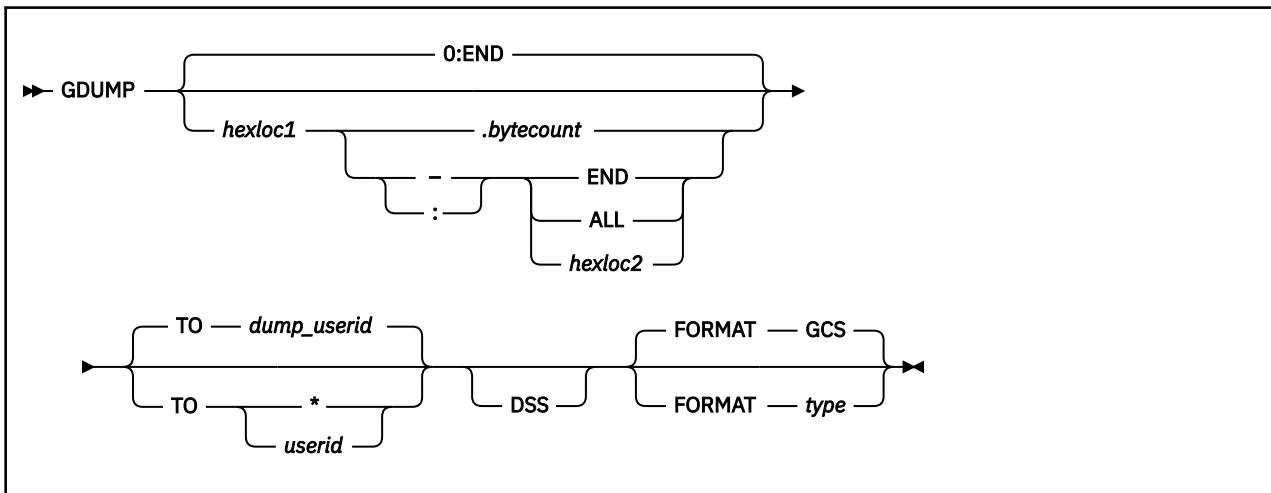
## Messages

- **GCTFLD001E** Invalid option '*option*' RC=24
- **GCTFLD002E** Invalid parameter '*parameter*' in the option '*option*' field RC=24
- **GCTFLD003E** '*option*' option specified twice RC=24
- **GCTFLD004E** '*option1*' and '*option2*' are conflicting options RC=24
- **GCTFLD005S** Virtual storage capacity exceeded RC=104
- **GCTFLD006E** Invalid parameter '*parameter*' RC=24
- **GCTFLD011E** Invalid character in fileid '*fn ft*' RC=20
- **GCTFLD017E** Disk mode not accessed RC=36
- **GCTFLD021E** Invalid mode '*mode*' RC=24
- **GCTFLD023E** No file type specified RC=24
- **GCTFLD301E** Invalid device '*device name*' RC=24
- **GCTFLD302E** Parameter missing after DDNAME RC=24
- **GCTFLD303I** No user defined FILEDEFS in effect
- **GCTFLD304I** Invalid CLEAR request
- **GCTFLD320E** Error during FILEDEF CLEAR processing, DCBs not closed RC=40

For more information on messages, see [z/VM: Other Components Messages and Codes](#).

# GDUMP

## Format



## Purpose

Use the GDUMP command to produce a copy of the contents of your virtual machine's storage.

## Parameters

### *hexloc1*

The hexadecimal address in virtual storage where the dump is to start. If no starting address is specified, 0 is assumed.

### *.bytecount*

Specifies the number of bytes to be included in the dump. No embedded blanks are allowed.

If you wanted 65597 (X'1003D') bytes of storage, dumped starting at address X'4F023', you would use:

```
4F023.1003D
```

### **- (dash)**

### **: (colon)**

The dash and colon are range indicators that specify a range of storage to be dumped.

If the storage you want to dump begins at address X'4F023' and ends at X'5F05F', you could express the range this way:

```
4F023-5F05F
```

or:

```
4F023:5F05F
```

Embedded blanks are not allowed.

### **END**

Specifies that the dump is to end at the last address of virtual storage.

If you omit the *hexloc2* and END parameters, END is assumed.

**ALL**

Specifies that the dump is to end at the last address of the virtual machine's address space. The dump includes virtual machine storage plus saved segments.

**hexloc2**

The hexadecimal address in virtual storage where the dump is to end. This must be preceded by (and adjoined to) a dash or colon. If you do not specify it, and either the colon or dash is used, then the last address in virtual storage is assumed.

**Note:** Dumps are always generated in 4KB pages. These pages correspond to the 4KB pages into which storage is segmented. If you request that a certain portion of storage be dumped, the entire 4KB page into which that portion falls is included in the dump. So, your request is always rounded up and down to the nearest page boundaries.

**TO *dump\_userid***

Specifies the dump is to be sent to the user ID, which was predefined in the GROUP EXEC. Also, note that this is the default.

**TO \***

Specifies that you want the dump sent to the virtual reader of the machine that is issuing this GDUMP command.

If the issuer of an GDUMP command (with TO \* specified) is not on the list of authorized user IDs (specified with the GROUP EXEC), any fetch protected data that does not have a storage key of 14 is omitted from the dump. However, all requested nonfetch protected data and Key 14 storage is included.

**TO *userid***

Specifies that you want the dump sent to the virtual reader of a specific user (even if your group has a common dump receiver).

If the user ID receiving the dump is not on the list of authorized user IDs (specified with the GROUP EXEC), fetch protected data is omitted from the dump. However, all requested nonfetch protected data and key 14 storage is included.

Unauthorized user IDs can request a dump containing fetch protected data and send it to an authorized receiver. That way, the fetch protected data will be included. However, those unauthorized user IDs are prevented from using the CP TRANSFER command to transfer the dump-containing spool file to their own machines.

If you do not specify TO, the dump goes to the common dump receiver (if you specified one with the GROUP EXEC). Otherwise, it goes to the virtual reader of the machine issuing the GDUMP command.

**DSS**

Specifies that any saved systems, or discontinuous saved segments, in your machine (the one where you were issuing the command) be included in the dump.

**FORMAT *type***

Describes the type of virtual machine contents you are dumping (CMS, GCS, RSCS, or another type). This format type will later become the DVF format type of the dump.

If you omit this operand, a format type of GCS is assumed.

**Usage**

No dump will be produced if dumps are suppressed through the SET DUMP OFF command.

**Examples**

```
gdump 0:CB8F7 TO * DSS
```

Requests a dump of the issuer's virtual storage contents, from address 0 to CB8F7, and sends it to the issuer's own virtual reader. This dump includes any discontinuous saved segments the virtual machine

may be using and, if the user ID is authorized, any fetch-protected data (other than key 14) that can be found within the specified address range. The virtual machine type is GCS (the default).

```
gdump
```

Requests a dump of the issuer's virtual storage contents (excluding any discontiguous saved segments) and sends it to the common dump receiver. If the common dump receiver is an unauthorized user ID, no fetch-protected data other than key 14 will be included in the dump.

## Messages

- **GCTDUM009E** Operand is missing or invalid RC=12
- **GCTDUM010I** Command Complete
- **GCTDUM363E** Dump suppressed via SET DUMP OFF command RC=32
- **GCTDUM525E** Userid is missing or invalid RC=20
- **GCTDUM526E** Userid '*user ID*' not in CP directory RC=16
- **GCTDUM527E** Invalid range RC=24
- **GCTDUM529E** Partial dump taken RC=4
- **GCTDUM531E** Dump failed: spooling error RC=8
- **GCTDUM532E** Dump failed: I/O error RC=28
- **GCTDUR528I** Dump complete
- **GCTDUR529E** Partial dump taken
- **GCTDUR530E** Dump failed
- **GCTDUR362I** Dump suppressed

For more information on messages, see [z/VM: Other Components Messages and Codes](#).

## Return Codes

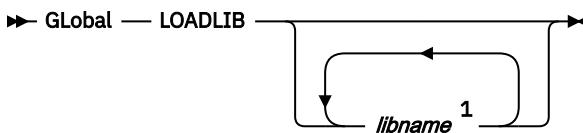
The meanings of return codes for these messages are:

Hex Code	Decimal Code	Meaning
X'00'	0	The dump finished successfully. All requested areas were recorded in the dump.
X'04'	4	Not all the requested areas were recorded in the dump.
X'08'	8	A spooling error in CP prevents the dump from being recorded.
X'0C'	12	An operand is missing or invalid.
X'10'	16	The recipient's user ID is not in the CP directory.
X'14'	20	The TO operand was specified but the user ID was missing or invalid.
X'18'	24	An invalid address range was specified.
X'1C'	28	CP experienced an I/O error when paging in the parameter list or dump list. No dump was recorded.
X'20'	32	Dump suppressed.

# GLOBAL

---

## Format



Notes:

<sup>1</sup> You can specify a maximum of 63 libnames.

## Purpose

Use the GLOBAL command to define the CMS load libraries you want searched for modules.

Programs you run under GCS may be members of CMS load libraries. Before GCS can call a program residing in a load library, you must identify the load library where it can be found.

Use the GLOBAL command to specify what load libraries GCS should search whenever you attempt to start a program.

## Operands

### LOADLIB

An operand indicating that you are referring to CMS load libraries.

### libname

The file names of the load libraries you want searched for modules. No more than 63 load libraries may be specified in the GLOBAL command. Whenever the load libraries are searched, they are searched in the order they are specified in this command.

If no library names are specified, the command cancels the effects of any previous GLOBAL command.

To find out what load libraries are currently identified to be searched, type:

```
query loadlib
```

## Messages

- **GCTGLB005S** Virtual storage capacity exceeded RC=104
- **GCTGLB013E** No function specified RC=24
- **GCTGLB014E** Invalid function '*function*' RC=24
- **GCTGLB024E** File '*fn ft fm*' not found RC=28
- **GCTGLB220E** Unable to open file '*filename*' RC=28
- **GCTGLB221S** More than *nnn* libraries specified RC=88
- **GCTGLB222E** File '*fn ft fm*' contains invalid record formats RC=32
- **GCTGLB223S** Error '*xx*' reading file '*fn [ft fm]*' from disk RC=100

For more information on messages, see [z/VM: Other Components Messages and Codes](#).

# GROUP

## Format



## Purpose

Use the GROUP command to set up a Group Control System (GCS) configuration file for building a GCS nucleus. You can invoke GROUP from the CMS environment. GROUP displays a series of panels on which you enter information about your virtual machine group.

## Operands

### **systemname**

is an optional parameter that specifies the file name that you want to assign to the configuration file. If you enter this parameter with the GROUP command, then the Primary Option Menu panel appears with the SYSTEM NAME field filled in. If you enter the GROUP command without this parameter, then the field called SYSTEM NAME displays GCS on the Primary Option Menu panel. The default system name entry is GCS.

## Usage

1. You manually invoke the GROUP command to display the GCS configuration files.
2. If you do not have a full-screen display device, you cannot use the GROUP panels to build the GCS configuration file. You must build the file manually, using the build macros.

## Messages and Return Codes

### **GCTGRP007E**

Extraneous parameter '*parameter*' RC=24

### **GCTGRP018E**

Disk '*mode*' is Read/Only RC=36

### **GCTGRP019E**

No Read/Write '*mode*' disk accessed RC=36

### **GCTGRP024E**

File '*fileid*' not found RC=36

### **GCTGRP428S**

'*mode (vdev)*' not attached RC=36

For more information on messages, see [z/VM: Other Components Messages and Codes](#).

For more information on GROUP, see:

- [“GROUP Panels” on page 103](#)
- [“Function Keys” on page 105](#)

## GROUP Panels

The GROUP command displays six panels:

- Primary Option Menu ([Figure 16 on page 103](#))
- Authorized VM User IDs ([Figure 17 on page 103](#))
- Saved System Information, Page 1 ([Figure 18 on page 104](#))
- Saved System Information, Page 2 ([Figure 19 on page 104](#))
- Automatic Saved Segment Links ([Figure 20 on page 105](#)).
- User IDs Requiring Reserved Storage for VSAM ([Figure 21 on page 105](#)).

GRP1	GCS GROUP - PRIMARY OPTION MENU										Primary	
<hr/>												
Fill in the blanks with the required information and then press the ENTER key.												
Type/change the name of the saved system that is being defined.												
SYSTEM NAME : GCS.....												
Type one number from the list below to display/update the:												
1. Authorized VM Userids. 2. Saved System Information. 3. Saved Segment Links. 4. VM Userids requiring reserved storage for VSAM.												
Type your choice here: _												
<hr/>												
PF: 1	HELP	2	CLEAR	3	END	4	...	5	...	6	...	
PF: 7	...	8	...	9	...	10	...	11	...	12	CURSOR	
>>>												

*Figure 16. GROUP Primary Option Menu Panel*

GRP11	AUTHORIZED VM USERIDS FOR < >										PAGE 1 OF 1	
<hr/>												
To ADD, fill in the blanks with the authorized VM userids. To CHANGE, type a new userid over the userid to be changed. To DELETE, type blanks on the line. To SAVE, press the ENTER key or PF6.												
.....												
.....												
.....												
.....												
.....												
<hr/>												
PF: 1	HELP	2	MORE	3	RETURN	4	...	5	REFRESH	6	SAVE	
PF: 7	PREVIOUS	8	NEXT	9	VERIFY	10	...	11	...	12	CURSOR	
>>>												

*Figure 17. GROUP Authorized VM User IDs Panel*

GRP121 SAVED SYSTEM INFORMATION FOR < > PAGE 1 OF 2

---

To ADD, fill in the blanks with the information.  
To CHANGE, type the information over the displayed value.  
To DELETE, type blanks on the line.  
To SAVE, press the ENTER key or PF6.

RECOVERY MACHINE USERID (required) . . . . : .....  
USERID to RECEIVE STORAGE DUMPS. . . . . : .....  
GCS TRACE TABLE SIZE (minimum 4K). . . . . : \_\_\_\_\_16 K  
Common storage above 16M line (YES or NO).: YES.  
Single user environment (YES or NO). . . . : NO.

Saved system information is continued on the next screen.

---

PF: 1	HELP	2	CLEAR	3	RETURN	4	...	5	REFRESH	6	SAVE
PF: 7	...	8	NEXT	9	VERIFY	10	...	11	...	12	CURSOR

====>

*Figure 18. Saved System Information Panel, Page 1*

**Note:** If you want to change the start and end addresses of common storage above the 16MB line, refer to “[Changing GCS Nucleus Options](#)” on page 507.

GRP122 SAVED SYSTEM INFORMATION FOR < > PAGE 2 OF 2

---

To ADD, fill in the blanks with the information.  
To CHANGE, type the information over the displayed value.  
To DELETE, type blanks on the line.  
To SAVE, press the ENTER key or PF6.

MAXIMUM NUMBER of VIRTUAL MACHINES (required) . . . : ---

SYSTEM ID (maximum 130 characters) . . . . . : -----

---

NAME of the VSAM SEGMENT . . . . . : CMSVSAM.  
NAME of the BAM SEGMENT . . . . . : CMSBAM..  
GCS saved system is restricted (YES or NO) . . . : YES  
TRACE TABLE in private storage (YES or NO) . . . : YES

---

PF: 1 HELP 2 CLEAR 3 RETURN 4 . . . 5 REFRESH 6 SAVE  
PF: 7 PREVIOUS 8 . . . 9 . . . 10 . . . 11 . . . 12 CURSOR

---

====>

*Figure 19. GROUP Saved System Information Panel, Page 2*

**Note:** If the names of the VSAM and BAM segments are CMSVSAM and CMSBAM, respectively, no changes are required. For installation of the CMSVSAM and CMSBAM segments, refer to the VSE/VSAM for VM Program Directory.

GRP13 AUTOMATIC SAVED SEGMENT LINKS FOR < > PAGE 1 OF 1

To ADD, fill in the blanks with the saved segment names that will be linked automatically during initialization of this virtual machine group.  
 To CHANGE, type a new saved segment name over the saved segment name to be changed.  
 To DELETE, type blanks on the line.  
 To SAVE, press the ENTER key or PF6.

.....  
 .....  
 .....  
 .....

PF: 1 HELP 2 MORE 3 RETURN 4 ... 5 REFRESH 6 SAVE  
 PF: 7 PREVIOUS 8 NEXT 9 VERIFY 10 VEROVER 11 ... 12 CURSOR

====>

Figure 20. GROUP Automatic Saved Segment Links Panel

GRP14 USERIDS REQUIRING RESERVED STORAGE FOR VSAM PAGE 1 OF 1

To ADD, fill in the blanks with the VM userids for VSAM.  
 To CHANGE, type a new userid over the userid to be changed.  
 To DELETE, type blanks on the line.  
 To SAVE, press the ENTER key or PF6.

.....  
 .....  
 .....  
 .....

PF: 1 HELP 2 MORE 3 RETURN 4 ... 5 REFRESH 6 SAVE  
 PF: 7 PREVIOUS 8 NEXT 9 VERIFY 10 ... 11 ... 12 CURSOR

====>

Figure 21. GROUP User IDs Requiring Reserved Storage for VSAM Panel

## Function Keys

The following table shows the function keys used with the GROUP panels.

Table 10. Function Keys Used with the GROUP Panels

KEY	FUNCTION
PF1 HELP	Shows information about the panel you are looking at.
PF2 CLEAR/ MORE	<p>Clears the input areas where you enter information.</p> <ul style="list-style-type: none"> <li>• On panels where multiple values are accepted, pressing PF2 gives you an additional screen that lets you enter more values.</li> <li>• Information from the previous panel does not need to be saved until you are ready to exit that panel's function.</li> </ul>

Table 10. Function Keys Used with the GROUP Panels (continued)

KEY	FUNCTION
<b>PF3</b> END/ RETURN	Leaves the present panel and returns you to a previous one. <ul style="list-style-type: none"> <li>• If you press <b>PF3</b> on the Primary Option Menu, you return to CMS.</li> <li>• If you press <b>PF3</b> on any other screen, you return to the Primary Option Menu.</li> </ul>
<b>PF5</b> REFRESH	Fills in the panel's input areas with the values you last saved there.
<b>PF6</b> SAVE	Saves information that you entered on the panel or panels (for the configuration file <i>systemname</i> GROUP).
<b>PF7</b> PREVIOUS	Returns to the previous panel, if there is one.
<b>PF8</b> NEXT	Moves ahead to the next panel, if there is one.
<b>PF9</b> VERIFY	This PF key validates the user IDs or the saved segments entered on the panel.
<b>PF10</b> VEROVER	Checks to see if segments you have entered on the panel overlap each other.
<b>PF12</b> CURSOR	Moves the cursor to the panel's command line.
<b>PF4</b> , <b>PF11</b>	Not Used
<b>ENTER</b> <b>PROCESS</b>	Saves information that you entered and processes any valid CP or CMS command typed on the command line. A specific command you can enter is CANCEL, entered on any panel, to return you to CMS.

# HX

---

## Format

```
►► HX ►►
```

## Purpose

Use the HX command to halt execution of all programs and commands active in a virtual machine.

Sometimes you may want to halt the processing of a command or program after you have already issued it. Use the HX command to halt processing of all commands and programs active in a virtual machine. Issuing HX will also clear commands you have stacked and waiting to be processed, including any of your own commands defined with a LOADCMD command.

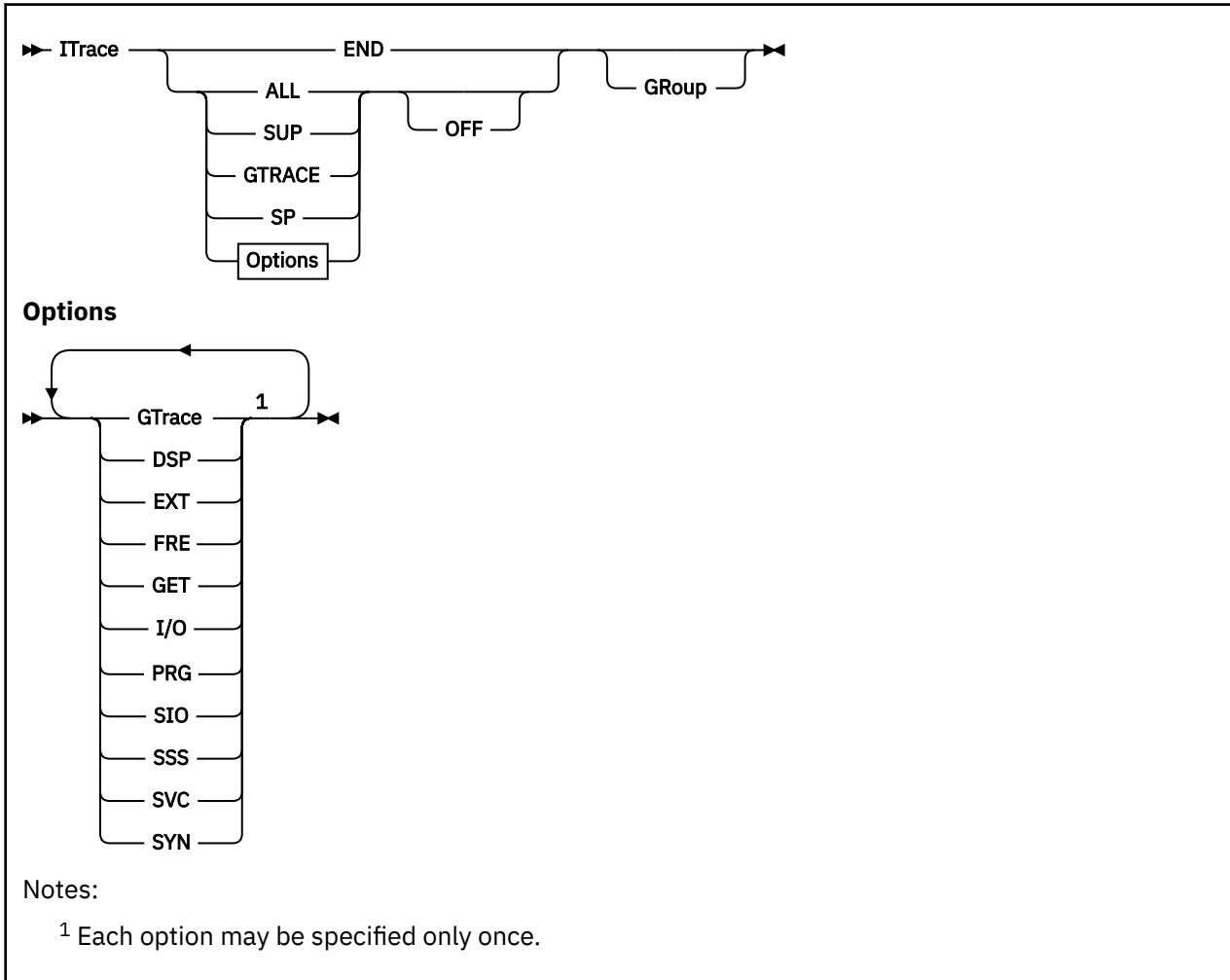
## Messages

- **GCTABD225I** Hx complete

For more information on messages, see [z/VM: Other Components Messages and Codes](#).

# ITRACE

## Format



## Purpose

The ITRACE command is provided to enable you to perform GCS problem diagnosis.

Use the ITRACE command to enable or disable recording of internal trace events for a virtual machine or an entire group.

GCS maintains an internal trace table that contains records of:

- Supervisor events
- GTRACE events.

GCS records all supervisor events that occur within your virtual machine. It also can record data from programs or applications (GTRACE events) within your virtual machine. This latter information is gathered through the GTRACE macro, and you can use it for debugging purposes.

The ITRACE command lets you control what goes in the internal trace table. Internal tracing for supervisor events starts out active, or *enabled*, recording all events that occur from the time you join the group. However, if you want GCS to begin recording GTRACE events or SP service points, you have to enable the tracing yourself. You can enter the ITRACE command to turn off tracing of supervisor events

and later to turn it back on. You can turn GTRACE event tracing on and later turn it off. If you have an authorized virtual machine, you can control internal tracing for the entire group.

## Operands

### **GTrace**

Indicates that you want to affect only the internal tracing of data passed through the GTRACE macro.

### **DSP**

Indicates that you want to affect only the internal tracing of each task switch (dispatch of a different task).

### **EXT**

Indicates that you want to affect only the internal tracing of each external interrupt.

### **FRE**

Indicates that you want to affect only the internal tracing of FREEMAIN events invoked through SVC and Branch Entry calls.

### **GET**

Indicates that you want to affect only the internal tracing of GETMAIN events invoked through SVC and Branch Entry calls.

### **I/O**

Indicates that you want to affect only the internal tracing of each I/O interrupt.

### **PRG**

Indicates that you want to affect only the internal tracing of each program interrupt.

### **SIO**

Indicates that you want to affect only the internal tracing of each request by the GCS supervisor of the SIO instruction.

### **SP**

Indicates that you want to affect only the internal tracing of Service Points. Service Points include the branch entries to WAIT, POST, SCHEDEX, VALIDATE, and IUCVCOM. SP cannot be specified for GROUP. SP trace points will not be traced during the processing of program interrupts.

### **SSS**

Indicates that you want to affect only the internal tracing of IUCV interrupts on the Signal System Service path.

### **SUP**

Indicates that you want to affect only the internal tracing of GCS supervisor events. It includes DSP, EXT, FRE, GET, I/O, PRG, SIO, SSS, and SVC events.

### **SVC**

Indicates that you want to affect only the internal tracing of each SVC interrupt.

### **SYN**

Indicates that you want to affect only the internal tracing of APPC/VM synchronous events.

### **OFF**

Halts internal tracing of the events you indicated. ON is assumed, unless you specify OFF.

### **END**

Terminates, or disables, all internal tracing. You must specify this option by itself or with the GROUP operand. These are the only two ways you can use it.

### **ALL**

Indicates that you want to apply this command to the internal tracing of both GTRACE and supervisor events.

### **GGroup**

Indicates that you want this command to apply to the entire virtual machine group rather than just to your issuing machine. It will also apply to machines that join the group later.

To use this operand, you need to have an authorized user ID. Commands you enter with the GROUP option take precedence over commands issued without. However, an authorized virtual machine can disable tracing for itself although another authorized virtual machine started internal tracing for the entire group.

## Examples

```
ITRACE GTRACE
```

Enables tracing of GTRACE events (program or application data) in the virtual machine that issued this command.

```
ITRACE GTRACE GROUP
```

Enables tracing of GTRACE events for the virtual machine group. The virtual machine issuing this command must be authorized.

```
ITRACE GTRACE OFF GROUP
```

Disables tracing of GTRACE events for the virtual machine group. The virtual machine issuing this command must be authorized.

```
ITRACE SUP OFF
```

Disables internal tracing of supervisor events for the virtual machine issuing this command.

```
ITRACE END
```

Disables internal tracing of all events for the virtual machine issuing this command. If tracing had been enabled for the entire group, you would need an authorized virtual machine to issue this for yourself. If the tracing was enabled just for your virtual machine, you do not have to be authorized to issue this for yourself.

## Messages

- **GCTYTG001E** Invalid option '*option*' RC=4
- **GCTYTG009E** Operand missing or invalid
- **GCTYTG517I** ITRACE set ON for *event-types*
- **GCTYTG518I** ITRACE set ON for *event-types* for GROUP
- **GCTYTG519I** ITRACE set OFF for *event-types*
- **GCTYTG520I** ITRACE set OFF for *event-types* for GROUP
- **GCTYTG521E** ITRACE GROUP option is in effect for *event-types* RC=8

For more information on messages, see [z/VM: Other Components Messages and Codes](#).

## Return Codes

The meanings of return codes for these messages are:

Hex Code	Decimal Code	Meaning
X'00'	0	The tracing of events has been successfully enabled or disabled.
X'04'	4	An invalid operand was specified, or an unauthorized user specified the GROUP operand. Your request was ignored.

Hex Code	Decimal Code	Meaning
X'08'	8	An authorized virtual machine had enabled tracing of user events using the GROUP operand. An unauthorized virtual machine then attempted to disable this tracing. The request was ignored.

## LOADCMD

---

### Format

```
►► LOADCmd — name member ►►
```

### Purpose

Use the LOADCMD command to define a program module to be executed as a command.

LOADCMD is a feature that lets you define your own commands; it lets you assign a command name to a program module. (The module for this program must reside in a CMS load library that you have defined with a GLOBAL command.) When you enter the command name, this module gets control and executes. It remains in storage, waiting to be run again when you enter its assigned name from either the console or the CMDSI macro or a command file (EXEC).

For example, to run the GCS application ACF/VTAM, you first have to define the VTAM command. VTAM is a command name that will be processed by one of ACF/VTAM's program modules. After you have defined and issued this VTAM command name, you can enter any of the following ACF/VTAM commands:

- START
- HALT
- VARY
- MODIFY
- DISPLAY

### Operands

#### *name*

The name of the command you are defining.

#### *member*

The member of a CMS load library associated with the command you have defined. This member is the module that executes the command you have defined; it is loaded into private, free storage.

When you enter the *name*, GCS calls the *member* to run the command. Here's what your registers will contain:

Register	Contents
0	Address of an extended parameter list.
1	Address of a tokenized parameter list of consecutive doublewords. The first item in the list is the name of your called routine or program. Other items in the list may contain arguments you want passed to it.
3	Address of a word (UWORD) in storage that is available for the command's use.
12	Address of the entry point to your program. You can use this address as a base address to establish immediate addressability in your program.
13	Address of a 96-byte save area for your program's use.
14	Return address of the SVC handling routines. The program returns control to this address after it finishes executing.

Register	Contents
15	<p>Same as Register 12, except that you should not use this one as a base register. The SVCs use it to communicate with the program, and GCS uses it to return a completion code. Any time that completion code is nonzero, you will see it in the ready message (if you entered the command at the console):</p> <pre style="background-color: #f0f0f0; padding: 5px;">Ready(nnnnn);</pre> <p>If the program you run does not return a completion code in Register 15, make sure it puts a zero there before transferring control. Otherwise, your ready message may contain meaningless data (whatever was in Register 15 at the time).</p>

When you enter a command, a GCS scan routine sets up two distinct parameter lists:

- The first list is a tokenized parameter list. (Register 1 contains its address.) The parameters listed there line up on consecutive doubleword boundaries. Blanks and parentheses serve as delimiters separating each parameter. (Parentheses show up in the list, each on a doubleword boundary.)
- The second is an extended, or *not tokenized*, parameter list. (Register 0 contains its address.) It contains addresses that map out the extended form of a command. This extended parameter list has the following format:

```

EPLMAP DC A(CMDBEG) ADDR OF COMMAND TOKEN
        DC A(ARGBEG) ADDR OF BEGINNING OF ARGUMENTS
        DC A(ARGEND) ADDR OF END OF ARGUMENTS
        DC A(0) ADDR OF EXEC FILEBLOCK
        DC A(0) ADDR OF FUNCTION ARGUMENT LIST
        DC A(0) ADDR FOR RETURN OF FUNCTION DATA
        DS X INDICATOR (see the following note)
        DS 3X RESERVED
    
```

**Note:** An INDICATOR byte of X'00' is a sign that a program issued the command. X'0B' is a sign that it was issued from the console. X'01' is a call from REXX when ADDRESS COMMAND is specified. X'05' is used by REXX for function calls.

Here are two ways you might enter a command and two sets of accompanying tokenized and extended parameter lists that result:

1. You enter:

```
=====> loadcmd cmdname memname
```

The scan routine sets up the following tokenized parameter list:

```

FORMAT: DC CL8 'LOADCMD'
        DC CL8 'CMDNAME'
        DC CL8 'MEMNAME'
        DC 8X 'FF'
    
```

The scan routine sets up the extended parameter list with the following references:

```

CMDBEG  DC  C 'loadcmd'
ARGBEG   DC  C 'cmdname memname'
ARGEND   EQU *
    
```

The first nonblank character following ‘loadcmd’ determines the start of ARGBEG.

**Note:** The tokenized parameter list is passed with uppercase characters. The extended parameter list is passed with mixed case (as entered) characters.

2. You enter ‘loadcmd’ without specifying any arguments:

```
=====> loadcmd
```

The scan routine sets up the following tokenized parameter list:

```
FORMAT: DC CL8 'LOADCMD'
        DC 8X 'FF'
```

The scan routine sets up the extended parameter list with the following references:

```
CMDBEG  DC  C'loadcmd'
ARGBEG   EQU  *
ARGEND   EQU  *
```

With no arguments specified, ARGBEG is set equal to ARGEND.

**Note:** The tokenized parameter list is passed with uppercase characters. The extended parameter list is passed with mixed case (as entered) characters.

For more information on parameter lists, see [z/VM: CMS User's Guide](#).

## Usage

If the program defined by the LOADCMD is reentrant, then it is loaded into key 0 storage. This ensures that it is not accidentally modified or tampered with.

## Examples

```
LOADCMD MYCMD MYMOD
```

Defines the command named MYCMD to GCS. The module containing the code for this command can be found in a CMS load library under the member name of MYMOD. Then, by issuing:

```
MYCMD
```

The module named MYMOD is invoked.

## Messages

- **GCTLDC212E** Member cannot be loaded. Command is not defined RC=xx
- **GCTLDC240I** No entry points were loaded by the LOADCMD command

For more information on messages, see [z/VM: Other Components Messages and Codes](#).

## Return Codes

The meanings of return codes for the GCTLDC212E message are:

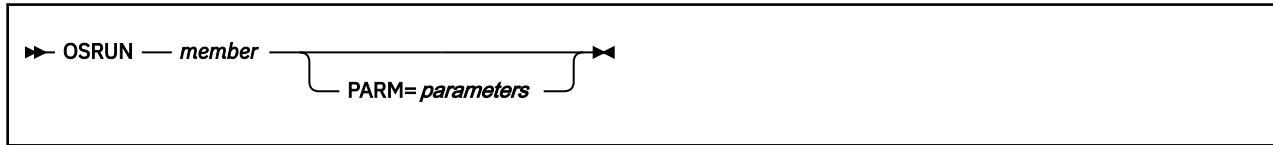
Hex Code	Decimal Code	Meaning
X'01'	1	The command has already been defined.
X'04'	4	The module is marked not executable. The module is not loaded and the command is not defined. The module is not suitable for use as a command module. Consult the information provided by the linkage editor, at the time the module was created, to determine why the module is not executable.
X'0A'	10	The module is an overlay structure. The module is not loaded and the command is not defined. If this module is to be used as a command module, it must be redefined so that it does not require overlays.

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'0C'	12	The module is marked only loadable. The module is not loaded and the command is not defined. This module is not suitable for use as a command module.
X'OE'	14	The command name specified is a GCS immediate command or an abbreviation for one.
X'18'	24	Too many operands were specified.
X'1C'	28	The specified member cannot be found.
X'20'	32	No member name was specified.
X'24'	36	A permanent I/O error was found when the system attempted to search the CMS LOADLIB directory.
X'28'	40	Insufficient virtual storage was available to read the directory entry for this module.
X'29'	41	Insufficient free storage was available to build the nucleus extension control blocks representing this command.

# OSRUN

---

## Format



## Purpose

Use the OSRUN command to start a GCS application program.

The application program must either be a member of a CMS load library (defined with the GLOBAL command) or reside in a saved segment. The OSRUN command maintains control until the program ends; therefore, cannot be executed other commands while the program is running.

## Operands

### *member*

The member of the CMS load library you want to process.

### **PARM=parameters**

The OS parameters that you want to pass to the module. If these parameters contain blanks or special characters, they must be enclosed them in quotation marks. To include a quotation mark in a parameter, enter two quotation marks side-by-side (' '). Parameters may be no longer than 100 characters.

The parameters are passed to the module in OS format: Register 1 points to a fullword containing the address of the character string. (The first halfword field contains the length of the character string.)

## Messages

- **GCTLOS220E** Unable to open file '*filename*'
- **GCTLOS223S** Error '*nn*' reading file '*fn [ft fm]*' from disk
- **GCTLOS224E** Member '*membername*' not found in library
- **GCTOSR006E** Invalid parameter '*parameter*' RC=24
- **GCTOSR022E** No filename specified RC=24
- **GCTOSR219E** Parm field contains more than 100 characters RC=24
- **GCTOSR236E** Ending apostrophe is missing RC=24
- **GCTABD237E** Command ended without detaching subtasks

For more information on messages, see [z/VM: Other Components Messages and Codes](#).

# QUERY

---

## Purpose

Use the QUERY command to request information about your GCS virtual machine.

## Operands

*Table 11. Query commands.* This table lists operands that can be issued with the QUERY command and the location of more information for each:

Command	Location
ADDRESS	See page “ <a href="#">QUERY ADDRESS</a> ” on page 119
AUTHUSER	See page “ <a href="#">QUERY AUTHUSER</a> ” on page 120
COMMON	See page “ <a href="#">QUERY COMMON</a> ” on page 121
DISK	See page “ <a href="#">QUERY DISK</a> ” on page 122
DLBL	See page “ <a href="#">QUERY DLBL</a> ” on page 124
DUMP	See page “ <a href="#">QUERY DUMP</a> ” on page 126
DUMLOCK	See page “ <a href="#">QUERY DUMLOCK</a> ” on page 127
DUMPVM	See page “ <a href="#">QUERY DUMPVM</a> ” on page 128
ETRACE	See page “ <a href="#">QUERY ETRACE</a> ” on page 129
FILEDEF	See page “ <a href="#">QUERY FILEDEF</a> ” on page 130
GCSLEVEL	See page “ <a href="#">QUERY GCSLEVEL</a> ” on page 131
GROUP	See page “ <a href="#">QUERY GROUP</a> ” on page 132
IPOLL	See page “ <a href="#">QUERY IPOLL</a> ” on page 133
ITRACE	See page “ <a href="#">QUERY ITRACE</a> ” on page 134
LOADALL	See page “ <a href="#">QUERY LOADALL</a> ” on page 135
LOADCMD	See page “ <a href="#">QUERY LOADCMD</a> ” on page 136
LOADLIB	See page “ <a href="#">QUERY LOADLIB</a> ” on page 137
LOCK	See page “ <a href="#">QUERY LOCK</a> ” on page 138
MODDATE	See page “ <a href="#">QUERY MODDATE</a> ” on page 139
REPLY	See page “ <a href="#">QUERY REPLY</a> ” on page 140
REXXSTOR	See page “ <a href="#">QUERY REXXSTOR</a> ” on page 141
SEARCH	See page “ <a href="#">QUERY SEARCH</a> ” on page 142
SYSNAMES	See page “ <a href="#">QUERY SYSNAMES</a> ” on page 143
TRACETAB	See page “ <a href="#">QUERY TRACETAB</a> ” on page 144
TSLICE	See page “ <a href="#">QUERY TSLICE</a> ” on page 145

## Messages

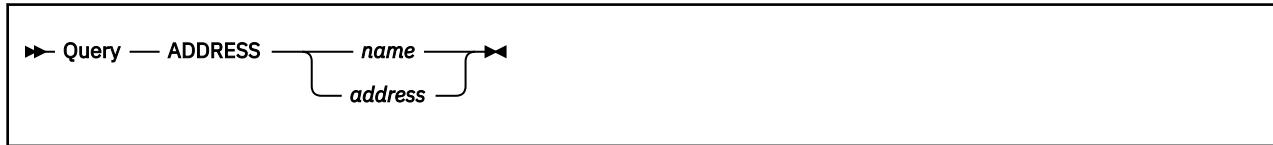
All QUERY messages except GCTQRL032T are issued without message numbers.

- **GCTQAD365I** Address of *variable* is *address*
- **GCTQRD239I** No entry points are currently loaded in this virtual machine
- **GCTQRD240I** No entry points were loaded by the LOADCMD command
- **GCTQRL032T** Supervisor error 5. Re-IPL *sysname*
- **GCTQRL217E** The common lock is free
- **GCTQRL218I** The common lock is held by *userid*
- **GCTQRQ514I** All external trace events are disabled
- **GCTQRQ515I** External trace is enabled for *event-types*
- **GCTQRQ516I** External trace is enabled for *event-types* for GROUP
- **GCTQRQ522I** Internal trace is enabled for *event-types*
- **GCTQRQ523I** Internal trace is enabled for *event-types* for GROUP
- **GCTQRQ524I** All internal trace events are disabled
- **GCTQRR005S** All internal trace events are disabled
- **GCTQRR009I** Operand is missing or invalid
  - Name is not a GCS module or table.
  - Address is not within a GCS module or table.
- **GCTQRR214I** No replies outstanding
- **GCTQRR215I** The following replies are outstanding:
  - **GCTQRR216I** GROUPID=*systemname*, USERS: CURRENT=nnnnn, MAXIMUM=mmmmm
  - **GCTQRR244I** *userid* is now the virtual machine receiving dumps
  - **GCTQRR245I** *userid* can now IPL as an authorized virtual machine
  - **GCTQRR247I** The trace table is now being maintained in *location* storage
  - **GCTQRR248I** No users are currently authorized
  - **GCTQRR364I** IPOLL = *setting*
  - **GCTQRR366I** Address is *name* + X'nnnnnnnn'
  - **GCTQRR367I** Date of *name* is mm/dd/yy
  - **GCTQRR368I** Date of *name* is not available
  - **GCTQRR370I** REXXSTOR = *nn*
  - **GCTQRR900I** z/VM Version *n* Release *n*, Service level *n*
  - **GCTQRS015E** '*parameter*' is invalid for '*function*' function RC=24
  - **GCTQRS017E** Disk {*mode/vdev/volumeid*} not accessed
  - **GCTQRS019E** No Read/Write *mode* disk accessed RC=1
  - **GCTQRS020E** No Read/Write disk with space available accessed RC=2
  - **GCTQRU303I** No user defined FILEDEFs in effect
  - **GCTQRX005S** Virtual storage capacity exceeded RC=8
  - **GCTQRX006E** Invalid parameter '*parameter*' RC=24
  - **GCTQRX303I** No user defined DLBLs in effect
  - **GCTQRY005S** Virtual storage capacity exceeded RC=8
  - **GCTQRY006E** Invalid parameter '*parameter*' RC=24
  - **GCTQRY013E** No function specified RC=24

For more information on messages, see [z/VM: Other Components Messages and Codes](#).

# QUERY ADDRESS

## Format



## Purpose

Use the QUERY ADDRESS command to display the address of a module, table, or control block, or the name of a GCS module, table, or control block, at a specified address. See the *z/VM: Diagnosis Guide* for major control block names.

## Operands

### **ADDRESS**

specifies that you wish to know an address in GCS, or the name of a GCS module or table at a specified address.

### **name**

is the name of a GCS module or table. For a list of names that can be used see Appendix C.

### **address**

is an address within the GCS supervisor or a GCS table.

## Response

Address of *name* is *address*

or

Address is *name* + *displacement*

### **Where:**

#### **name**

is the name of a GCS module or table.

#### **address**

is the address location of the requested module or table.

#### **displacement**

is the displacement within the GCS module or table.

## Messages

For messages that apply to the QUERY command, see [“QUERY” on page 117](#).

## **QUERY AUTHUSER**

---

### **Format**

```
►► Query — AUTHUSER ►►
```

### **Purpose**

Use the QUERY AUTHUSER command to display the list of authorized users.

### **Operands**

#### **AUTHUSER**

Displays the list of authorized users.

### **Response**

```
userid can now IPL as an authorized virtual machine.
```

```
or
```

```
No users are currently authorized.
```

#### **Where:**

#### **userid**

is the user ID authorized to IPL the Virtual Machine.

### **Messages**

For messages that apply to the QUERY command, see [“QUERY” on page 117](#).

# QUERY COMMON

## Format

```
►► Query — COMMON ►►
```

## Purpose

Use the QUERY COMMON command to display the available common storage above and below the 16MB line.

## Operands

### COMMON

Displays the available common storage above and below the 16MB line.

## Response

```
sysname available common storage is nnnnn KB BELOW and nnnnn KB ABOVE the 16MB line
```

### Where:

#### sysname

is the name that identifies the GCS saved system.

#### nnnnn

is the available storage in Kilobyte (KB) above or below the 16MB line.

## Messages

For messages that apply to the QUERY command, see “[QUERY](#)” on page 117.

## QUERY DISK

### Format



### Purpose

Use the QUERY DISK command to display the status of a disk.

### Operands

#### **DISK**

Displays disk information.

#### **\***

Displays one line of information for each disk that is accessed. If no DISK option is specified, this is the default.

#### **fm**

Displays information about the single disk associated with that file mode.

#### **R/W**

Displays information for each accessed disk in read/write mode.

#### **MAX**

Displays information for the R/W disk having the most available space.

### Response

```
LABEL VDEV M STAT CYL TYPE BLKSIZE FILES BLKS USED-(%) BLKS LEFT BLK TOTAL  
label vdev m stat cyl type blksize files blks_used     blks_left blk_total
```

#### **Where:**

##### **label**

The label assigned to the disk when it was formatted. If an OS or DOS disk, this is the volume label.

##### **vdev**

The virtual device number.

##### **m**

The access mode letter.

##### **stat**

Indicates whether the disk is read/only (R/O) or read/write (R/W).

##### **cyl**

The number of cylinders available on the disk. For an FB-512 device, this field contains the abbreviation FB rather than the number of cylinders.

##### **type**

The device type of the disk.

**blksize**

The number of units that make up a block on the disk.

**files**

The number of files on the disk. If you have an OS or DOS disk, this field will contain either OS or DOS.

**blks\_used**

The number of disk blocks in use.

**blks\_left**

The number of disk blocks left. (The actual number of disk blocks remaining is lower because this number also counts control blocks.)

**blk\_total**

The total number of blocks.

## Messages

For messages that apply to the QUERY command, see [“QUERY” on page 117](#).

## QUERY DLBL

### Format



### Purpose

Use the QUERY DLBL command to display the contents of the current data set definitions.

### Operands

#### DLBL

Querying DLBL yields information that is explained in the response section following.

#### MULT

Indicates that you want to enter volume specifications that refer to an existing multivolume VSAM data set. For more information on the requirements for VSE/VSAM, see the *VSE/VSAM User's Guide*.

### Response

DDNAME	MODE	TYPE	CATALOG	VOL	BUFSPC	PERM	DISK	DATASET.NAME
xxxxxx	nn	xxxx	xxxxxxx			xxx	xxx	xxxxxxx
:	:	:	:	:	:	:	:	:
:	:	:	:	:	:	:	:	:

Where:

#### DDNAME

The program ddname.

#### MODE

The disk on which the data set resides.

#### TYPE

The type of data set defined. This field will always be VSAM.

#### CATALOG

The ddname of the VSAM catalog you want searched for the specified data set.

#### VOL

The number of volumes (if greater than one) on which VSAM resides. This field will be blank if the VSAM data set resides on only one volume. The actual volumes may be displayed by entering either DLBL (MULT) or the QUERY DLBL MULT commands.

#### BUFSPC

The size of the VSAM buffer space, if entered at DLBL definition time.

#### PERM

Indicates whether the DLBL definition was made with the PERM option. This field will contain YES or NO.

#### DISK

Indicates whether the data set resided on a CMS or DOS/OS disk at DLBL definition time. The values for this field are DOS and CMS.

**DATASET.NAME**

For a data set residing on a CMS disk, the CMS file name and file type are given; for a data set residing on a DOS/OS disk, the data set name (maximum 44 characters) is given. This field will be blank if you failed to enter a DOS/OS data set name at DLBL definition time.

If no DLBL definitions are active, you will get the following message:

No user defined DLBL's in effect

**Messages**

For messages that apply to the QUERY command, see [“QUERY” on page 117](#).

## **QUERY DUMP**

---

### **Format**

```
►► Query — DUMP ►►
```

### **Purpose**

Use the QUERY DUMP command to display the status of the dump facility.

### **Operands**

#### **DUMP**

Describes the state of the dump facility.

### **Response**

option

#### **ON**

A dump will always be taken.

#### **OFF**

No dumps will be taken.

#### **DEFAULT**

A dump will be taken only if there is a system error or if the DUMP option was specified on the ABEND macro.

### **Messages**

For messages that apply to the QUERY command, see [“QUERY” on page 117](#).

# QUERY DUMPLOCK

## Format

```
►► Query — DUMPLOCK ►►
```

## Purpose

Use the QUERY DUMPLOCK command to display the status of the dumplock facility.

## Operands

### DUMPLOCK

Describes the state of the dumplock.

## Response

option

### ON

The common storage lock will be held while common storage is being dumped.

### OFF

The common storage lock will not be held while common storage is being dumped.

## Messages

For messages that apply to the QUERY command, see “[QUERY](#)” on page 117.

## **QUERY DUMPVM**

---

### **Format**

```
►► Query — DUMPVM ►►
```

### **Purpose**

Use the QUERY DUMPVM command to display the name of the virtual machine to receive dumps.

### **Operands**

#### **DUMPVM**

Displays the name of the virtual machine to receive dumps.

### **Response**

```
userid is now the virtual machine receiving dumps
```

#### **userid**

is the name of the virtual machine.

### **Messages**

For messages that apply to the QUERY command, see “[QUERY](#)” on page 117.

# QUERY ETRACE

## Format

```
►► Query — ETRACE ►►
```

## Purpose

Use the QUERY ETRACE command to display the list of the events that are enabled for external tracing.

## Operands

### ETRACE

Displays a list of the events that are enabled for external tracing (recording in a spool file).

## Response

```
All external trace events are disabled
External trace is enabled for event-types
External trace is enabled for event-types for GROUP
```

### *event-types*

are the various events that are enabled for tracing.

## Messages

For messages that apply to the QUERY command, see “[QUERY](#)” on page 117.

## **QUERY FILEDEF**

---

### **Format**

```
►- Query — FILEDEF -►
```

### **Purpose**

Use the QUERY FILEDEF command to display all file definitions in effect.

### **Operands**

#### **FILEDEF**

Displays all file definitions in effect.

### **Response**

```
ddname device fn ft
. . .
. . .
. . .
```

If you have no file definitions in effect, you will receive the following message:

```
No user defined FILEDEFS in effect
```

### **Messages**

For messages that apply to the QUERY command, see “[QUERY](#)” on page 117.

# QUERY GCSLEVEL

## Format

```
► Query — GCSLEVEL ►
```

## Purpose

Use the QUERY GCSLEVEL command to display the release and service level of the GCS component of z/VM.

## Operands

### GCSLEVEL

Displays the release and service level of the GCS component of z/VM.

## Response

```
I z/VM Version n Release n.n, Service Level ffxx
```

The service level uses the format *ffxx*:

- *ff* indicates the latest feature pack number.
- *xx* indicates the fix pack number for the latest feature pack.

The fix pack number is reset to zero by each feature pack. Fix pack numbering restarts at 01 after each new feature pack. For example, the service level established by the RSU supplied with the z/VM 7.4 installation media is 0001, which indicates feature pack 0 and fix pack 1. As another example, a service level of 0403 would indicate that feature pack 4 and its third fix pack (fix pack 3) is installed.

## Messages

For messages that apply to the QUERY command, see [“QUERY” on page 117](#).

## **QUERY GROUP**

---

### **Format**

```
►► Query — GROUP ►►
```

### **Purpose**

Use the QUERY GROUP command to display the user IDs of the virtual machines in the GCS group.

### **Operands**

#### **GROUP**

Displays the user IDs of the virtual machines in the GCS group of the issuer.

### **Response**

```
GROUPID=groupname, USERS: CURRENT=ccccc, MAXIMUM=mmmmm  
VMUSERID(s)
```

### **Messages**

For messages that apply to the QUERY command, see [“QUERY” on page 117](#).

## QUERY IPOLL

### Format

```
►► Query — IPOLL ►►
```

### Purpose

Use the QUERY IPOLL command to display the ipoll setting for the virtual machine.

### Operands

#### IPOLL

Displays the status of the ipoll setting for the virtual machine.

### Response

```
IPOLL = setting
```

#### Where:

```
setting = ON or OFF
```

### Messages

For messages that apply to the QUERY command, see [“QUERY” on page 117](#).

## QUERY ITRACE

---

### Format

```
►► Query — ITRACE ►►
```

### Purpose

Use the QUERY ITRACE command to display the list of the events that are enabled for internal tracing.

### Operands

#### ITRACE

Displays a list of the events that are enabled for internal tracing.

### Response

```
Internal trace is enabled for event-type
Internal trace is enabled for event-type for GROUP
All internal trace events are disabled
```

#### *event-types*

are the various events that are enabled for tracing.

### Messages

For messages that apply to the QUERY command, see [“QUERY” on page 117](#).

# QUERY LOADALL

---

## Format

```
► Query — LOADALL ►
```

## Purpose

Use the QUERY LOADALL command to display the entry point names and addresses for the entry points that have been loaded and currently reside in the virtual machine storage.

## Operands

### LOADALL

Displays the entry point names and addresses for the entry points that have been loaded and currently reside in the virtual machine storage.

QUERY LOADALL does not display the names of modules loaded with the ADDR parameter of the LOAD macro.

## Response

ENTRY NAME	ENTRY ADDRESS	TYPE
nnnnnnnn	aaaaaaaa	t

### nnnnnnnn

Is the entry point name.

### aaaaaaaa

Is the entry point address.

### t

Is the entry point type.

*Where:*

### M

The entry point is a major entry point.

### A

The entry point is an alias entry point.

### I

The entry point is one defined by the IDENTIFY macro.

## Messages

For messages that apply to the QUERY command, see [“QUERY” on page 117](#).

## **QUERY LOADCMD**

---

### **Format**

```
► Query — LOADCMD ►
```

### **Purpose**

Use the QUERY LOADCMD command to locate the entry point addresses for all entry points that are loaded by the LOADCMD command.

### **Operands**

#### **LOADCMD**

Locates the entry point addresses for all entry points that are loaded by the LOADCMD command.

### **Response**

ENTRY NAME	COMMAND NAME	ENTRY ADDRESS
nnnnnnnn	cccccccc	aaaaaaaa

#### **nnnnnnnn**

Is the entry point name.

#### **cccccccc**

Is the command name.

#### **aaaaaaaa**

Is the entry point address.

### **Messages**

For messages that apply to the QUERY command, see [“QUERY” on page 117](#).

# QUERY LOADLIB

## Format

```
► Query — LOADLIB ►
```

## Purpose

Use the QUERY LOADLIB command to display the names of all files (of file type LOADLIB) that will be searched for load modules.

## Operands

### LOADLIB

Displays the names of all files (of file type LOADLIB) that will be searched for load modules. This gives you a list of all LOADLIBs specified on the last GLOBAL LOADLIB command, if any.

## Response

### *Response:*

```
LOADLIB=libname1 . . . libname8  
      .      .      .  
      :      :      :  
      .      .      .
```

up to eight names are displayed per line, for as many lines as necessary.

If no libraries are to be searched, the response is:

```
LOADLIB = NONE
```

## Messages

For messages that apply to the QUERY command, see [“QUERY” on page 117](#).

## **QUERY LOCK**

---

### **Format**

```
►► Query — LOCK ►►
```

### **Purpose**

Use the QUERY LOCK command to display the status of the common lock.

### **Operands**

#### **LOCK**

Displays the status of the common lock. If the lock is held, the userID holding the lock is displayed.

### **Response**

```
The common lock and the data space lock are free  
The common lock is held by 'userID'
```

### **Messages**

For messages that apply to the QUERY command, see [“QUERY” on page 117](#).

# QUERY MODDATE

## Format

```
► Query — MODDATE — name —►  
      |  
      LAST
```

## Purpose

Use the QUERY MODDATE command to display the compilation date of the specified module.

## Operands

### MODDATE

specifies that you wish to know the compilation date of the specified module.

### *name*

is the name of a GCS module. For a list of names that can be used see Appendix C.

### *LAST*

specifies that you want a list of all modules compiled on the most recent compilation date.

## Response

```
Date of name is date  
Date of name is not available.
```

### Where:

### *name*

is the name of a GCS module specified in the command.

### *date*

is the last compilation date of the module in mm/dd/yyyy format.

## Messages

For messages that apply to the QUERY command, see [“QUERY” on page 117](#).

## **QUERY REPLY**

---

### **Format**

```
►- Query — REPLY -►
```

### **Purpose**

Use the QUERY REPLY command to display the text and the identification number of all messages waiting for a reply.

### **Operands**

#### **REPLY**

Displays the text and the identification number of all messages waiting for a reply.

### **Response**

```
The following replies are outstanding:  
xx  yyyyyyyy  
xx  yyyyyyyy
```

If no messages are waiting for a reply, the response is:

```
No replies outstanding
```

### **Messages**

For messages that apply to the QUERY command, see “[QUERY](#)” on page 117.

# QUERY REXXSTOR

## Format

```
► Query — REXXSTOR ►
```

## Purpose

Use the QUERY REXXSTOR command to display the REXXSTOR setting.

## Operands

### REXXSTOR

Displays the REXXSTOR setting.

## Response

```
REXXSTOR = 24  
REXXSTOR = 31
```

## Messages

For messages that apply to the QUERY command, see [“QUERY” on page 117](#).

## QUERY SEARCH

---

### Format

```
►- Query — SEARCH -►
```

### Purpose

Use the QUERY SEARCH command to display the search order of your accessed disks.

### Operands

#### SEARCH

Displays the search order of your accessed disks.

### Response

label	vdev	mode	R/O	OS
			R/W	DOS
:	:	:	:	:
:	:	:	:	:
:	.	.	.	.

#### label

The label assigned to the disk when it was formatted. If an OS or DOS disk, this is the volume label.

#### vdev

The virtual device number.

#### m

The file mode letter assigned to the disk when it was accessed.

#### R/O or R/W

Indicates whether the disk is read/only or read/write.

#### OS or DOS

Indicates an OS or DOS disk.

### Messages

For messages that apply to the QUERY command, see [“QUERY” on page 117](#).

# QUERY SYSNAMES

## Format

```
►► Query — SYSNAMES ►►
```

## Purpose

Use the QUERY SYSNAMES command to display the names of the standard saved systems.

## Operands

### SYSNAMES

Displays the names of the standard saved systems.

### Response

```
SYSNAMES:  GCSVSAM   GCSBAM  
ENTRIES:  entry...  entry...
```

where:

### SYSNAMES

The names that identify the saved systems (discontiguous shared segments).

### ENTRIES

The default system names or the system names established through the SET command.

## Messages

For messages that apply to the QUERY command, see [“QUERY” on page 117](#).

## **QUERY TRACETAB**

---

### **Format**

```
►► Query — TRACETAB ►►
```

### **Purpose**

Use the QUERY TRACETAB command to display the location of the internal trace table.

### **Operands**

#### **TRACETAB**

Displays the location of the internal trace table.

### **Response**

The trace table is now being maintained in *location* storage.

### **Messages**

For messages that apply to the QUERY command, see [“QUERY” on page 117](#).

# QUERY TSLICE

## Format

```
► Query — TSLICE ►
```

## Purpose

Use the QUERY TSLICE command to obtain the dispatching time slice value.

## Operands

### TSLICE

Displays a positive integer that represents the number of milliseconds that a single task will be allowed to be dispatched before the GCS dispatcher dispatches another task.

## Response

Time slice	Tasks ready to run	Time slice X tasks ready
----- n milliseconds	n	----- n milliseconds

### Note:

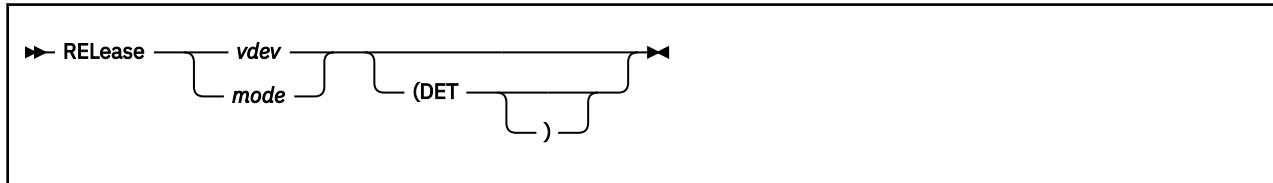
1. "Time slice X tasks ready" may provide an indication of the time it could take for a task to be redispached, if all other ready tasks were dispatched first and each ran for the specified timeslice.
2. The number of tasks ready to run will not include the tasks that are in wait state.

## Messages

For messages that apply to the QUERY command, see ["QUERY" on page 117](#).

## RELEASE

### Format



### Purpose

Use the RELEASE command to release a disk.

After an application no longer needs files on a particular disk, you should enter the RELEASE command for that disk.

### Operands

#### vdev

The virtual device number of the disk to be released.

The valid range is from X'0001' through X'FFFF' (X'0001' through X'1FFF' for 370 accommodation).

#### mode

The mode letter at which the disk is currently accessed.

#### DET

Specifies that the disk is to be detached from your virtual machine.

When the disk is detached, you receive the message:

```
DASD 'vdev' DETACHED
```

For more information on using the RELEASE command, see the [z/VM: CMS Commands and Utilities Reference](#).

### Messages

- **GCTARE006E** Invalid parameter '*parameter*'
- **GCTARE017E** DISK {*mode/vdev/volumeid*} not accessed RC=36
- **GCTARE021E** Invalid mode '*mode*'
- **GCTARE415E** Invalid device address '*vdev*'
- **GCTARE243S** Parameter list delimiter missing RC=24
- **GCTARE416E** No device specified

For more information on messages, see [z/VM: Other Components Messages and Codes](#).

# REPLY

---

## Format



## Purpose

Use the REPLY command to reply to messages sent to a GCS virtual machine operator.

GCS programs can use the WTOR macro to send a message to a GCS virtual machine operator's console and request a reply. The message may request the operator to set up certain devices for the program, provide data, or perform some other service.

The issuer of a WTOR macro expects the operator to reply. Use the REPLY command to respond to messages received through the WTOR macro.

## Operands

### *id*

The identification number (0-99), as specified in the message requesting the response. Leading zeros may be omitted.

### *text*

The text of the response to the message. The maximum text length is 119 characters (responses longer than 119 characters are truncated to 119).

### Note:

1. The WTOR macro allows its issuer to specify the maximum length of the expected operator's response. If the operator attempts to send a response that is longer than the issuer of the WTOR specified, the response will not be transmitted, and a message is issued to that effect.
2. A list of all messages awaiting reply, along with their identification numbers, can be obtained by issuing:

```
query reply
```

## Examples

```
reply 16 disk is mounted at address 250
```

The operator informs the issuer of a WTOR, whose identification number is 16, that a disk has been mounted at address 250.

## Messages

- **GCTR PY206E** Reply not accepted, ID not specified
- **GCTR PY207E** Reply not accepted, ID number not 00 to 99 RC=8
- **GCTR PY208I** Reply xx not outstanding RC=4
- **GCTR PY209E** Reply xx not accepted, reply too long for requester RC=8
- **GCTR PY210E** Reply not accepted, invalid ECB address RC=10

- **GCTRPy211E** Reply not accepted, invalid reply buffer address RC=10

For more information on messages, see [z/VM: Other Components Messages and Codes](#).

## Return Codes

---

Hex Code	Decimal Code	Meaning
X'00'	0	Your reply is accepted.
X'04'	4	No message requiring a reply is associated with the identification number you specified.
X'08'	8	Your reply was not accepted. Its format was invalid.
X'0A'	10	The reply buffer address or ECB address was not accessible.

# SET

---

## Purpose

Use the SET command to replace a saved system name entry in the SYSNAMES table for VSAM or to set or reset a particular function in your GCS machine. Only one function may be specified per SET command.

## Operands

*Table 12. Set commands.* This table lists operands that can be issued with the SET command and the location of more information for each:

Command	Location
DUMP	See “ <a href="#">SET DUMP</a> ” on page 150.
DUMPLOCK	See “ <a href="#">SET DUMPLOCK</a> ” on page 151.
IPOLL	See “ <a href="#">SET IPOLL</a> ” on page 152.
REXXSTOR	See “ <a href="#">SET REXXSTOR</a> ” on page 153.
SYSNAME	See “ <a href="#">SET SYSNAME</a> ” on page 154.
TSLICE	See “ <a href="#">SET TSLICE</a> ” on page 155.

## Usage

You can establish a default for each set command in your GROUP EXEC. For example, your GROUP EXEC has SET DUMP ON in it and you have IPLed your machine, then you can just type SET DUMP. Otherwise, if you did not have it in your GROUP EXEC, then this would have resulted with an error (322E).

When GCS is generated, the default names of saved systems for VSAM (CMSVSAM and CMSBAM) become entries in your SYSNAMES table. The table entry looks like this:

SYSNAMES:	GCSVSAM	GCSBAM
ENTRIES:	CMSVSAM	CMSBAM

GCSVSAM and GCSBAM are merely headings here. CMSVSAM and CMSBAM are the actual saved system names. Before VSAM is initialized (by the first VSAM operation after IPL), you can change these saved system names with the SET command. After you initialize VSAM, these saved system names cannot be changed.

To display the saved system names currently available to your virtual machine, enter:

```
query sysnames
```

## Messages

- **GCTSET006E** Invalid parameter ‘parameter’ RC=24
- **GCTSET013E** No function specified RC=24
- **GCTSET321E** Saved system name ‘name’ is invalid. Only GCSVSAM or GCSBAM allowed RC=24
- **GCTSET322E** New system name missing after ‘name’ RC=24
- **GCTSET323E** Parameter missing after SYSNAME RC=24
- **GCTSET351E** System name not changed. VSAM already initialized. RC=24

For more information on messages, see [z/VM: Other Components Messages and Codes](#).

## SET DUMP

### Format



### Purpose

Use the SET DUMP command to set or reset the DUMP function in your GCS machine. Only one function may be specified per SET command. During IPL, SET DUMP is initially set to DEFAULT.

### Operands

#### DUMP ON

Take a dump under all conditions.

#### DUMP OFF

No dumps are to be taken.

#### DUMP DEFAULT

The dump setting will be set to the default shown.

Dump setting:			
Dump will be produced by:	DEFAULT	ON	Off
GDUMP command	Yes	Yes	No
SDUMP macro	Yes	Yes	No
SDUMPX macro	Yes	Yes	No
ABEND DUMP parameter	Yes	Yes	No
ABEND no DUMP parameter	No	Yes	No
SYSTEM	Yes	Yes	No

### General Information about the SET Command

For usage notes and messages that apply to the SET command, see “[SET](#)” on page 149.

## SET DUMPLOCK

### Format



### Purpose

Use the SET DUMPLOCK command to set or reset the DUMPLOCK function in your GCS machine. Only one function may be specified per SET command. During IPL, SET DUMPLOCK is initially set to ON.

### Operands

#### DUMPLOCK OFF

The common storage lock will not be held while common storage is being dumped.

#### DUMPLOCK ON

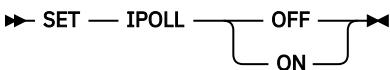
The common storage lock will be held while common storage is being dumped.

### General Information about the SET Command

For usage notes and messages that apply to the SET command, see [“SET” on page 149](#).

## **SET IPOLL**

### **Format**



### **Purpose**

Use the SET IPOLL command to set or reset the IPOLL function in your GCS machine. Only one function may be specified per SET command. During IPL, IPOLL is initially set to OFF.

### **Operands**

#### **IPOLL OFF**

Indicates that you do not want GCS to use the IUCV subfunction IPOLL to handle IUCV external interrupts. This is the initial setting.

#### **IPOLL ON**

Indicates that you want GCS to use the IUCV subfunction IPOLL to handle IUCV external interrupts and that the virtual machine is capable of handling buffered interrupts. GCS will poll pending replies and messages to provide for more efficient IUCV interrupt handling.

### **Usage**

- Buffering of interrupts may change the sequence of IUCV external interrupts presented to the virtual machine. IUCV functions, such as PURGE and SEVER, that were issued directly to CP may have affected interrupts which are now buffered to GCS. Applications that alter the usual external interrupt sequence may not be able to use the IPOLL subfunction. For additional information, see [z/VM: CP Programming Services](#).

### **General Information about the SET Command**

For usage notes and messages that apply to the SET command, see [“SET” on page 149](#).

## SET REXXSTOR

### Format



### Purpose

Use the SET REXXSTOR command to indicate that storage for REXX variables may be obtained above the 16 megabyte line.

### Operands

#### REXXSTOR 24

Indicates that storage for REXX variables must be obtained below the 16 megabyte line. This is the default.

#### REXXSTOR 31

Indicates that storage for REXX variables may be obtained above the 16 megabyte line. If this setting is specified, all programs referencing REXX variables must run in AMODE 31.

### Usage

- If REXXSTOR 31 is specified, all programs referencing REXX variables must run in AMODE 31.

### General Information about the SET Command

For *usage notes* and *messages* that apply to the SET command, see “[SET](#)” on page 149.

## SET SYSNAME

### Format

```
► SET — SYSNAME — GCSBAM — entry_name ►
      |           |
      GCSVSAM
```

### Purpose

Use the SET SYSNAME command to replace a saved system name entry in the SYSNAMES table for VSAM. Only one function may be specified per SET command.

### Operands

#### SYSNAME

Specifies that a saved system name in the SYSNAMES table is to be replaced.

#### GCSVSAM

Indicates that the entry name you are about to supply will go under the heading GCSVSAM in your SYSNAMES table. GCSVSAM does not automatically become the new entry name of the VSAM system. For more information on VSAM systems, see [“Changing GCS Default Definitions” on page 512](#).

#### GCSBAM

Indicates that the entry name you are about to supply will go under the heading GCSBAM in your SYSNAMES table. (You need a BAM system to support VSAM.) GCSBAM does not automatically become the new entry name of your BAM saved system. For more information on BAM systems, see [“Changing GCS Default Definitions” on page 512](#).

#### entry\_name

The name of the alternative saved system that will replace your default VSAM or BAM system. The VSAM and BAM systems you use for GCS can be the same as the CMSVSAM and CMSBAM systems you use for CMS. Separate systems are not required.

### Usage

The loading of the BAM and VSAM segments in a z/VM environment is supported by the GROUP exec which allows you to specify special names for your VSAM and BAM segments other than the default names of CMSBAM and CMSVSAM and automatically reserves storage for these segments for all USERIDs in the GCS group that are specified when the group exec is run.

The SET SYSNAME function can be used to replace the default names of CMSBAM and CMSVSAM in the SYSNAME table for VSAM if the following requirements are met:

1. The USERID where the SET SYSNAME is to be issued was not specified in the GROUP exec to automatically have storage reserved for the BAM and VSAM segments specified in the GROUP exec.
2. The BAM and VSAM segments that you specify in the SET SYSNAME command must be loaded above the virtual machine since storage will not be reserved at IPL time.

### General Information about the SET Command

For usage notes and messages that apply to the SET command, see [“SET” on page 149](#).

# SET TSLICE

## Format

```
► SET — TSLICE — number ►
```

## Purpose

Use the SET TSLICE command to change the dispatching time slice.

## Operands

### TSLICE *number*

*number* is a positive integer from 1 to 999, that represents the number of milliseconds that a single task will be allowed to be dispatched before the GCS dispatcher dispatches another task.

## Usage

- The default dispatching time slice of 300 milliseconds should be sufficient for most GCS applications.
- The dispatching time slice should be set lower when many tasks are running concurrently. Lowering the dispatching time slice can avoid time out problems. Use the output of the QUERY TSLICE command to determine the current time slice and the number of tasks ready to run. QUERY TSLICE also reports an estimated delay time, computed by multiplying the time slice times the number of tasks ready to run. If this estimated delay time is higher than a line delay time out, decrease the time slice value until the estimated delay time is acceptable.
- As the time slice is decreased, the overhead in the GCS dispatcher may increase. Therefore, setting the time slice lower than necessary may cause performance degradation.

## General Information about the SET Command

For usage notes and messages that apply to the SET command, see [“SET” on page 149](#).

**SET TSLICE**

# Chapter 5. GCS Macros

The GCS macros are presented in alphabetic order in this section. The data management service macros are described in Chapter 6, “QSAM and BSAM Data Management Service Macros,” on page 379 and Chapter 7, “VSAM Data Management Service Macros,” on page 417.

The GCS macros grouped by the functions they control are:

Table 13. GCS Macros (Part 1 of 2)

Task Management Service	Program Management Service	Timer Service	Console I/O Service	Unauthorized GCS Service	Authorized GCS Service
ABEND	BLDL	STIMER	WTO	AUTHCALL	AUTHNAME
ATTACH	CALL	TIME	WTOR	CMDSSI	LOCKWD
CHAP	DELETE	TTIMER		CSRCMPSC <sup>1</sup>	MACHEXIT
DEQ	ESPIE			EXECCOMM	PGLOCK
DETACH	IDENTIFY			GCSTOKEN	PGULOCK
ENQ	LINK			GENIO	SCHEDEX
ESTAE	LOAD				TASKEXIT
IHASDWA	RDJFCB				VALIDATE
POST	RETURN				
SETRP	SAVE				
WAIT	SPLEVEL				
	SYNCH				
	XCTL				

Table 14. GCS Macros (Part 2 of 2)

Storage Management Service	Serviceability	IUCV Service	Installation (Build)	Data Area
FREEMAIN	GTRACE SDUMP	IUCVCOM IUCVINI	AUTHUSER CONFIG CONTENTS RESSTOR SEGMENT	ADSR CSRYCMPS <sup>1</sup>
GCSSAVE	SDUMPX			CSRYCMPS <sup>1</sup>
GETMAIN	SYMREC			CVT DEVTYPE ECVT FLS GCSLEVEL IHADVA

## GCS Macro Level and Parameter Lists

Certain macros used by GCS have expanded parameter lists which are designed for use with 31-bit addressing mode. These parameter lists are incompatible with the 370 Accommodation Facility. However the SPLEVEL macro allows the user to select the 24-bit version or the 31-bit version of the expansion. The macros affected are:

- ATTACH
- ESTAE
- STIMER
- WTOR.

<sup>1</sup> Information on this macro is provided in the *z/VM: CMS Macros and Functions Reference*.

A program using a 24-bit parameter list can run in XA mode or XC mode with addresses below the 16MB line. If a program is to function in 31-bit addressing mode, the 31-bit version of the parameter list must be used; no modification to the program is necessary except recompilation because the default is SPLEVEL SET=2.

## Addressing Mode and the Macros

---

If a parameter passed by a program executing in 31-bit addressing mode must be located in 24-bit addressable storage, the restriction is stated in the description of the macro.

Usually a program executing in 24-bit addressing mode cannot pass addresses as parameters above 16MB in virtual storage to GCS. Some exceptions to this rule exist; as for example, a program executing in 24-bit addressing mode can:

- Free storage above 16MB using the FREEMAIN macro
- Allocate storage above 16MB using the GETMAIN macro.

## GCS Macro Formats

---

Generally, there are four possible macro formats. Each macro tells you exactly which of these formats applies and provides more detailed information. Usually, the significance of each format is as follows:

### Standard Format

Generates an in-line parameter list to the macro. It also generates nonreentrant code that executes the function as part of the macro expansion.

### List Format

Generates an in-line parameter list to the macro but generates no code that executes the function.

### List Address Format

Generates no code that executes the function. However, it does generate executable code that moves the parameter values that you specify in the instruction to a parameter list at some designated address.

### Execute Format

Generates code that executes the function. Optionally, it generates executable code that moves parameter values into a parameter list. The execute format requires that you specify the address of a parameter list that you previously created.

#### Note:

1. Not every GCS macro is available in each of these formats. However, each is available in a standard format. Several are also available in list and execute formats. A few are available in all four formats.
2. The VSAM macros listed in this book differ somewhat. See [Appendix B, “Using VSAM,” on page 517](#).

## GCS Macro Coding Conventions

Coding conventions for GCS macros are the same as those for all assembler language macros. The macro format descriptions show optional operands in the format:



indicating that if you are going to use this operand, it must be preceded by a comma (unless it is the first operand coded). If a macro statement overflows to a second line, you must use a continuation character in column 72.

**Note:** No blanks may appear between operands.

When a macro offers a choice of operands, one and only one of which must be specified, the operands are stacked one per line and shown below the line of the syntax diagram.

Many operands can be specified with an argument in the form of either an expression or a register containing a value. When this is the case, the macro expects a register designation to begin with a left parenthesis. Therefore, specifying an expression that starts with a left parenthesis will produce unpredictable results, just as specifying a register without parentheses would.

Incorrect coding of any macro may result in assembler errors and MNOTEs. MNOTEs are unnumbered responses that can result from executing system generation macros or service programs. They are documented in logic listings only.

Where applicable, the end of a macro description contains a list of the possible error conditions that may occur during the execution of the macro, and the associated return codes. These return codes are always placed in register 15. The macros that produce these return codes have ERROR= operands that allow you to specify the address of an error handling routine that can check for particular errors during macro processing. If an error occurs during macro processing and no error address is provided, execution continues with the next sequential instruction following the macro.

## Formatting Conventions

You will notice that each macro entry is accompanied by a format box that defines the proper format of the instruction.

As you examine these format boxes more closely, the first thing that you notice is the lack of blank spaces in the instructions. There are only two places where a blank space can appear in a macro. These are between the label and the instruction, and between the instruction and its first parameter. You will probably notice that the parameters themselves are not delimited by blanks, but by commas. In these respects, macros closely resemble assembler language instructions.

Let us show this by looking at a fictitious macro called DUCK. The DUCK macro takes three parameters: A, B, and C. And, like most other instructions, an optional label can be applied.

Its format box looks like this:



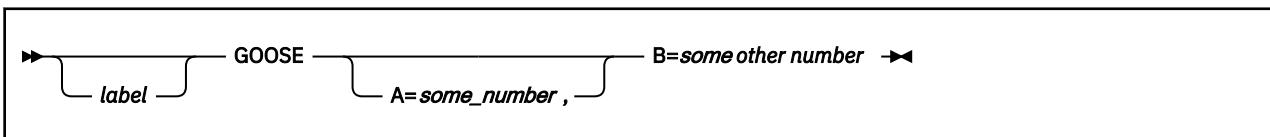
Therefore, you might code something like this:

```
QUACK DUCK A,B,C=7
```

You coded the mnemonic label QUACK and left one blank space (though more than one is permissible). Then, mindful that macros cannot be abbreviated, you followed with the full name of the macro itself, DUCK. You left another blank space, though you could have left more than one, and followed with the parameters. Notice that only comma's delimit the parameters.

Few macros are this trivial. Many instructions have parameters that are optional. Whether you choose them sometimes depends on your own needs, and sometimes on circumstances. Another fictitious macro, GOOSE, has two parameters, one of which is optional.

Its format box looks like this:



You could code GOOSE like this:

```
GOOSE B=77
```

Note that you did not supply a comma before the B parameter, because there is no other parameter present from which to separate it. Notice too that you did not supply a label this time.

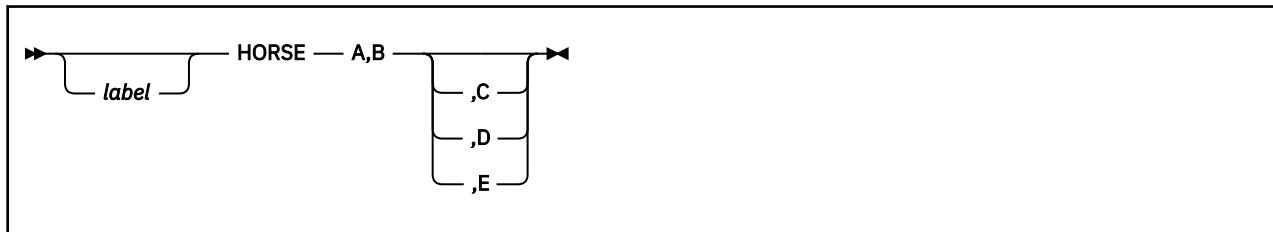
You could also code GOOSE like this:

```
HONK GOOSE A=34,B=77
```

This time you supplied the A parameter because, for some reason, it suited your purpose.

The format boxes of some macros stack optional parameters in a list.

The fictitious HORSE macro format box looks like this:



Notice that C, D, and E parameters are stacked. These stacking mean two things. First, all three of the parameters are optional. You can ignore this list entirely, if it suits your purpose, or choose from the list. Second, if you choose from the list, then you can choose either C, or D, or E. You cannot choose two or three of them, but only one.

So, if you code

```
HORSE A,B,C,D
```

it is an error because you chose two optional parameters from the same stacked list, namely C and D.

```
HORSE A,B,C
```

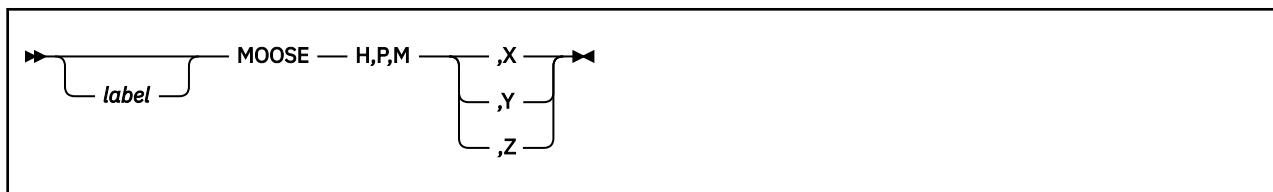
is correct because you chose only one optional parameter. Of course,

```
HORSE A,B
```

is also correct, because you chose to omit all of the optional parameters.

Some macros force you to make a choice from among a stacked list of options.

The MOOSE macro format box looks like this:



So,

```
MOOSE H,P,M
```

is incorrect, because you did not select an option from the stacked list of options. So,

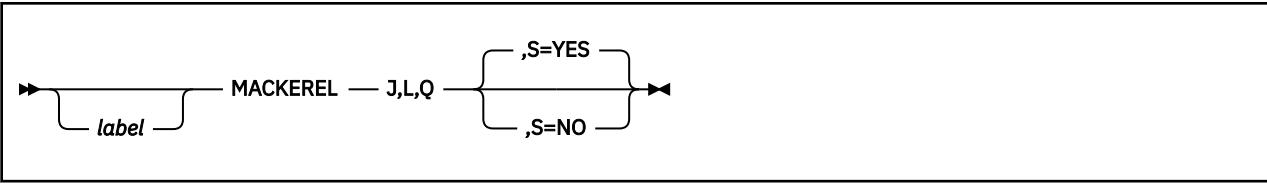
```
MOOSE H,P,M,X,Z
```

is incorrect because you selected more than one parameter from the list. So,

```
MOOSE H,P,M,Z
```

is correct because you made your choice and it was only one parameter.

The MACKEREL macro format box looks like this:



Notice that:

- The parameters ,S=YES and ,S=NO are stacked above and below the line.

It is not that difficult to figure this out if you just remember that option above the main line means this is the default and you do not have to choose. But if an option is on the line, like the X is in the MOOSE Macro, then it means you must choose. The main line through simply means that you can choose the S parameter or ignore it. So,

```
MACKEREL J,L,Q,S
```

is incorrect, because you chose the S parameter but did not choose either YES or NO. So,

```
MACKEREL J,L,Q
```

is correct, because you omitted the S parameter altogether, allowing ,S=YES to take effect by default. So,

```
MACKEREL J,L,Q,S=YES
MACKEREL J,L,Q,S=NO
```

are correct, because you specified the S parameter correctly in each.

## Parameter Notation Conventions

You will notice that under each parameter description there is a statement on how that parameter can be expressed in the macro. Several terms appear frequently in this context. They are defined as follows:

### Symbol

Any symbol that is valid in the assembler language. That is, an alphabetic character followed by 0 through 7 alphanumeric characters. A symbol cannot contain any special characters or imbedded blanks.

### Register (2) through (12)

One of the general registers 2 through 12. Presumably, the register you specify contains a right-justified value or address that pertains in some way to the parameter in question. Any high-order bits in the register should be reset to zero. You can express the register number symbolically or through an absolute expression. Unless otherwise specified, parentheses must surround the register expression.

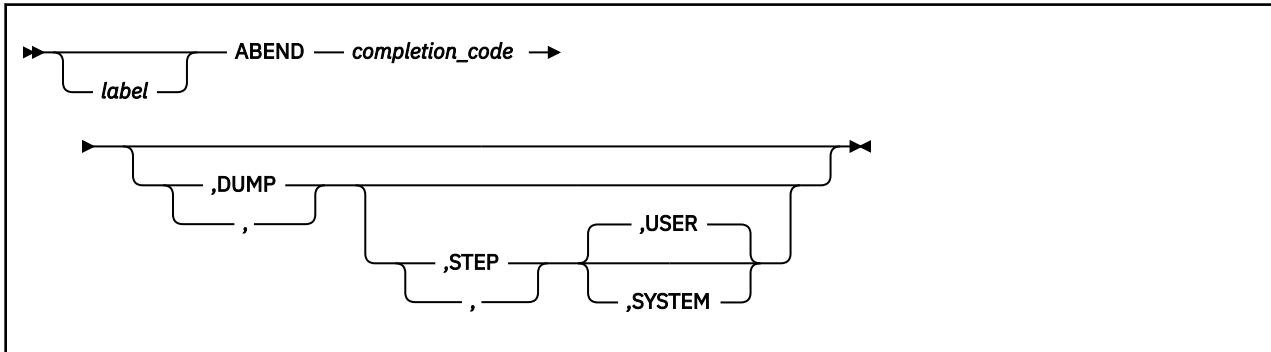
### RX-type address

Any address that is valid in an RX-type assembler language instruction.

# ABEND

---

## Format



## Purpose

Use the ABEND macro to abnormally terminate the active task.

For several reasons, task running under GCS may decide that it should abnormally terminate itself.

## Parameters

### *completion\_code*

Specifies the completion code that describes the condition under which the task terminated itself.

A completion code is a number from 0 to 4095.

If you specified the address of an event control block in the ATTACH macro that created the ABENDING task, then the completion code is placed there. (If necessary, review the entry titled “[ATTACH](#)” on page 165.) If it is a user completion code, then it is stored in bits 20-31 of the ECB completion code field. If it is a system completion code, then it is stored in bits 8-19.

If you specify the DUMP parameter, then this completion code will also appear in the dump's control block.

The meaning of each user completion code is defined by the application. The meaning of each system completion code is defined by the GCS supervisor. The USER and SYSTEM parameters, as described in the following, govern which type of completion code you receive.

You can write this parameter as any symbol, as a decimal or hexadecimal number, or as register (1) through (12).

### DUMP

GCS sends the dump to the virtual reader belonging to the member of your virtual machine group designated to receive dumps if SET DUMP is not OFF. If this member is not authorized, then only nonfetch-protected key 14 data will be included in the dump.

### STEP

Indicates that the entire command or application, of which the task in question is a part, is to be abnormally terminated.

### USER

Indicates that the completion code specified is defined by the user or the application. Unless otherwise stated, this is the case, by default.

### SYSTEM

Indicates that the completion code specified is defined by the GCS supervisor.

## Usage

1. If any subtasks are defined for the task in question, then they are also terminated abnormally. This applies to any of their descendants, as well.
2. When a task terminates, the GCS supervisor performs usual task termination activities on the former's behalf. These activities include the release of locks, storage, and other resources associated with the task.

However, you may have defined an exit routine for the task through the ESTAE macro. (If necessary, review the entry titled “ESTAE” on page 223.) The exit routine may attempt to retry the failed function or request that the supervisor continue with usual task termination.

3. It may be that no exit routine was defined for the task in question. It may also be that an exit routine was defined for the task but the exit routine directed that termination continue anyway. In either case, GCS checks to see if the task in question is a subtask of another task. If so, then the other task is the immediate ancestor task of the task in question.

If the task in question has an immediate ancestor, then GCS checks to see if the ancestor task included the ETXR parameter in the ATTACH macro it used to attach the task in question to itself. If so, then GCS schedules the routine specified in the ETXR parameter for execution. If the ancestor task specified the ECB parameter in the same ATTACH macro, then GCS posts the appropriate event control block.

See “[ATTACH](#)” on page 165.

4. Some of the subtasks of the task being terminated may have ESTAE exit routines defined for themselves. If so, none of them ever receives control.

## Examples

```
ENDIT ABEND 899,DUMP
```

The active task wants to terminate itself abnormally. A user completion code of 899 describes the reason for this. The task requests that a dump of its virtual storage be produced to aid in diagnosing the problem (only if SET DUMP is not OFF). ENDIT is the label on this instruction.

```
ENDIT ABEND 899,,STEP
```

Note the two commas in the example above. The parameters are positional.

## Return Codes and ABEND Codes

The ABEND macro does not generate return codes.

---

<b>ABEND Code</b>	<b>Meaning</b>
20D	A descendant subtask of this task issued the ABEND macro with the STEP parameter specified. This task was abnormally terminated.

---

20D	A descendant subtask of this task issued the ABEND macro with the STEP parameter specified. This task was abnormally terminated.
-----	--

---

## **ADSR**

---

### **Format**

►► ADSR ►►

### **Purpose**

Use the ADSR macro to get a symbolic name for each field in a symptom record. Each symbolic name can be used as a displacement in an assembler language instruction to gain access to the corresponding field in the symptom record. The SYMREC macro creates the symptom record.

### **Usage**

1. To use the DSECT you have created to find your way around the ADSR, assign the address of the ADSR to a base register. Then use the symbolic name of a field in the DSECT as the displacement to the corresponding field in a symptom record.
2. For more information about the format of the symptom records, see the [z/VM: CP Programming Services](#).

### **Return Codes and ABEND Codes**

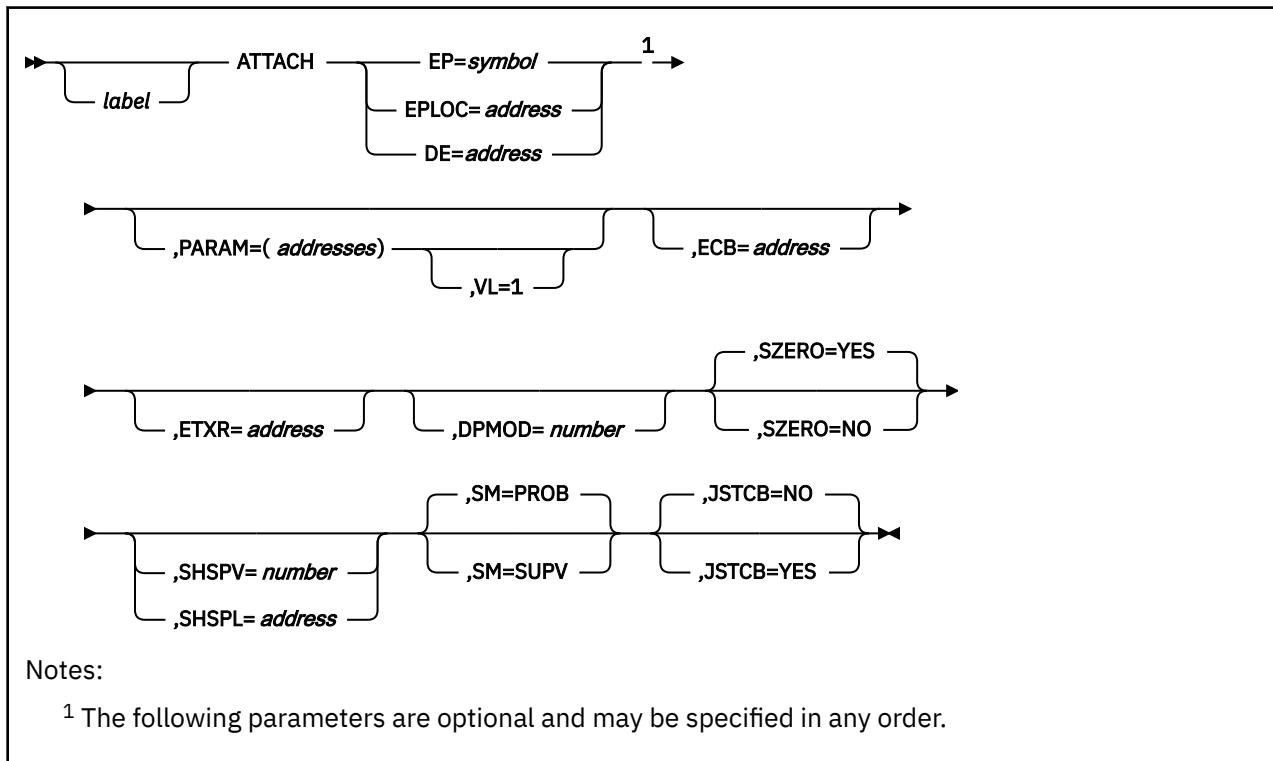
The ADSR macro generates no return codes or abend codes.

# ATTACH

The ATTACH macro is available in standard, list and execute formats.

## Standard Format

See also “List Format” on page 170 and “Execute Format” on page 171.



## Purpose

Use the ATTACH macro to set up a new subtask.

In order for the code representing a new subtask to be usable, a task block must be created for it by its immediate ancestor task. Also, the subtask's code must be brought into virtual storage if it is not already there.

The ATTACH macro should be used by a task to create a task block for one of its own new subtasks. This will bring the subtask into virtual storage if it is not already there. The task issuing the ATTACH macro thereby becomes the immediate ancestor of the subtask in question.

If running in XA mode on entry to the attached routine, the high order bit, bit 0, of register 14 is set to indicate the addressing mode of the issuer of the ATTACH macro. If bit 0 is 0, the issuer is executing in 24-bit addressing mode; if bit 0 is 1, the issuer is executing in 31-bit addressing mode.

## Parameters

### EP

Specifies the 8-byte name of the entry point within the program that receives control when your new subtask runs.

The entry point name can be any one of the following:

## ATTACH

- The name of the entry point as previously defined through the IDENTIFY macro. See “[IDENTIFY](#)” on [page 270](#).
- The name of the entry point declared in a shared segment directory through the CONTENTS macro. See “[CONTENTS](#)” on [page 197](#).
- A member name (or alias) in the directory of a load library.

When looking for the entry point name that you specify, GCS searches:

1. Your private storage, because the module associated with the entry point name may already be loaded.
2. Any shared segment directories that may have been created through the CONTENTS macro.
3. The directories of any load libraries that may have been defined for your virtual machine through the GLOBAL LOADLIB command. See “[GLOBAL](#)” on [page 101](#).

If the subtask code is in a load library, then the ATTACH macro will bring the subtask's code into your private storage for you.

You must write this parameter as a symbol.

### **EPLOC**

Specifies the address that contains the 8-byte name of the entry point of the program that receives control when your new subtask runs.

You can write this parameter as an assembler program label or as register (2) through (12).

### **DE**

Specifies the address of the NAME field within the directory list entry associated with the entry point.

This is the same list entry you placed in the directory using the BLDL macro. See “[BLDL](#)” on [page 181](#).

You can write this parameter as an assembler program label or as register (2) through (12).

### **PARAM**

Specifies one or more parameter addresses that are to be passed to your subtask program after it receives control.

GCS builds a parameter list containing these addresses in the order which you specify them. Then, the address of this parameter list is passed in register 1 to your subtask program.

Note that this parameter list must be surrounded by parentheses and each member of the list must be separated from the others by a comma.

You can write these parameters as assembler program labels or as registers (2) through (12).

### **VL=1**

Indicates that the subtask expects a variable number of parameters to be passed to it.

This parameter causes the high-order bit of the last parameter address in the list to be set to 1. This enables the subtask to find the end of a variable-length parameter list.

You must write this parameter exactly as shown. And, you can use it only with the PARAM parameter. To omit the VL=1 parameter is to say that the subtask expects a set number of parameters.

### **ECB**

Specifies the event control block (ECB) associated with your new subtask.

“[WAIT](#)” on [page 365](#) and “[POST](#)” on [page 314](#) describe how your new subtask can be treated as an event associated with an ECB. GCS posts the ECB with the subtask's completion code or return code when the latter terminates.

Remember, if you specify the address of an ECB in the ATTACH macro, then you must explicitly enter the DETACH macro when you are finished with the subtask in question. The DETACH macro releases all the storage associated with your subtask, including its control blocks. See “[DETACH](#)” on [page 208](#).

You can write this parameter as an assembler program label or as register (2) through (12).

**ETXR**

Specifies the address of the end-of-task exit routine that is to receive control when your new subtask terminates either normally or abnormally.

It is your responsibility to provide this exit routine and to be certain that it is in virtual storage when needed. Also, if your exit routine is to be shared by several subtasks, then it must be reentrant.

If in XA mode, the exit will be run in the AMODE of the caller.

Remember, if you specify the address of an exit routine in the ATTACH macro, then you must explicitly enter the DETACH macro when you are finished with the subtask in question. Normally the DETACH macro is issued somewhere in the exit routine itself.

You can write this parameter as an assembler program label or as registers (2) through (12).

**DPMOD**

Specifies the number that is to be added to the dispatching priority of the immediate ancestor task to produce the dispatching priority of your new subtask.

The larger the dispatching priority number of a task, the more readily the task is executed. So, if a positive number were assigned to the DPMOD parameter, then the sum of this number and the priority of the ancestor task would produce a higher priority for your new subtask. Conversely, a negative number assigned to the DPMOD parameter would result in a priority for your subtask that is lower than its immediate ancestor.

The dispatching priority for a problem state application task must be a number from 0 to 240. Should the sum of the DPMOD parameter and the priority of the ancestor task be less than zero, then the dispatching priority of your subtask will be 0. If this sum is greater than 240, then the dispatching priority of your subtask will be 240.

The dispatching priority for a supervisor state application task must be a number from 0 to 250. Should the sum of the DPMOD parameter and the priority of the ancestor task be less than zero, then the dispatching priority of your subtask will be 0. If this sum is greater than 250, then the dispatching priority of your subtask will be 250.

**Note:** If the task issuing the ATTACH macro is running on the system task, then the dispatching priority for its subtask will be the sum of 240 plus the value assigned to the DPMOD parameter.

**SZERO**

Indicates whether your new subtask is to share subpool 0 storage with its immediate ancestor task.

A subpool is a number from 0 to 255 that is assigned to a block of storage to describe its characteristics. Subpool 0 specifies private, fetch-protected storage.

If a main task issues the GETMAIN macro for storage in subpool 0, then GCS automatically frees the storage when the task terminates. Also, for a subtask that is attached to a main task with the SZERO=NO parameter specified.

However, if the subtask was attached with the SZERO=YES parameter specified (or defaulted), then GCS associates the storage with the oldest ancestor task with which this subtask is sharing the subpool. The storage block is not automatically freed by GCS when the subtask terminates. The storage is freed only when the oldest ancestor task terminates.

**YES**

Specifies that subpool 0 storage will be shared by your new subtask with its immediate ancestor task. This is the case, by default.

**NO**

Specifies that subpool 0 storage will not be shared by them.

**SHSPV**

Specifies a storage subpool that will be shared by your new subtask with its immediate ancestor (and with the latter's ancestor, if it shares with the task that attached it).

## **ATTACH**

If a main task issues the GETMAIN macro for storage from subpools 1 through 127, then GCS automatically frees the storage when the task terminates. Also, for a subtask that was attached to that task without a subpool having been specified in the SHSPV or SHSPL parameter.

However, if the subtask was attached with a subpool specified in the SHSPV or SHSPL parameter in the ATTACH macro, then GCS associates the storage with the oldest ancestor task with which this subtask is sharing the subpool. Hence, the storage is not automatically freed by GCS when the subtask terminates. The storage is freed only when the oldest ancestor task terminates.

Because subpools greater than 127 cannot be shared, you should write this parameter as a number from 1 to 127.

### **SHSPL**

Specifies the address of a list of subpool numbers, each of which refers to a subpool to be shared by your new subtask with its immediate ancestor task.

The rules governing the SHSPV parameter also apply here. Besides, the first byte in the list must contain the number of bytes remaining in the list. Each byte following must contain a subpool number from 1 to 127.

You can write this parameter as an assembler program label or as register (2) through (12).

### **SM**

Indicates the state which your new subtask will run. This parameter is valid only if the task issuing the ATTACH macro is running in supervisor state. Otherwise this parameter is ignored.

### **PROB**

Indicates that your new subtask will run in problem state. If you omit the SM parameter altogether, then the subtask will run in problem state, by default.

### **SUPV**

Indicates that your new subtask will run in supervisor state.

### **JSTCB**

Indicates whether your new subtask is an independent application. Unless your program is running on the system task, this parameter is ignored.

### **YES**

Indicates that your subtask is an independent application.

An independent application does not go away when the command through which it was created terminates. This means that the application must be explicitly detached through the DETACH macro when it is no longer needed.

### **NO**

Indicates that your subtask is not an independent application. This is the case, by default.

## **Usage**

1. The ATTACH macro does not transfer control to your new subtask. It merely sets up a task block for your subtask based on the information you provide in the ATTACH macro.  
When the new subtask is dispatched the first time, it receives control. At this point, the programs it contains are enabled for interrupts. Also, the subtask runs in the same key which its ancestor task ran when the latter issued the ATTACH macro.
2. The ATTACH macro assigns a unique task identifier to each new subtask. This task ID is returned to the task issuing the instruction in the 2 low-order bytes of register 1. Further, the 2 high-order bytes of this register will contain the appropriate virtual machine ID.

This task ID refers to your new subtask if you decide to delete it from the system or change its dispatching priority. See "[DETACH](#)" on page 208 and "[CHAP](#)" on page 187.

**Note:** Soon after the ATTACH macro completes execution, be certain to save the task id somewhere in virtual storage. You will need this task id later as a parameter to the DETACH and CHAP macros.

3. Do not use the ATTACH macro in an ESTAE exit routine.

4. An end-of-task exit routine will always run in the same key and state as the task that issued the ATTACH macro originally.
5. If neither the ECB nor ETXR parameter is specified, then the subtask is automatically removed from the system when it terminates.
6. The ATTACH macro can attach a load module in either 24-bit or 31-bit addressing mode and which is physically resident above or below the 16MB line. The AMODE and RMODE attributes (located in the directory entry for the load module) provide this information. RMODE indicates where the module is to be placed; AMODE indicates the addressing mode of the module. If the AMODE of the entry module being attached is ANY, it will be attached with the same addressing mode as the caller.
7. The SPLEVEL macro need not be issued unless you want an ATTACH macro used by GCS that has an expanded parameter list, which is designed for use in the 31-bit addressing mode. A 31-bit parameter list is incompatible if you are running under the 370 Accommodation Facility. However the SPLEVEL macro lets you select either the 24-bit version or the 31-bit version of the expansion.
8. When an exit routine specified in an ATTACH macro receives control, the contents of the registers are:

<b>Register</b>	<b>Contents</b>
0	Unpredictable.
1	The task ID for the subtask that just terminated.
2 - 12	Unpredictable.
13	The address of an 18 word register save area provided by the GCS supervisor.
14	The return address within the GCS supervisor.
15	The address of the exit routine.

9. When the new subtask receives control, the contents of the registers are:

<b>Register</b>	<b>Contents</b>
0	Unpredictable.
1 - 12	Propagated to the new subtask.
13	The address of a new user save area.
14	The return address within the ancestor task.
15	The address of the entry point.

10. If the program that receives control after the new subtask becomes active is reentrant, then it is loaded into key 0 storage. This ensures that it is not accidentally modified or tampered with.
11. This macro supports both 24 and 31 bit address expansions of the parameter list. The macro expansion is controlled by the internal macro SPLEVEL. The default value is 31.

## Examples

```
ATTACH EPLOC=(4),PARAM=((5),(6),(7)),VL=1,ECB=MYECB,SHSPL=SPLIST
```

A task requests that a new subtask be created.

The name of the entry point for the program associated with the new subtask can be found at the address in register 4. Registers 5, 6, and 7 contain the addresses of three parameters to be passed as a list to the subtask's program when it receives control. Because the new subtask's program can accept a variable number of parameters, the VL=1 parameter is specified. The event control block associated with the new subtask can be found at the address associated with the label MYECB. A list of storage subpools that are to be shared by the subtask with its immediate ancestor task can be found at the address associated with the label SPLIST.

## Return Codes and ABEND Codes

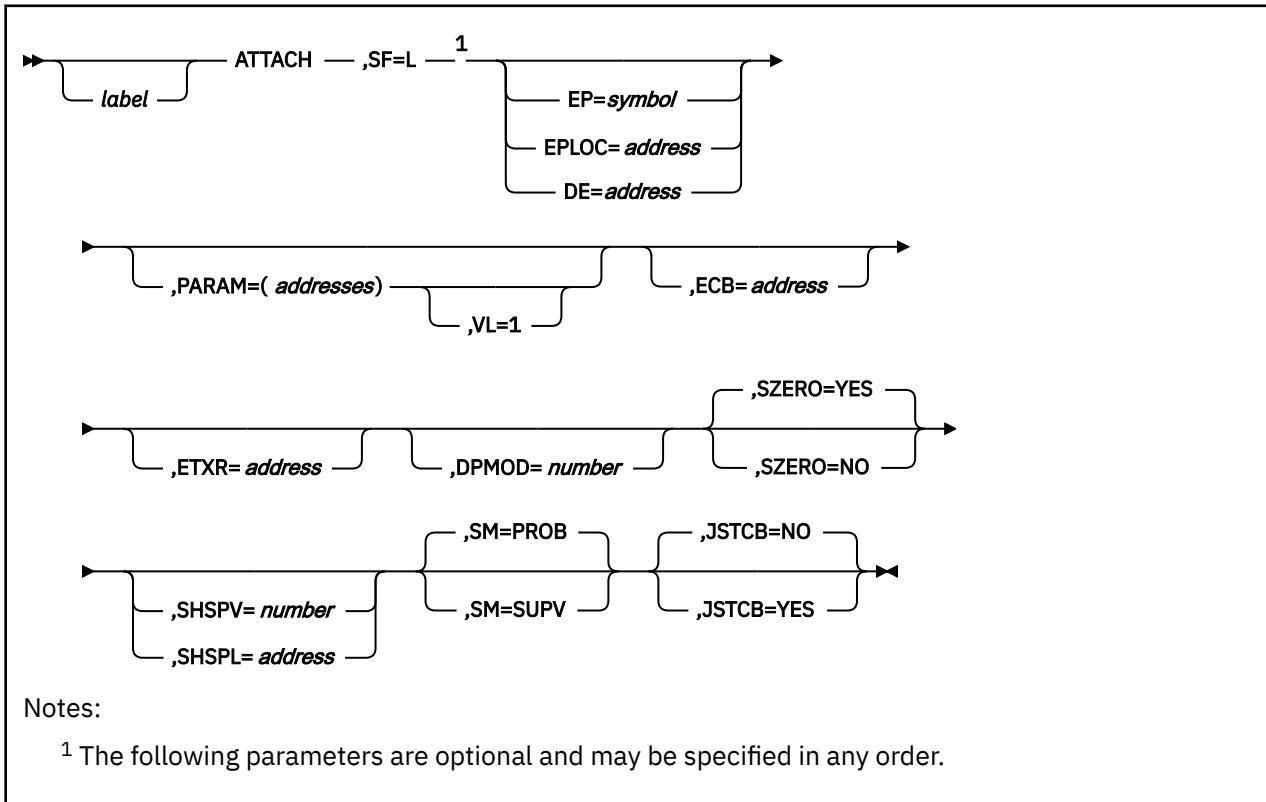
When this macro completes processing, it passes to the caller a return code in register 15.

Hex Code	Decimal Code	Meaning
X'00'	0	Function successfully completed.
X'04'	4	An ATTACH macro was issued in an ESTAE exit routine. The subtask was not attached.

### ABEND Code Meaning

22A	You specified a subpool number greater than 127 in the SHSPL or SHSPV parameter.
42A	The ECB parameter specified an invalid address.
704	An uncorrectable machine, system, or indeterminate error occurred while processing the GETMAIN macro.
72A	Invalid parameter list.

## List Format



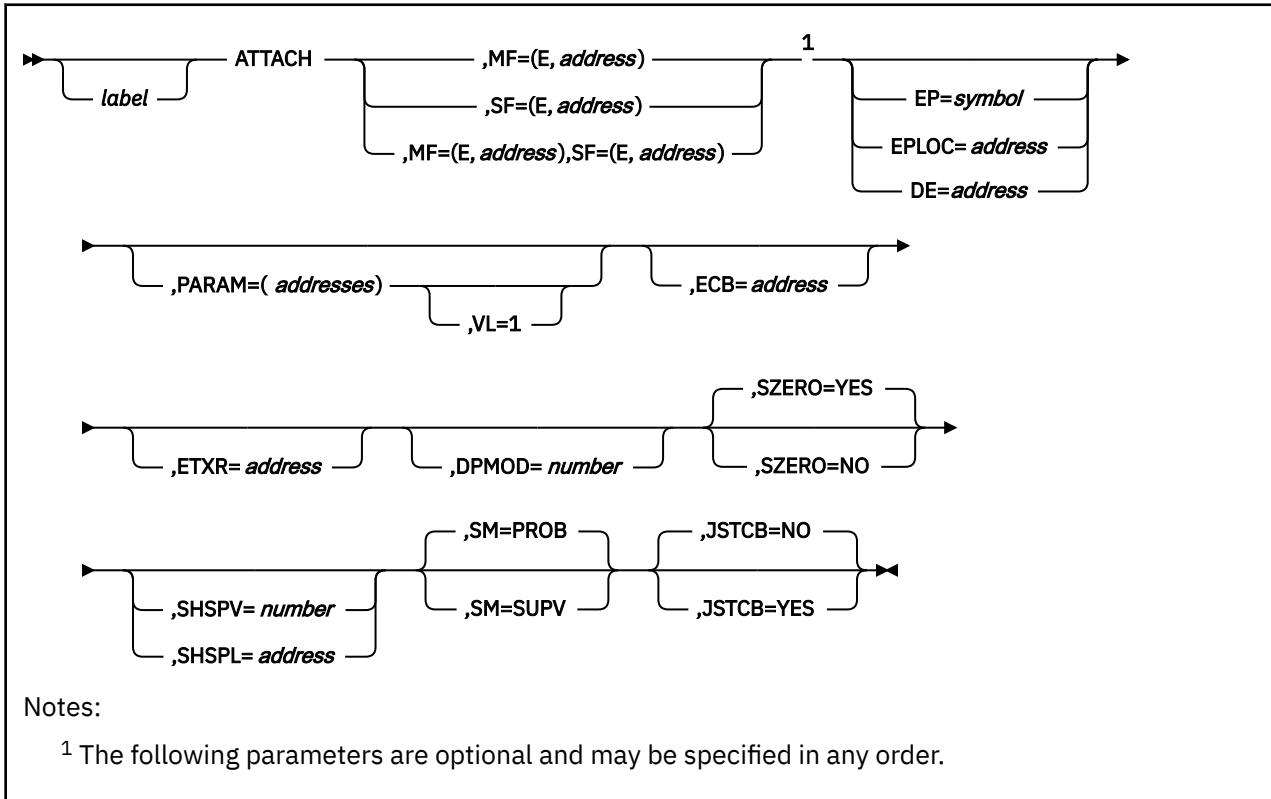
## Purpose (List Format)

This format of the macro generates an in-line parameter list based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation.

## Added Parameter

**SF=L**

Specifies the list format of this macro.

**Execute Format****Purpose (Execute Format)**

This format of the macro generates code that executes the function, using a parameter list whose address you specify.

Problem program or control program parameters specified are provided in parameter lists expanded in-line.

**Added Parameter (Execute Format)****MF=(E, address)**

*address* specifies the address of the remote parameter list to be used by the program that receives control when the new task becomes active.

You can add or modify values in this parameter list by specifying them in this instruction.

**SF=(E, address)**

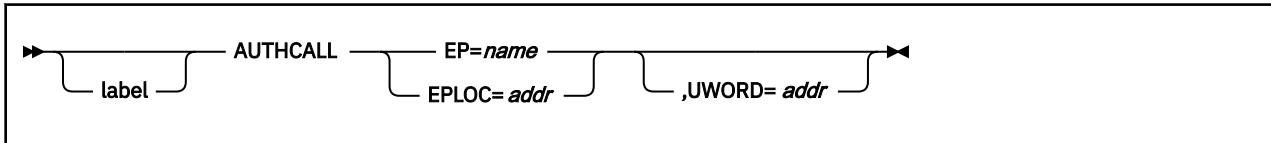
*address* specifies the address of the parameter list passed to the control program and provided by the list form of the ATTACH macro.

*address* specifies the address of the parameter list to be used by the macro that you generate using the list form of the ATTACH macro.

You can add or modify values in this parameter list by specifying them in this instruction.

## AUTHCALL

### Format



### Purpose

Use the AUTHCALL macro to call an authorized program from an unauthorized program. An important feature of GCS is that it permits an authorized program to be called by an unauthorized program. The authorized program resides in a shared segment, having been linked to its virtual machine at GCS initialization time. The unauthorized program resides in one of the virtual machines that makes up the group.

**Note:** In this context, an *authorized program* is one running in supervisor state, an *unauthorized program* is one running in problem state.

The AUTHCALL macro allows an unauthorized program to call an authorized program. However, AUTHCALL is not an authorized GCS function.

### Parameters

#### EP

Specifies the name by which the authorized program is known to the unauthorized program. Note that this name is from one to eight alphanumeric characters long.

#### EPLOC

Specifies the address at which the name of the authorized program can be found. Again, this is the name by which the authorized program is known to the unauthorized program.

You can write this address as an assembler program label, as register (0), or as register (2) through (12). The name of the authorized program, as stored at this address, should be padded on the right with blanks if the name occupies fewer than 8 bytes.

#### UWORD

Specifies an optional fullword address that may be passed to the authorized program when it is called by the unauthorized program.

You can use this parameter to pass any information you wish to the authorized program.

The UWORD may be written as an assembler program label or as register (1) through (12). If you write it as a label, then the UWORD is passed to the authorized program as the address associated with that label. If you write it as a register, then the UWORD is passed to the authorized program as the contents of that register. If no UWORD is specified, it is passed as the value zero.

### Usage

1. It is impossible for an unauthorized program to call an authorized program through the AUTHCALL macro unless the AUTHNAME macro is issued for that authorized program first. If necessary, review the entry titled “[AUTHNAME](#)” on page 174.
2. Any program started through the AUTHCALL macro runs in key 0.
3. Any program started through the AUTHCALL macro will run in the AMODE specified for it in the CONTENTS macro. See “[CONTENTS](#)” on page 197.

## Examples

```
AUTHCALL EP=PATH
```

Calls an authorized program named PATH.

```
AUTHCALL EPLOC=(2),UWORD=(5)
```

Calls an authorized program whose name can be found at the address in register 2. Register 5 contains information that the program expects to receive from the program that called it.

The authorized program being called receives the following information in its registers.

Register	Contents
0	The user word (UWORD) specified in the associated AUTHNAME macro.
1	The user word (UWORD) specified in the AUTHCALL macro.
13	The address of the register save area.
14	The address to which control is to return after the authorized program completes execution.
15	The address of the entry point in the program being called.

## Return Codes and ABEND Codes

Except for the return code noted in the following, the authorized program will pass its return code to the program that called it in register 15. The AUTHCALL macro generates the following return code.

If you receive a return code of -3 in register 15, do not mistake it for a return code generated by the program that you called.

Hex Code	Decimal Code	Meaning
X'FFFFFD'	-3	The system could not find the program whose address you specified.
X'30'	48	The CONTENTS entry has AMODE=CONTENTS or AMODE=CALLER, the caller is in AMODE 24 and the exit address is above the 16MB line.

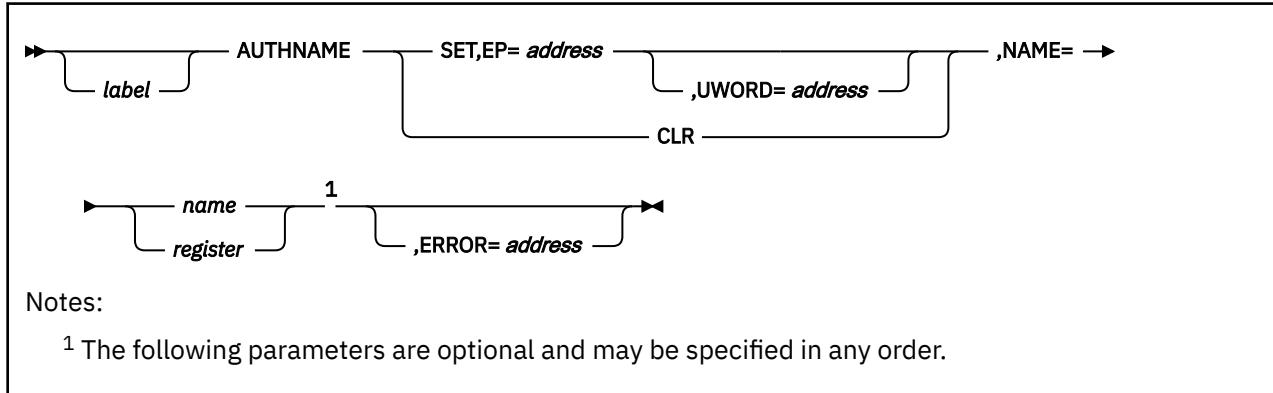
ABEND Code	Reason Code	Meaning
FCB	0100	A call was made to an authorized program that is not available to the unauthorized program.
FCB	0102	The GETMAIN macro, issued by GCS, was unable to obtain enough storage to complete your request.

## AUTHNAME

The AUTHNAME macro is available in standard, list, list address and execute formats.

### Standard Format

See also [“List Format” on page 176](#), [“List Address Format” on page 177](#) and [“Execute Format” on page 177](#).



### Purpose

Use the AUTHNAME macro to define or withdraw the name of an authorized program that is to be called by an unauthorized program.

An important feature of GCS is that it permits an authorized program to be called by an unauthorized program. The authorized program resides in a shared segment that was linked to its virtual machine at GCS initialization time. The unauthorized application resides in one of the virtual machines that makes up the virtual machine group.

**Note:** In this context, an *authorized program* is one running in supervisor state, an *unauthorized program* is one running in problem state.

The AUTHNAME macro creates (or clears, depending on your intent) a control block that contains information the unauthorized program needs to call the authorized program. This information includes, among other things, the name by which the authorized program is known by the various applications within the virtual machine group; the address of the authorized program; the key which the calling program is running; the state of the calling program (problem or supervisor); and the address of a user-defined fullword, which will be described later.

### Parameters

#### SET

Indicates that a control block is to be created for the authorized program in question.

After this is done, the unauthorized program can call the authorized program.

#### EP

Specifies the address of the authorized program in question.

The authorized program must be resident in a shared segment. That is, it must be a program whose entry point is defined in a shared segment directory that was created through the CONTENTS macro. See [“CONTENTS” on page 197](#).

The AMODE of the authorized program will be taken from the correspondent entry in the CONTENTS macro. See “[CONTENTS](#)” on page 197.

This parameter is required when you use the SET option. The parameter is meaningless with the CLR parameter.

You can write this parameter as an assembler program label or as register (2) through (12).

#### **UWORD**

A fullword of storage in the control block that you can use in any way you please.

For example, perhaps the authorized program expects the address of a parameter list or some other value be passed to it. You can use the UWORD for that, if you wish. However, this parameter has meaning only when used with the SET parameter.

You can write this parameter as an assembler program label or as register (2) through (12). If you write it as a label, then the content at the label or the address itself is placed in the UWORD field of the control block. If you write it as a register, then the contents of that register are placed in the UWORD field. If you omit this parameter altogether, then it is passed as a fullword of zeros.

#### **CLR**

Indicates that the authorized program in question is no longer needed by any unauthorized program. Therefore, the control block for the authorized program is cleared away.

#### **NAME**

Specifies the name by which the authorized program is known to the unauthorized program.

If you choose the SET parameter, then this name refers to the authorized program for which a control block is to be created. If you choose the CLR option, then the name refers to the authorized program that is no longer needed and whose control block is to be cleared.

Note that this name can be no more than eight characters long.

You can write this parameter as the name of the program itself, or you can write it as register (2) through (12). If you do the latter, then the register must contain the address where the name is stored.

#### **ERROR**

Specifies the address of the routine that is to receive control if an error occurs in the AUTHNAME macro.

You can omit this parameter if you wish, test the return code from the macro, and proceed in an appropriate manner.

Otherwise, you can write this parameter as an assembler program label or as register (2) through (12).

### **Usage**

1. The authorized program is always loaded at GCS initialization time. It is possible for one virtual machine to call this program after another machine has cleared it. This is because of the time lag between issuing the CLR function and completing it. The authorized program should be designed with this in mind. See “[AUTHCALL](#)” on page 172.
2. It is impossible for an unauthorized program to call an authorized program through the AUTHCALL macro unless the AUTHNAME macro has been issued for the authorized program first. The control block created by the AUTHNAME macro is, in effect, *permission* for an unauthorized program to call an authorized program.
3. Generally, the AUTHNAME macro is issued by an authorized program (RSCS or VTAM, for example) for unauthorized programs.
4. The authorized program called by the unauthorized program (through AUTHCALL) will have the same PSW key as the program that issued the corresponding AUTHNAME macro.
5. The AUTHNAME macro places the control block for an authorized program in common storage. Hence, any unauthorized application in the group can call it.
6. If the task that issued the AUTHNAME macro terminates, the AUTHNAME is no longer in effect.

**Examples**

```
AUTHNAME SET,NAME=BLUE,EP=(3)
```

Make an authorized program available to unauthorized programs. The authorized program will be known to the unauthorized programs as BLUE. The address of this authorized program is in register 3.

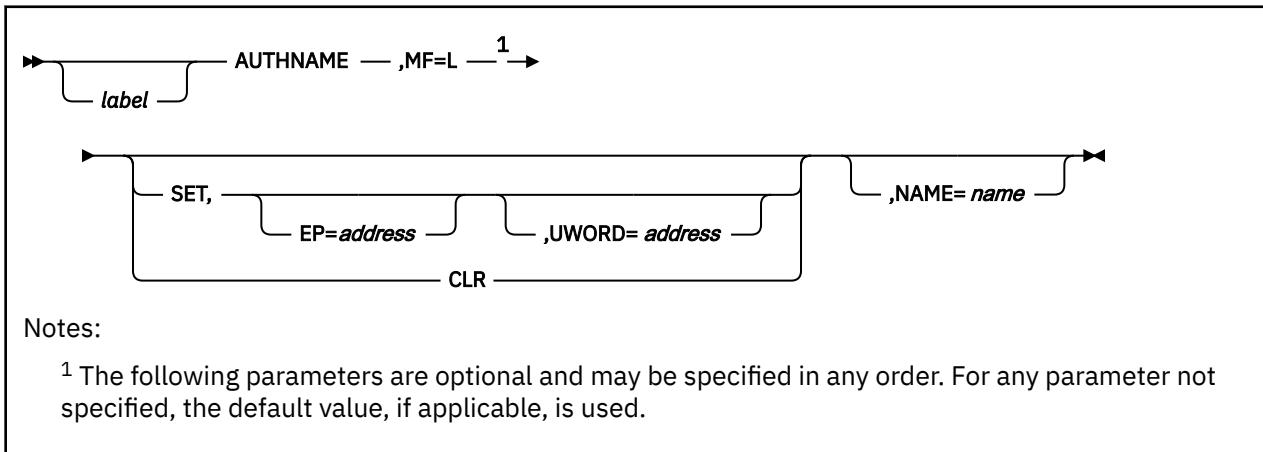
```
AUTHNAME SET,NAME=RED,EP=PURPLE,ERROR=REDERR
```

Make an authorized program available to unauthorized programs. The authorized program will be known to the unauthorized programs as RED. This program can be found at the address associated with the label PURPLE. If an error occurs in the AUTHNAME macro, control will be transferred to the routine at the address associated with the label REDERR.

**Return Codes and ABEND Codes**

When this macro completes processing, it passes to the caller a return code in register 15.

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'00'	0	Request was completed normally.
X'04'	4	A control block already exists for this authorized program.
X'08'	8	The address you specified for the EP parameter is not in a shared segment.
X'18'	24	Parameter list is invalid.
X'2C'	44	No authorized program has the name you specified.

**List Format****Purpose (List Format)**

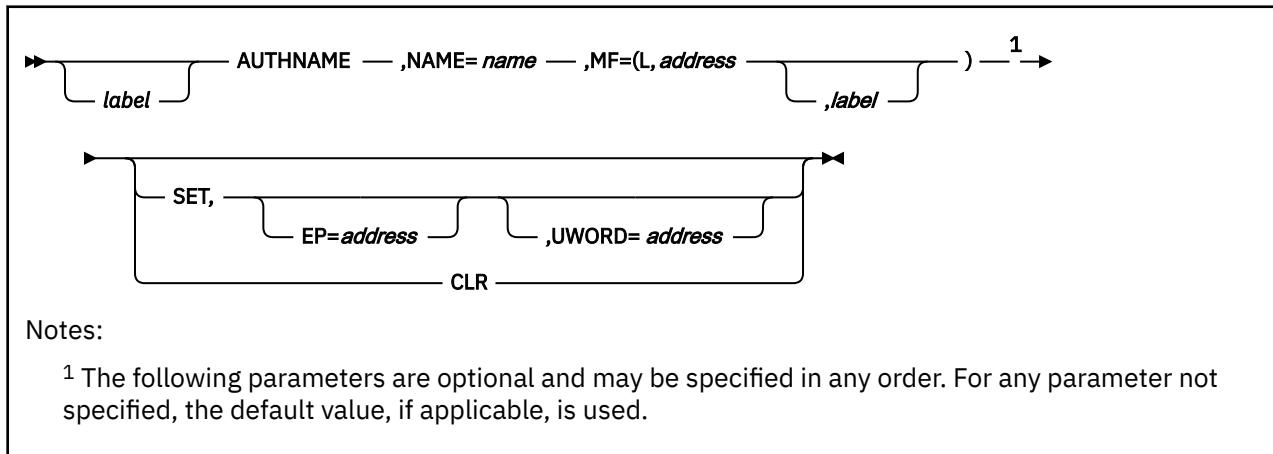
This format of the macro generates an in-line parameter list based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation. Also, note that only the preceding parameters listed are valid in the list format of this macro.

**Added Parameter**

**MF=L**

Specifies the list format of this macro.

### List Address Format



### Purpose (List Address Format)

This format of the macro does not produce any executable code that calls the function. However, it does produce executable code that moves the parameter values that you specify into a certain parameter list. If you enter the macro using this format, then you must do so before any related invocation of the macro using the execute format. Note that only the preceding parameters listed are valid in the list address format of this macro.

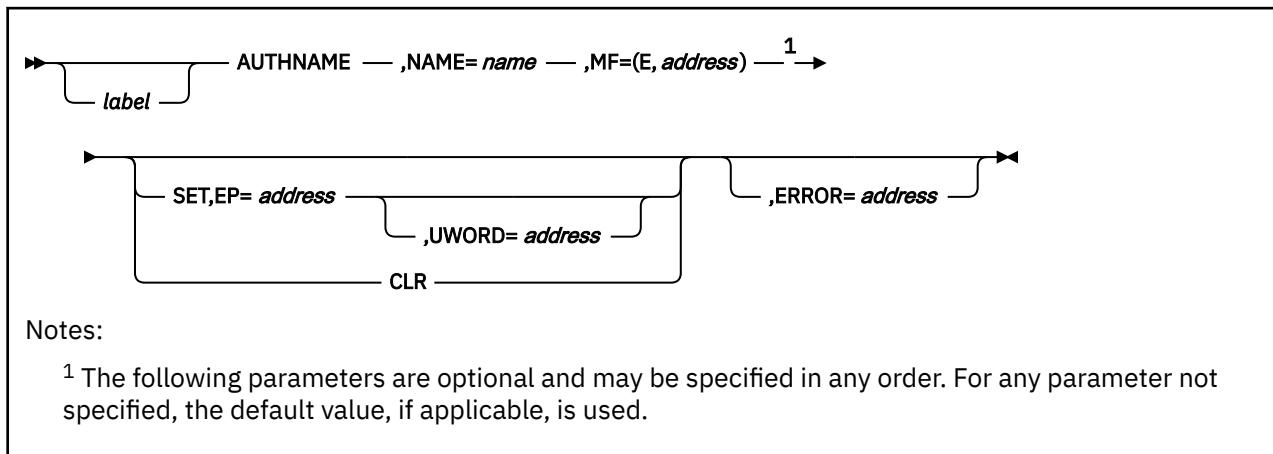
### Added Parameter (List Address Format)

#### **MF=(L,*address*,*label*)**

*address* specifies the address of the parameter list into which you want the parameter values you mention placed. This address can be within your program or somewhere in free storage.

*label* is optional and is a user-specified label, indicating that you want to determine the length of the parameter list. The macro expansion equates the label you specify with the length of the parameter list.

### Execute Format



**Purpose (Execute Format)**

This format of the macro generates code that executes the function using a parameter list whose address you specify. Note that only the preceding parameters listed are valid in the list address format of this macro.

**Added Parameter (Execute Format)****MF=(E,*address*)**

*address* specifies the address of the parameter list to be used by the macro.

You can add or modify values in this parameter list by specifying them in this macro.

# AUTHUSER

---

## Format



## Purpose

Use the AUTHUSER macro to create an authorized user block for the GROUP CONFIGURATION FILE.

The GROUP CONFIGURATION FILE describes a GCS virtual machine group. This file is divided into three data blocks:

### Configuration

Defines the virtual machine group's configuration so that it conforms to the needs of your installation. See “[CONFIG](#)” on page 193.

### Segment

Identifies which saved segments will be automatically linked to each member of the group at IPL time. See “[SEGMENT](#)” on page 338.

### Authorized User

Identifies which members of the group are *authorized*. That is, which members are permitted to perform authorized GCS functions. The Authorized GCS Service Macros are identified in [Chapter 5, “GCS Macros,” on page 157](#).

## Parameters

### START

Indicates that this AUTHUSER macro marks the beginning of the authorized user block.

The authorized user block must begin with an AUTHUSER macro with this parameter specified.

### NAME

Specifies the user ID of the virtual machine that is to have authorized status.

### END

Indicates that this AUTHUSER macro marks the end of the authorized user block.

The authorized user block must end with an AUTHUSER macro with this parameter specified.

## Usage

1. Most installations will not explicitly use the AUTHUSER macro to build the GROUP CONFIGURATION FILE. Those equipped with at least one full-screen display terminal can take advantage of GCS build panels. These data entry panels, called by the GROUP command, eliminate the need to build the file by explicitly coding these macros. When you call the GROUP command without a full-screen terminal, your file will have to be built using the editor and coding the macros manually.
2. The GROUP CONFIGURATION FILE adopts the system name as its file name. This name corresponds exactly with that specified in the SYSNAME parameter of the CONFIG macro. The file type of the GROUP CONFIGURATION FILE is always GROUP.

3. Remember that in using the AUTHUSER macro you are creating blocks of information. Thus, all occurrences of the AUTHUSER macro must be physically grouped together in the GROUP CONFIGURATION FILE.

### **Examples**

This example illustrates the authorized user block of a GROUP CONFIGURATION FILE.

```
.  
. .  
AUTHUSER START  
AUTHUSER NAME=GSC455JX  
AUTHUSER NAME=NHGT78FC  
AUTHUSER NAME=KJGR99BV  
AUTHUSER NAME=KJGD03NJ  
AUTHUSER END  
. .  
. .
```

The block begins with the AUTHUSER macro with the START parameter specified. Four user IDs are then specified, indicating that these virtual machines are to have authorized status. The authorized block is then concluded with an AUTHUSER macro with the END parameter specified.

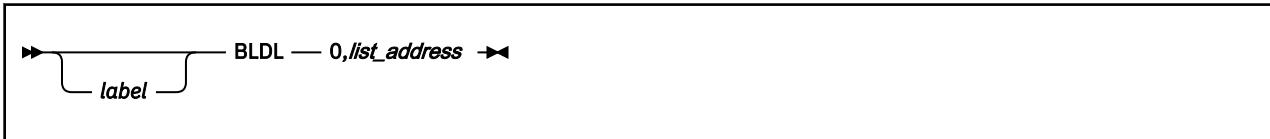
### **Return Codes and ABEND Codes**

The AUTHUSER macro generates no return codes and no ABEND codes.

# BLDL

---

## Format



## Purpose

Use the BLDL macro to build a directory entry list to aid in invoking one or more load library members.

Frequently the programs you write to run under GCS need to call other programs. Some of these programs may be modules resident in load libraries stored on disks. To bring a member of a load library into virtual storage and run it, GCS needs certain information: the module's name, the load library of which the module is a member, the module's address on the disk, relocation information, and so forth.

Your program can issue the BLDL macro to build a directory entry list for each load library member expected to be called. The needed information is extracted from the directory of the load library containing the module and placed in the directory entry list.

If you do not enter the BLDL macro, then GCS will do it for you whenever you load a new module. This is satisfactory if you plan to load, use, and delete the module only once. However, if you plan to use the same module several times, it is more efficient for you to enter the BLDL macro once. That way, the module can be loaded once and executed several times using the same directory entry list.

## Parameters

**0**

The numeral zero, written exactly as shown.

It indicates that the BLDL macro is to search for the information it needs only in the directories of the load libraries identified previously in your GLOBAL LOADLIB command.

For more information on the GLOBAL command see [“GLOBAL” on page 101](#).

***list\_address***

Specifies the address of the directory entry list.

The skeleton for this list (and certain basic information for it) must be provided by your program. The discussion of the skeleton follows.

You can write this address as an RX-type address, as register (0), or as register (2) through (12).

## Specify the ASIT and ranges as follows:

As mentioned before, your program must provide the storage necessary for the directory entry list. It must also provide certain information about the list, and the names of the modules the list is to describe. The BLDL macro then fills in the blanks with information necessary for the invocation of the modules.

The basic format of a directory entry list is:

LIST INFORMATION	LIST ENTRY #1	LIST ENTRY #2	LIST ENTRY #3	....	LIST ENTRY #64K-1
------------------	---------------	---------------	---------------	------	-------------------

## List Information 1

The list information for the directory entry list is contained in the first two fields. Note that the numbers in parentheses indicate the number of bytes in each field.

FF (2)	LL (2)
--------	--------

These fields are described as follows:

### FF

Indicates the number of separate list entries in the directory entry list. It must be a binary number corresponding to the number of modules your list will describe.

### LL

Indicates the length of each separate list entry in the directory entry list, in bytes. It must be a binary number of at least 62 and it must be even.

## List Information 2

The directory entry list contains one or more list entries, as shown previously. Each list entry corresponds to one module from a load library that you intend to run. A single list entry is composed of the following fields. The number of bytes are in parentheses.

NAME (8)	TTR (3)	K (1)	Z (1)	C (1)	UD (at least 48)
----------	---------	-------	-------	-------	------------------

You need only supply one field in the list entry yourself:

### NAME

The name of the module (or its alias) that the particular list entry will describe.

This name must start in the first byte of this field. If the name is fewer than 8 bytes long, it must be padded on the right with blanks.

The list information and the name of each module is all the information your program has to supply. The remaining fields within each list entry are filled in by the BLDL macro. The significance of these fields is as follows:

### TTR

The relative position where the module may be found in the load library.

### K

Identifies the load library of which the program is a member. It is a number specifying the relative position of the load library's name in your GLOBAL LOADLIB command.

The number assigned to the first or only load library is zero.

### Z

A byte of binary zeros.

### C

Indicates whether the information your program put in the NAME field is the member program's name or its alias. It also indicates the length of the user data field in halfwords.

This field is 1-byte long. If bit 0 is reset to 0, it means you are using the member program's name. If bit 0 is set to 1, it means you are using its alias. Bits 1 and 2 are always reset to 0. Bits 3 through 7 contain the number of halfwords in the UD (user data) field.

### UD

This field contains the user data found in the load library associated with the member program. The user data information is used by the loader to relocate the module in storage.

This user data field is always at least 24 halfwords long. By increasing the number in the LL field, you increase the size of the UD field. This allows room for more user data, if necessary.

The byte at displacement 37 (X'25') into the list entry contains three bits that contain RMODE and AMODE information. If bit 3 is on, RMODE=ANY. If bit 6 is on, AMODE=31. If bit 7 is on, AMODE=24.

## Usage

1. The only load libraries that the BLDL macro will consider are those you specify in the GLOBAL LOADLIB command.
2. The BLDL macro will allow no more than 65,535 (64K-1) separate list entries in any single directory entry list, and no fewer than one.
3. If there is more than one list entry in the directory entry list, then it is wise to arrange them alphanumerically according to the NAME field. However, this is not a requirement.
4. Your program is responsible for providing the storage space for the directory entry list. It must also supply the list information and insert the name of each module in its respective list entry.
5. Many programmers find it convenient to use the BLDL macro simply to find out whether a program is really a member of a specific load library. Check the return code and reason code generated by the macro to find this out.

## Messages

When this macro completes processing, it passes to the caller a return code in the low-order byte of register 15. The reason code is returned to the caller in the low-order byte of register 0.

---

<b>Hex Codes</b>	<b>Decimal Codes</b>	<b>Reason Code</b>	<b>Meaning</b>
X'00'	0	00	Function successfully completed.
X'04'	4	00	One or more modules named in the directory entry list could not be found. The R-byte (byte 11) of its TTR field was reset to 0.
X'08'	8	00	A permanent I/O error was found when GCS attempted to search a load library directory.
X'08'	8	04	Insufficient virtual storage space was available for file management.

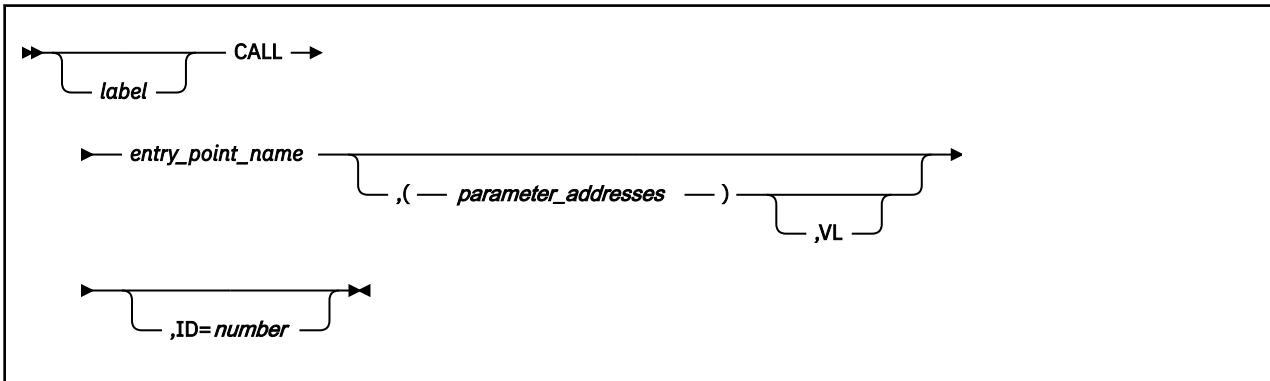
---

## CALL

The CALL macro is available in standard, list and execute formats.

### Standard Format

See also “List Format” on page 185 and “Execute Format” on page 186.



### Purpose

Use the CALL macro to pass control to an entry point. After the CALL is completed, control returns to the point from which it was passed.

### Parameters

#### *entry\_point\_name*

Specifies the name of the entry point that is to receive control.

Because the macro uses this name as a V-type address constant, the linkage editor and loader will have resolved this name into a virtual address.

If you specify a symbol for the entry point name, then the linkage editor will include the control section containing the entry point in question within the load module containing the CALL macro.

You can write this parameter as a symbol or as register (15).

#### *parameter\_addresses*

Specifies a list of one or more parameter addresses that you want to pass to the program at the specified entry point.

The CALL macro gathers these addresses into a parameter list in the order that you listed them in the instruction. The parameter list contains one or more fullwords, each on a fullword boundary and each containing the address of one parameter. The specified entry point receives the address of this parameter list in register 1.

Note that each parameter address in the instruction must be separated by a comma, with the complete list surrounded by parentheses.

You can write these addresses as assembler program labels or as registers (2) through (12).

#### **VL**

Indicates that the program receiving control expects a variable number of parameters to be passed to it. To omit this parameter is to say that the program receiving control expects a set number of parameters.

This parameter sets the high-order bit of the last address parameter in the list to 1, thereby indicating the end of the list.

## ID

A debugging aid for use when you enter several CALL macros.

You can assign this parameter a unique, mnemonic value that will be inserted in any dump you might request. This lets you associate an area within the dump with a specific CALL macro.

You can write this parameter as any number or symbol.

## Usage

1. If you specify the entry point name as register 15, then the load module that contains the entry point must be in virtual storage. Register 15 must contain the address of the entry point.
2. Use of the CALL macro implies that the issuing program ultimately expects to regain control.
3. It is the responsibility of the program issuing the CALL macro to provide storage where the values in its registers may be saved while the other program has control. The address of this save area must be placed in register 13. Also, the called program must save the values in these registers and, later, restore them.
4. The supervisor is not involved in passing control to the entry point. Therefore, it is your task's responsibility to maintain the usability of the program at that entry point. That is, if you modify the program in any way, then you must restore it to its original condition after you have finished. Your task must ensure that only one user has control of the program at a time.

## Examples

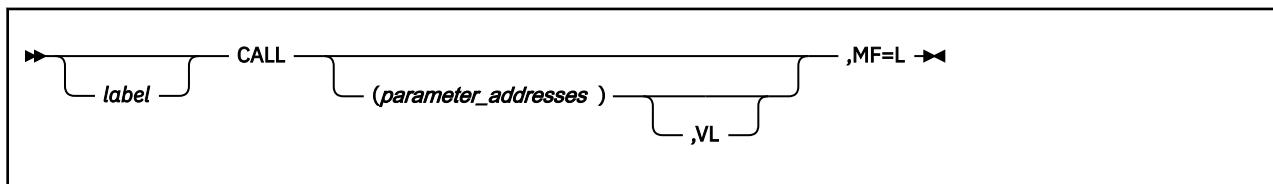
```
CALL (15),(PARAM1,PARAM2),VL
```

The program requests that control be passed to the entry point whose address is in register 15. GCS assumes that the entry point is in virtual storage, (the address specified at register 15). The program being called is to receive two parameter addresses arranged in a parameter list. The addresses of these parameters are associated with the labels PARAM1 and PARAM2. Because the VL parameter is specified, the program being called expects a variable number of parameters be passed to it. In this case, two.

## Return Codes and ABEND Codes

The CALL macro generates no return codes and no ABEND codes.

## List Format



## Purpose (List Format)

This format of the macro generates an in-line parameter list, based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation. Also, note that only the preceding parameters listed are valid in the list format of this macro.

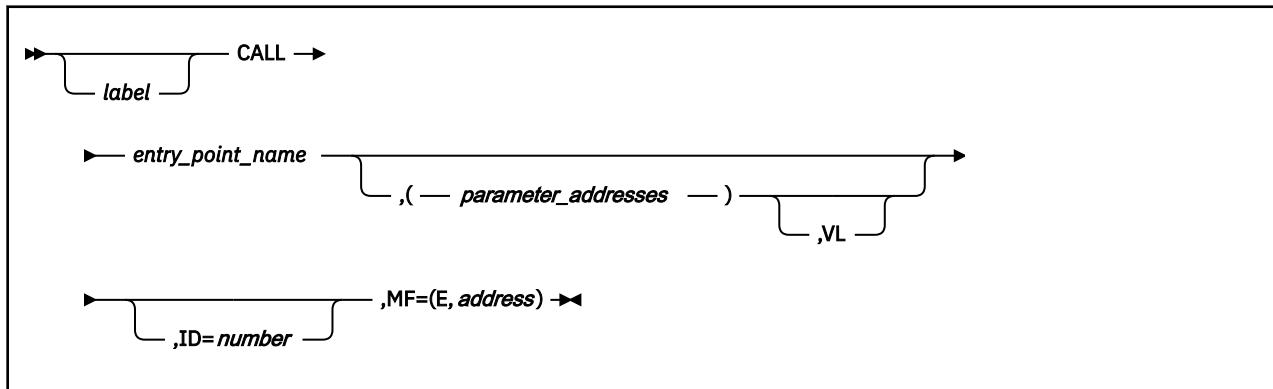
## Added Parameter

## CALL

### MF=L

Specifies the list format of this macro.

### Execute Format



### Purpose (Execute Format)

This format of the macro generates code that executes the function, using a parameter list whose address you specify.

### Added Parameter (Execute Format)

#### MF=(E, address)

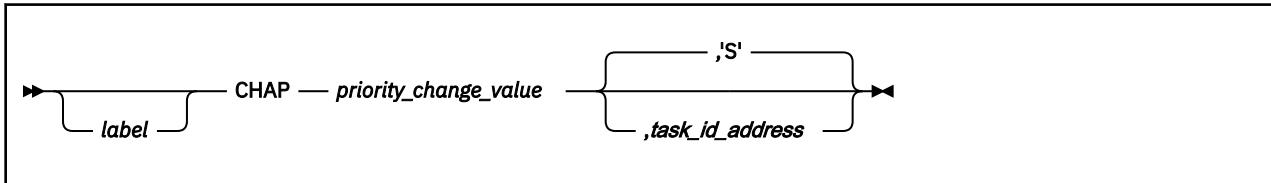
ADDRESS specifies the address of the parameter list to be used by the macro.

You can add or modify values in this parameter list by specifying them in this macro.

# CHAP

---

## Format



## Purpose

Use the CHAP macro to change the dispatching priority of a given task.

GCS is a multitasking system. This means that considerably more than one task can process in the same virtual machine at the same time.

In any multitasking environment some sort of priority system must be established to govern access to the processor by so many tasks. To that end, each task is assigned a dispatching priority number. These numbers determine the order that many competing tasks gain access to the processor.

The dispatching priority for any problem state application task must be a number from 0 to 240. The dispatching priority for any supervisor state application task must be a number from 0 to 250. The larger the number, the higher the dispatching priority of the task and the more readily that task gains access to the processor.

Use the CHAP macro to change the dispatching priority of any application task within your virtual machine.

## Parameters

### *priority\_change\_value*

Specifies a number that is to be added to the current dispatching priority of the task in question. The sum of these two numbers will be the task's new dispatching priority.

To raise the task's dispatching priority, specify a positive number in this parameter. To lower it, specify a negative number.

Should the sum of the two numbers result in a priority less than zero, then the task's new priority will be zero. Should the sum be greater than the highest priority allowed, then the task's new priority will be the highest allowed.

You can write this parameter as any symbol, as a decimal digit, as register (0), or as register (2) through (12). If you write it as a register and wish to specify a negative number, then you must store the number in the register in two's complement form.

### *task\_id\_address*

Specifies the address of a fullword that contains the task identifier of the task in question.

GCS assigned a task ID to your task when you issued the ATTACH macro for it. See “[ATTACH](#)” on page 165. Presumably, you saved the task ID somewhere when the ATTACH macro returned it to you. GCS assumes that the task ID is stored in the 2 low-order bytes at this address. GCS ignores the 2 high-order bytes.

If the address specified in the task ID address parameter equals zero, then GCS assumes that the dispatching priority of the task issuing the CHAP macro is the one to be changed.

You can write this parameter as an RX-type address or as register (1) through (12).

**S**

Indicates that the dispatching priority of the task issuing the CHAP macro is the one to be changed. If you omit both the S and the TASK ID ADDRESS parameters, then GCS treats the macro as though the S parameter were specified.

Note that this parameter must be surrounded by single quotation marks.

**Usage**

1. No task can change the dispatching priority of any other task unless the former issued the ATTACH macro for the latter. Put another way, no task can change the dispatching priority of another task unless the latter is a subtask of the former.
2. You cannot use the CHAP macro to change the priority of the system task.

**Return Codes and ABEND Codes**

The CHAP macro generates no return codes.

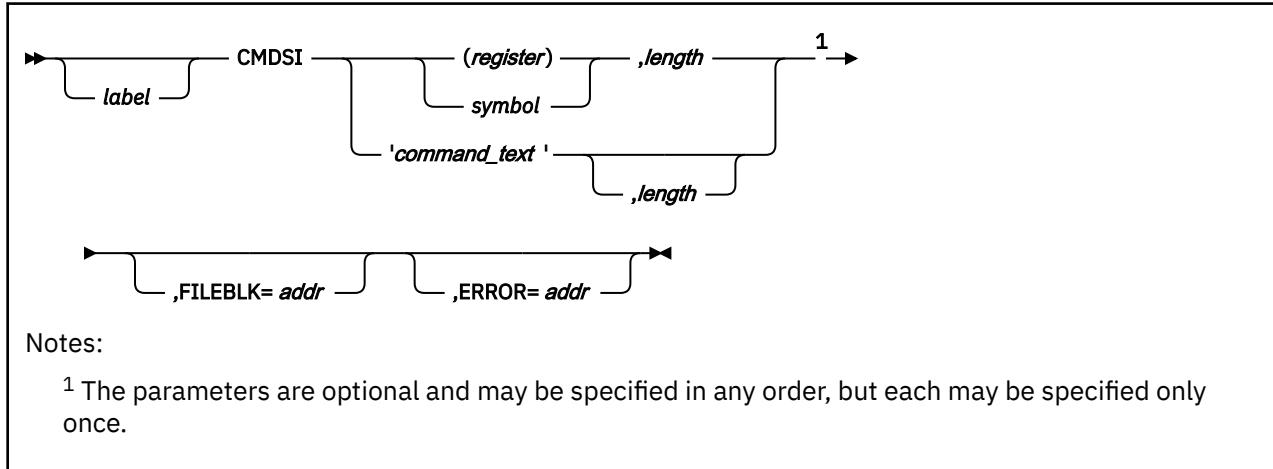
<b>ABEND Code</b>	<b>Meaning</b>
12C	The task ID specified was invalid because: <ul style="list-style-type: none"><li>• It is associated with the system task, not the user task</li><li>• It does not exist</li><li>• It does not refer to one of its immediate descendant tasks</li><li>• The task specified has already terminated.</li></ul>
22C	The address of the parameter list is invalid.

# CMDSI

The CMDSI macro is available in standard, list, list address and execute formats.

## Standard Format

See also “List Format” on page 191, “List Address Format” on page 191 and “Execute Format” on page 192.



## Purpose

Occasionally you will find it necessary to enter a *command* from a program running under GCS. *Command*, in this context, means any command that:

- Ordinarily would be entered directly from the console, this includes GCS commands, CP commands, and EXECs
- Previously defined to GCS using the LOADCMD command. See “[LOADCMD](#)” on page 112.

## Parameters

### **register**

Specify the register containing the address of the command in question. With this method you must specify the LENGTH parameter. Also, the reference to the register must be in parentheses.

### **symbol**

Specify the programming language symbol on the statement containing the command and its options or parameters. Note that you must specify the LENGTH parameter if you use this method.

### **command\_text**

Specify the actual text of the command, with any necessary parameters or options. The entire command statement must be surrounded by single quotation marks.

The LENGTH parameter need not be specified when using this method. If it is, it will be ignored.

### **length**

Specifies the length of the command in bytes. This includes the command itself, its parameters, options, operands, and all imbedded blanks.

It must be a number from 1 to 130.

You can write this parameter as an absolute expression or as register (2) through (12). If you write it as a register, the register must contain the length of the command.

**FILEBLK**

Use this parameter if the:

- REXX/VM interpreter is to process a non-GCS file
- Interpreter is to process from storage
- Address environment is inconsistent with the file type of the file containing the command.

This parameter specifies the address of the file block to be passed to the REXX/VM interpreter, which will interpret the code associated with the command.

This file block contains information necessary to start the code properly. This includes, among other things, the file name, file type, and file mode of the file containing the code; its address (if in storage), and its size. For more information, see [z/VM: REXX/VM User's Guide](#) or [z/VM: REXX/VM Reference](#).

You can write this address as a programming language label or as register (2) through (12).

**ERROR**

Specifies the address of a routine that is to receive control if an error occurs in the CMDSI macro.

**Note:**

1. If you omit this parameter and an error occurs, then control returns to the macro immediately following the CMDSI macro, just as it would were there no error.

You can write this parameter as a programming language label or as register (2) through (12).

2. The error routine does not receive control if an error occurs in the command you are trying to execute.

**Examples**

```
DOIT CMDSI MYCMD,48,FILEBLK=(8),ERROR=(6)
```

Enter the command at the address associated with the label MYCMD. The command is 48 characters long. Because the command calls an EXEC, the address of the file block can be found in register 8. Register 6 contains the address of an error routine that gets control if an error occurs in the CMDSI macro. Presumably, the LOADCMD command has been entered for the command. DOIT is the label on this instruction.

```
INQUIRE CMDSI 'QUERY DISK',ERROR=ERR1
```

Enter the GCS QUERY DISK command. No length is needed because length is implicit in the single quotation marks that surround the command. ERR1 is the label on the error routine that is to receive control if an error occurs in the CMDSI macro. INQUIRE is the label on this instruction.

**Return Codes and ABEND Codes**

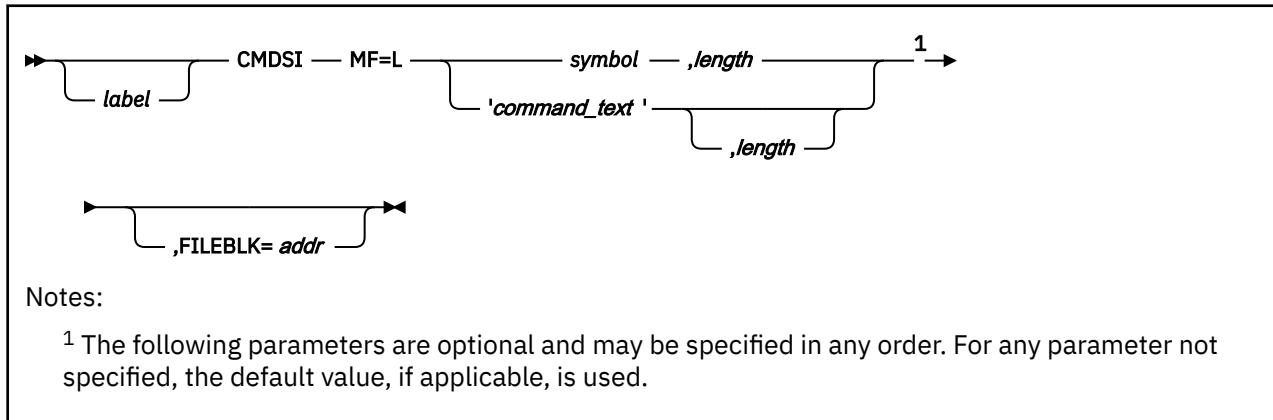
The only return codes generated by the CMDSI macro are defined in the following. They are passed to the caller in register 15. Any other return code passed by the CMDSI macro is really the return code from the command that it called.

Hex Code	Decimal Code	Meaning
X'FFFFFFFD'	-3	The command could not be found.
X'00'	0	The command was successfully executed.

**ABEND Code Reason Code Meaning**

FCA	0300	The parameter list was invalid.
-----	------	---------------------------------

## List Format



## Purpose (List Format)

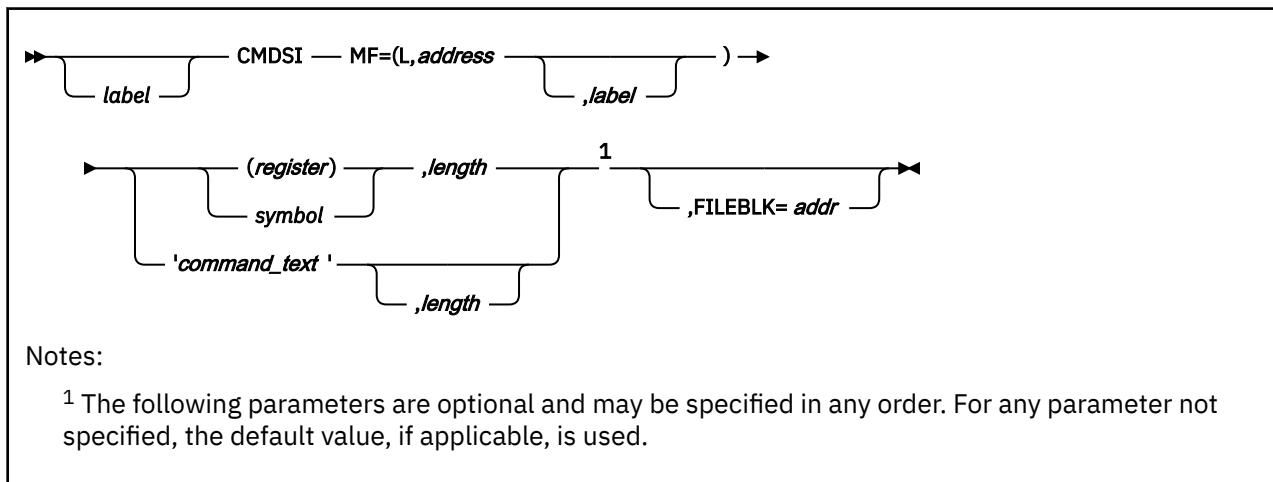
This format of the macro generates an in-line parameter list based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation. Also, note that only the preceding parameters listed are valid in the list format of this macro.

## Added Parameter

### MF=L

Specifies the list format of this macro.

## List Address Format



## Purpose (List Address Format)

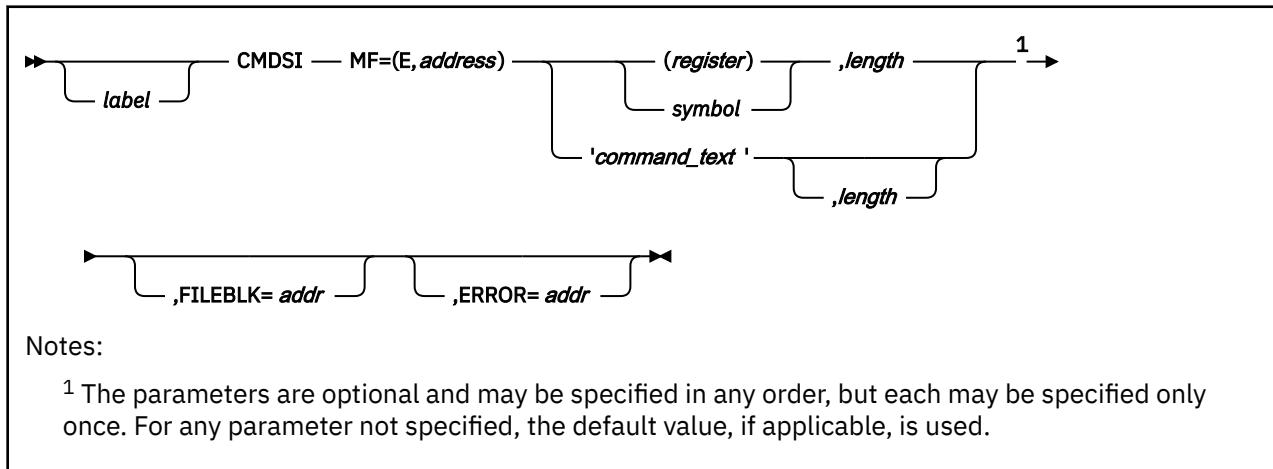
This format of the macro does not produce any executable code that calls the function. However, it does produce executable code that moves the parameter values that you specify into a certain parameter list. If you enter the macro using this format, then you must do so before any related invocation of the macro using the execute format. Also, note that only the preceding parameters listed are valid in the list format of this instruction.

## Added Parameter (List Address Format)

**MF=(L, address, label)**

*address* specifies the address of the parameter list into which you want the parameter values you mention placed. This address can be within your program or somewhere in free storage.

*label* is optional and is a user-specified label, indicating that you want to determine the length of the parameter list. The macro expansion equates the label you specify with the length of the parameter list.

**Execute Format****Purpose (Execute Format)**

This format of the macro generates code that executes the function, using a parameter list whose address you specify.

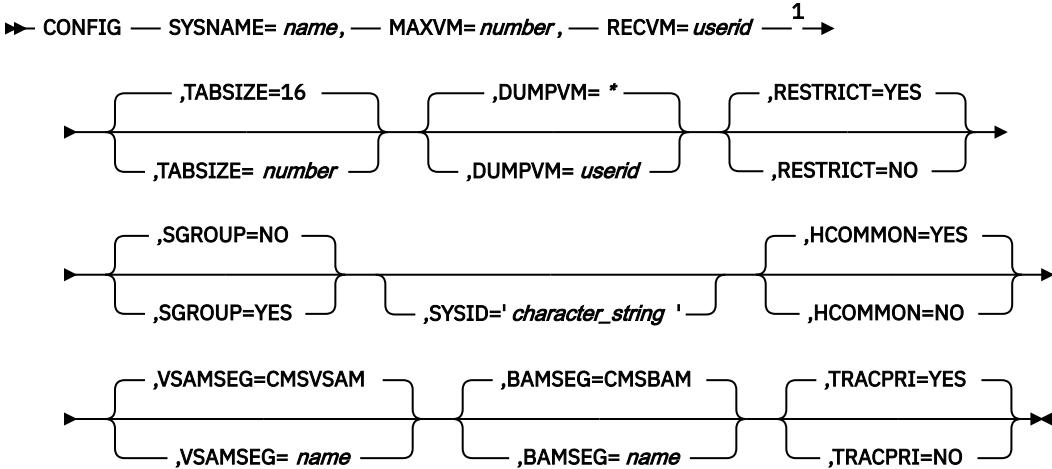
**Added Parameter (Execute Format)****MF=(E, address)**

*address* specifies the address of the parameter list to be used by the macro.

You can add or modify values in this parameter list by specifying them in this instruction.

# CONFIG

## Format



Notes:

<sup>1</sup> The following parameters are optional and may be specified in any order. For any parameter not specified, the default value, if applicable is used.

## Purpose

Use the CONFIG macro to define the configuration of your GCS virtual machine group.

**Note:** The default values listed are used when the GROUP file is assembled. They are not put in the GROUP CONFIGURATION file by the GROUP command.

The GROUP CONFIGURATION FILE describes a GCS virtual machine group. This file is divided into four data blocks:

### Configuration

Defines the virtual machine group's configuration so that it conforms to the needs of your installation.

### Segment

Identifies which saved segments will be automatically linked to each member of the group at IPL time. Review the entry titled “[SEGMENT](#)” on page 338.

### Authorized User

Identifies which members of the group are *authorized*. That is, which members are permitted to perform authorized GCS functions. The Authorized GCS Service Macros are identified in [Chapter 5, “GCS Macros,” on page 157](#).

### Reserve Storage User

Identifies which members of the group are to have storage reserved for the VSAM and BAM segments at IPL time. Also, review the entry titled “[RESSTOR](#)” on page 320.

## Parameters

### SYSNAME

The name under which this system (group) will be saved.

## CONFIG

Every saved system, which is what a virtual machine group is, must have a system name. The name of a GCS virtual machine group must correspond exactly with the file name of the GROUP CONFIGURATION FILE associated with the group. This name must correspond with the name of the saved system as entered in the system name table.

You can write this as any string of alphanumeric characters, eight characters long or less. But, you must be careful not to select a name that could easily be mistaken by the system for a hexadecimal device address, such as C or 595.

### MAXVM

Specifies the maximum number of virtual machines that can join the group whose configuration is being defined.

You can write this parameter as any number from 1 to 65535.

**Note:** If SGROUP is specified as YES (SGROUP=YES), then MAXVM is forced to 1 (MAXVM=1).

### RECVM

Identifies which virtual machine in the group will be designated as the recovery virtual machine.

It is the duty of the recovery virtual machine to perform various clean-up tasks for any virtual machine that is reset. A virtual machine that is *reset* is one that has logged-off, been re-IPLed, and so forth. *Clean-up* includes running any termination exit routines that the reset machine may have identified.

The recovery machine must be the first machine to join the group. Otherwise, an error will occur and the system will be reset.

Write this parameter as the user ID of the recovery machine.

### TABSIZE

Specifies the size of the GCS trace table that the virtual machine group will use.

The GCS supervisor records all supervisor events in this trace table. Users have the option to record user virtual machine events in the same table through the ITRACE command and GTRACE macro. For more information on the ITRACE and GTRACE, see the ["ITRACE" on page 108](#) and ["GTRACE" on page 265](#).

The size of the trace table is expressed in kilobytes. You can write this parameter as any number from 4 to 16384, the default is 16.

### DUMPVM

Specifies the user ID of the virtual machine that is to receive all dumps requested by any member of the group.

It is strongly recommended that the user ID specified by this parameter refer to an authorized user. Remember that dumps often contain fetch-protected, nonkey-14 data. Only authorized users are permitted to handle such data. Therefore, it is wise to have an authorized user designated to handle all dumps so that all types of data can be included.

Write this parameter as the user ID of the virtual machine you wish to designate as the recipient of all dumps.

### RESTRCT

Specifies if the GCS saved system should be restricted to authorized users in the CP directory to IPL this system.

#### YES

The GCS system is saved as restricted. This is the default.

#### NO

The GCS system is not saved as restricted.

**Note:** If SGROUP is specified as YES (SGROUP=YES), then RESTRCT is forced to NO (RESTRCT=NO).

### SGROUP

Specifies you are building a single user group.

**NO**

The GCS system is built with group communication and with shared common storage. This is the default.

**YES**

The GCS system is built without group communication and without shared common storage.

**Note:** If YES is specified (SGROUP=YES), the MAXVM number is forced to one (MAXVM=1), and RESTRCT is forced to NO (RESTRCT=NO).

**SYSID**

Specifies the text of the system identification.

The system ID is a message displayed to each user at IPL time. It can contain any information that the administrators of your system wish. You can write this parameter as any character string of up to 130 characters long.

**Note:**

1. The character string must be surrounded by single quotation marks. If you omit this parameter, then the text of the system ID will be a blank line, by default.
2. If you must include imbedded single quotation marks ('') or ampersands (&) within the SYSID character string, then make certain you include two single quotation marks or two ampersands for every one you intend. Also, be certain that there are no more than 126 of them.

**HCOMMON**

Specified if high common storage is desired.

**YES**

The GCS system is built with free common storage defined above the 16 meg line. This is the default. The start and end of high common can be updated on the GCS LOAD exec (GCTLOAD) if you do not want the default values of 16 meg and 18 meg.

**NO**

The GCS system is built with no common storage above the 16 meg line.

**VSAMSEG**

The name of the VSAM segment that is to be used by the GCS group that you are building a GCS segment. The name can be eight or fewer characters long. The default name is CMSVSAM.

**BAMSEG**

The name of the BAM segment that is to be used by the GCS group that you are building a GCS segment. The name can be eight or fewer characters long. The default name is CMSBAM.

**TRACPRI**

Specifies if the internal trace table is to be in private or common storage.

**YES**

The internal trace table will be in private storage. This is the default. Tracing will be for your virtual machine only.

**NO**

The internal trace table will be in common storage. Tracing will be for all virtual machines in the group.

**Usage**

1. Most installations will not explicitly use the CONFIG macro to build the GROUP CONFIGURATION FILE. Those equipped with at least one full-screen display terminal can take advantage of GCS build panels. These data entry panels, called by the GROUP command, eliminate the need to build the file by explicitly coding these macros. However, without a full-screen terminal, your file will have to be built using the editor and coding the macros manually.
2. The GROUP CONFIGURATION FILE adopts the system name (virtual machine group name) as its file name, the file type is always GROUP.

## **CONFIG**

3. Remember that in using the CONFIG macro you are creating blocks of information. Thus, all occurrences of the CONFIG macro must be physically grouped together in the GROUP CONFIGURATION FILE.

### **Examples**

```
CONFIG SYSNAME=MAIN,MAXVM=5,RECVM=VM1,DUMPVM=VM1,SYSTID='WELCOME!'
```

The name of the system being described is MAIN. No more than five virtual machines can join this group. The virtual machine, whose user ID is VM1, is designated as both the recovery machine and the handler of all dumps. And, the word *WELCOME!* is displayed to each user at IPL time.

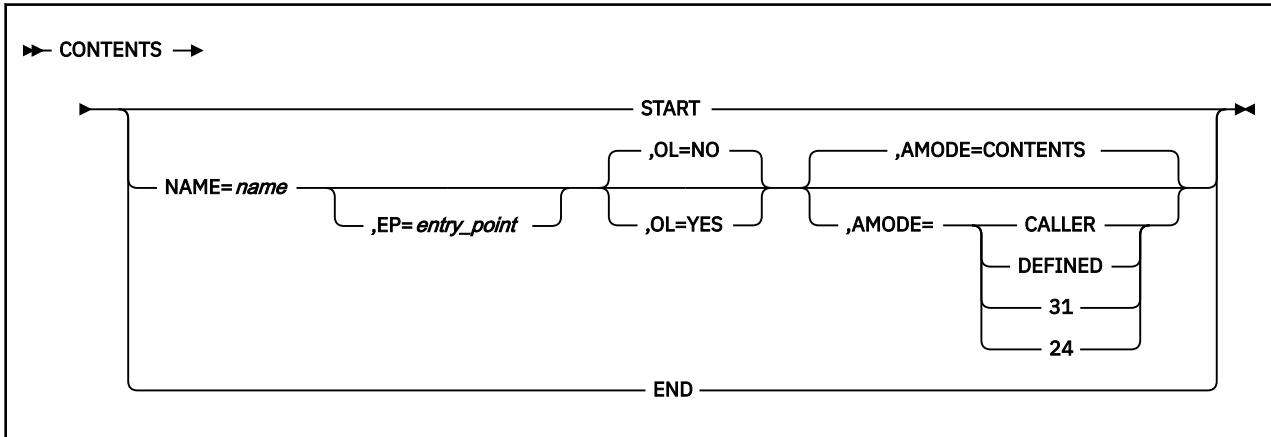
### **Return Codes and ABEND Codes**

The CONFIG macro generates no return codes and no ABEND codes.

# CONTENTS

---

## Format



## Purpose

Use the CONTENTS macro to define the entry points in a saved segment.

For a saved segment to be usable, the various entry points it contains must be defined in a directory. This directory contains the name of each entry point in the saved segment mapped to its address.

Use the CONTENTS macro to create such a directory (also called a CONTENTS MODULE) for a saved segment.

## Parameters

### START

Indicates that this CONTENTS macro marks the beginning of the CONTENTS MODULE. It marks the beginning of the saved segment itself.

### NAME

Specifies the name that an external program can use to pass control to the entry point.

This name is only resolved when the ATTACH, LINK, XCTL, or LOAD macro or the OSRUN command is issued. See “[ATTACH](#)” on page 165, “[LINK](#)” on page 293, “[XCTL](#)” on page 373, “[LOAD](#)” on page 298, or “[OSRUN](#)” on page 116.

This name can be the actual name of the entry point. Or, when the EP parameter is also specified, the name can be an alias. An alias is simply a second name that is associated with the real name of an entry point.

This name must be one of two things, the:

- Label on a CSECT assembler instruction
- Operand symbol in an ENTRY assembler instruction.

### EP

Specifies the actual name of the entry point in the saved segment.

If you specify the real name of the entry point in the NAME parameter, then the EP parameter is unnecessary. However, if you specify an alias for the entry point name in the NAME parameter, then the EP parameter must be specified with the real name of the entry point.

### OL

Indicates whether the code at the entry point in question is only loadable.

## CONTENTS

### YES

Indicates that the code is only loadable. That is, the code is not executable. An example of this would be a data area.

Remember that the LOAD macro is the only macro that can be used on such code. Macros like LINK, XCTL, and ATTACH do not work on code that is only loadable.

### NO

Indicates that the code is not only loadable. That is, the code is executable.

If the OL parameter is omitted, then the code is considered executable, by default.

### AMODE

Specifies the addressing mode.

### CONTENTS

The entry point will be run in the addressing mode specified by the entry point address. Bit 0 of the address determines the mode; if the bit is on AMODE 31 is in effect, otherwise an AMODE of 24 is used.

### CALLER

The entry point will be run in the AMODE of the caller.

### DEFINED

The entry point will be run in the addressing mode specified in bit 0 of the exit address passed to the TASKEXIT, MACHEXIT, AUTHNAME, AUTHCALL, SCHEDEX, LINK, LOAD, and XCTL macros.

### 31

The entry point will be run in AMODE 31.

### 24

The entry point will be run in AMODE 24.

### END

Indicates that this CONTENTS macro marks the end of the CONTENTS MODULE. What follows, then, is the first module in the saved segment.

## Usage

1. Each saved segment must begin with a CONTENTS MODULE. The first example shows the contents of such a module.
2. When the CONTENTS macros in the module are assembled, they expand to associate each entry point name specified with its address in the saved segment.
3. All entry point names in a particular saved segment must be unique. GCS searches multiple CONTENTS MODULES for the first occurrence of a particular entry point name. Therefore, if more than one saved segment (each with its own CONTENTS MODULE) is linked to your virtual machine, then all entry point names in all the saved segments must be unique.
4. When the CONTENTS macro is issued to define an exit, the addressing mode of the exit is determined by the AMODE parameter. (See the TASKEXIT, MACHEXIT, AUTHNAME, AUTHCALL, SCHEDEX, LINK, LOAD, and XCTL macros.)

## Examples

The following code represents a saved segment, consisting of a CONTENTS MODULE and four entry points. These entry points are named PROGRAMA, PROGRAMB, PROGRAMC, and DATA.

```
CONTENTS START
CONTENTS NAME=PROG1,EP=PROGRAMA,AMODE=DEFINED
CONTENTS NAME=PROG2,EP=PROGRAMA
CONTENTS NAME=PROG3,EP=PROGRAMA
CONTENTS NAME=PROGRAMB
CONTENTS NAME=PROGRAMC
CONTENTS NAME=DATA,OL=YES
CONTENTS END
PROGRAMA CSECT
.
```

```
.  
PROGRAMB CSECT  
. .  
PROGRAMC CSECT  
ENTRY DATA  
. .  
DATA DC  
. .
```

The CONTENTS MODULE begins with the CONTENTS macro with the START parameter specified. Then, the entry points are defined.

Of particular interest is the fact that the second through fourth CONTENTS macros have both the NAME and EP parameters specified. PROG1, PROG2, and PROG3 are defined as aliases for PROGRAMA. If an external program calls any of these names, then control will pass to the code at the entry point named PROGRAMA.

The entry points PROGRAMB and PROGRAMC have no alias defined for them. So, to start either of these, an external program would have to use either the name PROGRAMB or PROGRAMC.

The entry point DATA contains code that is ONLY LOADABLE.

The CONTENTS MODULE is concluded with a CONTENTS macro with the END parameter specified.

## Return Codes and ABEND Codes

The CONTENTS macro generates no return codes and no ABEND codes.

## CVT

---

### Format

►► CVT ►►

### Purpose

The CVT macro provides a mapping of the OS communication vector table in your virtual machine group's common storage. Most of the vector table is used by application programs running under GCS.

When using VTAM under GCS, a communication vector table is required.

### Parameters

The CVT macro accepts no parameters.

### Usage

1. The simulated CVT is in common storage following the GCS supervisor code. Only certain CVT fields are supported.
2. Within each member of your virtual machine group, the address of the CVT resides at location X'10'.
3. The following table shows the format of the CVT, as simulated by GCS.

Address	Field
116 (X'74')	CVTDCB -- CVTMVSE
140 (X'8C')	CVTECVT -- Address of ECVT
200 (X'C8')	RESERVED
204 (X'CC')	CVTUSER
248 (X'F8')	CVTSAF
256 (X'100')	RESERVED
304 (X'130')	CVTTZ
328 (X'148')	RESERVED
376 (X'178')	CVTFLAGS
376 (X'178')	CVTFLAG1
377 (X'179')	CVTFLAG2
378 (X'17A')	CVTFLAG3
379 (X'17B')	CVTFLAG4
504 (X'1F8')	RESERVED
576 (X'240')	RESERVED
624 (X'270')	CVTVWAIT
628 (X'274')	CVTEND

4. Bit 0 (CVTMVSE) of the CVTDCB field is the XA mode flag bit.
5. Bit 1 of the CVTDCB field contains 0 when under the GCS supervisor. Under CMS, it contains 1.
6. The CVTUSER field is reserved for the user.
7. The CVTTZ field contains the difference between local time and Coordinated Universal Time (UTC) in binary units of 1.048576 seconds. It is updated by GCS at initialization, and whenever a CP SET TIMEZONE command is issued.

When running in a single user group, only the machine specified as the recovery machine by the GROUP EXEC can receive updates to the CVTTZ field.

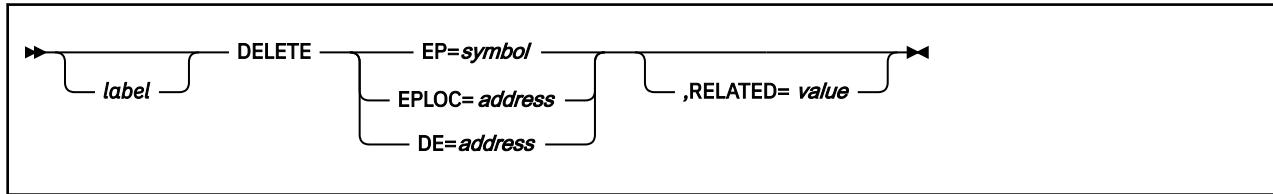
8. The CVTFLAG2 field indicates whether or not Data Compression Services and the hardware instruction (CMPSC) are supported. If Data Compression Services are supported, the CVTCMPSC bit will be on. If the machine supports hardware compression, the CVTCMPSH bit will be on.

## **Return Codes and ABEND Codes**

The CVT macro generates no return codes and no ABEND codes.

## DELETE

### Format



### Purpose

Use the DELETE macro to release control of a load module.

After a task is finished with a load module, that module should be released from the task's control. Generally this will free the storage space that the load module occupies.

In effect, the DELETE macro cancels the effect of the LOAD macro. See “[LOAD](#)” on page 298. Use the DELETE macro to release your task's control over a load module and, if it is no longer needed, to remove it from virtual storage.

### Parameters

#### EP

Specifies the name of the entry point contained in the load module.

This is the module you no longer wish to control.

You can write this parameter as any valid symbol.

#### EPLOC

Specifies the address in your program where you have stored the name of the load module's entry point.

This name may be up to 8 bytes long. If it is less than 8 bytes long, then it must be padded on the right with blanks.

You can write this parameter as an RX-type address, as register (0), or as register (2) through (12).

#### DE

Specifies the address of the NAME FIELD within the directory list entry associated with the entry point in question.

This is the same list entry you placed in the directory using the BLDL macro. See “[BLDL](#)” on page 181.

You can write this parameter as an RX-type address, as register (0), or as register (2) through (12).

#### RELATED

Specifies documentation data that you are using to relate this macro to a LOAD macro.

The value you assign to this parameter has nothing to do with the execution of the macro itself. It merely relates one macro (DELETE) to a macro that provides an opposite, though related, service (LOAD).

The format and contents of this parameter are at your discretion, and can be any valid coding value.

## Usage

1. The DELETE macro frees the storage occupied by the load module only if it resides in private storage and if the module is no longer needed.
2. The task that issues the DELETE macro to release a given load module must be the same task that issued the LOAD macro.
3. Modules loaded with the ADDR parameter cannot be deleted.

## Examples

```
LOADIT  LOAD EP=XYZ,RELATED=DELETEIT
:
.
DELETETIT DELETE EP=XYZ,RELATED=LOADIT
```

The task relinquishes control over the load module containing the entry point XYZ. This DELETE macro is cross-referenced with a related LOAD macro by use of the RELATED parameters in each.

## Return Codes and ABEND Codes

When this macro completes processing, it passes to the caller a return code in register 15.

---

Hex Code	Decimal Code	Meaning
X'00'	0	Function successfully completed.
X'04'	4	Either your task did not issue a corresponding LOAD macro, or the load module has already been deleted.

---

ABEND Code	Meaning
206	Invalid parameter list.

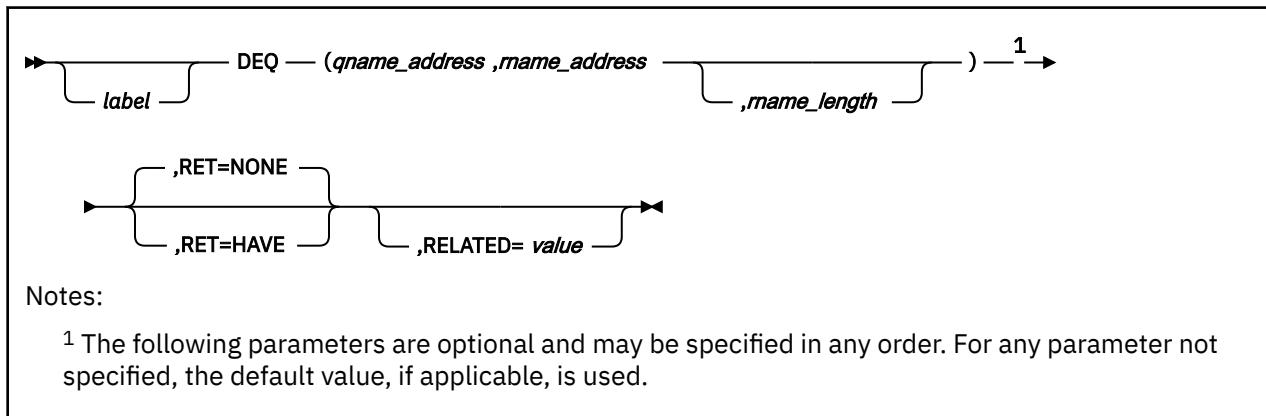
---

## DEQ

The DEQ macro is available in standard, list and execute formats.

### Standard Format

See also [“List Format” on page 206](#) and [“Execute Format” on page 207](#).



### Purpose

Use the DEQ macro to release control of a serially reusable resource.

A serially reusable resource (SRR) is a data resource that some tasks may want to update and that others may only want to examine. These SRRs should be coordinated carefully. Two programs may seek to update the resource simultaneously, leading to incorrect results. Meanwhile, another program may be looking at the same data, causing more confusion.

Use the ENQ macro to gain exclusive use of a serially reusable resource so it can be updated. No other task can touch the resource until the task that has exclusive control releases it. If an SRR is not being updated, but only looked at, several tasks can also share the resource using the ENQ macro. But they cannot alter the contents of the resource in any way. See [“ENQ” on page 213](#).

Use the DEQ macro to release your task's control of a serially reusable resource.

### Parameters

#### *qname\_address*

Specifies the address in virtual storage where the QNAME for the resource in question can be found.

The QNAME is the first of a pair of names that identifies the resource. It can be up to 8 bytes long and can contain any valid hexadecimal characters. Your installation has defined the QNAMEs of each serially reusable resource available to you. Each programmer is required to use the proper QNAME to identify an SRR.

You can write this parameter as an assembler program label or as register (2) through (12).

#### *rname\_address*

Specifies the address in virtual storage where the RNAME of the resource can be found.

The RNAME is the second of a pair of names that identifies the resource. Again, your installation has defined these and they must be used consistently. The name can be qualified and must be from 1 to 255 characters long.

You can write this parameter as an assembler program label or as register (2) through (12).

***rname\_length***

Specifies the length of the RNAME, in bytes.

It must be the same value as the RNAME LENGTH specified in the ENQ macro that gave the task control of the resource.

If you omit this parameter, then the RNAME is considered, by default, to be its assembled length. If you wish, you can override its assembled length with another within the range 1 through 255. If you specify 0 as the length, then the ENQ macro assumes that the first byte at the address specified for the RNAME ADDRESS contains the RNAME's correct length.

You must specify this parameter if there is no length associated with the RNAME itself. For example, you may specify the RNAME by using a register or by using a name appearing in an EQU assembler instruction.

You can write this parameter as a number between 0 and 255.

**RET**

Indicates the condition under which your request will be honored. If you omit this parameter, then your request will be considered unconditional.

**HAVE**

Indicates that the resource is to be released from your task's control only if the task has control of it at the moment.

**NONE**

Indicates that the request to release the resource from your task's control is unconditional.

**RELATED**

Specifies documentation data that you are using to relate this macro to an ENQ macro. The value you assign to this parameter has nothing to do with the execution of the macro itself. It merely relates one macro (DEQ) to a macro that provides an opposite, though related, service (ENQ).

The format and contents of this parameter are at your discretion and can be any valid coding values.

**Usage**

1. Control of a resource is surrendered when the task with control:

- Issues the DEQ macro
- Terminates abnormally, because it did not release the SRR itself.

2. If you choose the NONE parameter and your task does not have control of the resource, your task will terminate abnormally. It is important to find out if your task really does have control of the resource before using the NONE parameter, or simply use the HAVE parameter.

**Examples**

```
LETGO DEQ (PATH, (8),16),RET=NONE
```

A task is releasing a certain resource from its control. The QNAME of the resource can be found at the address associated with the label PATH. Its RNAME can be found at the address in register 8. Because the RNAME was specified by a register, the RNAME LENGTH was also specified as 16. The request is unconditional, so presumably the task tested to see if it had control of the resource before it issued the request. LETGO is the label on this instruction.

```
DEQ ((3),RN),RET=HAVE
```

A task is releasing a certain resource from its control. The QNAME of the resource can be found at the address in register 3. Its RNAME can be found at the address associated with the label RN. The length of the RNAME is not specified and will, therefore, be the assembled length of RN, by default. This request will be honored only if the resource is under the task's control at the moment.

## Return Codes and ABEND Codes

If register 15 contains the value zero, then the resource in question has been released. If register 15 does not contain 0, then it contains the address of the input parameter list of the macro. The DEQ macro places all nonzero return codes in byte 3 of the input parameter list.

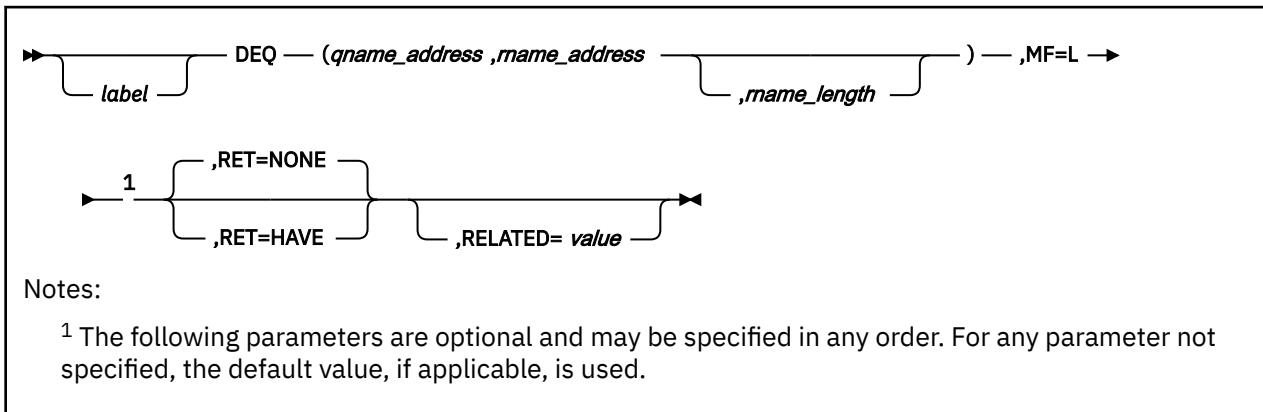
The return codes and abend codes are described as follows, according to the condition specified in the RET parameter.

### When RET=HAVE:

Hex Code	Decimal Code	Meaning
X'00'	0	The resource specified has been released.
X'04'	4	Your task requested control of the resource but has not yet received it. This return code results if a DEQ macro is issued within an exit routine that received control because of some interrupt.
X'08'	8	Either your task never had control of the specified resource or it already released control.

ABEND Code	Meaning
130	The resource was not previously specified in an ENQ macro. Nor was the RET=HAVE parameter specified in that instruction.
230	An invalid length was specified for the RNAME LENGTH parameter.
430	Invalid parameter list.
530	A task issued the ENQ instruction. Before the request could be honored, the same task issued the DEQ instruction without the HAVE parameter specified.
E30	Either your task attempted to make multiple requests with one DEQ instruction, or a parameter that is not supported by GCS was specified with the instruction.

## List Format



## Purpose (List Format)

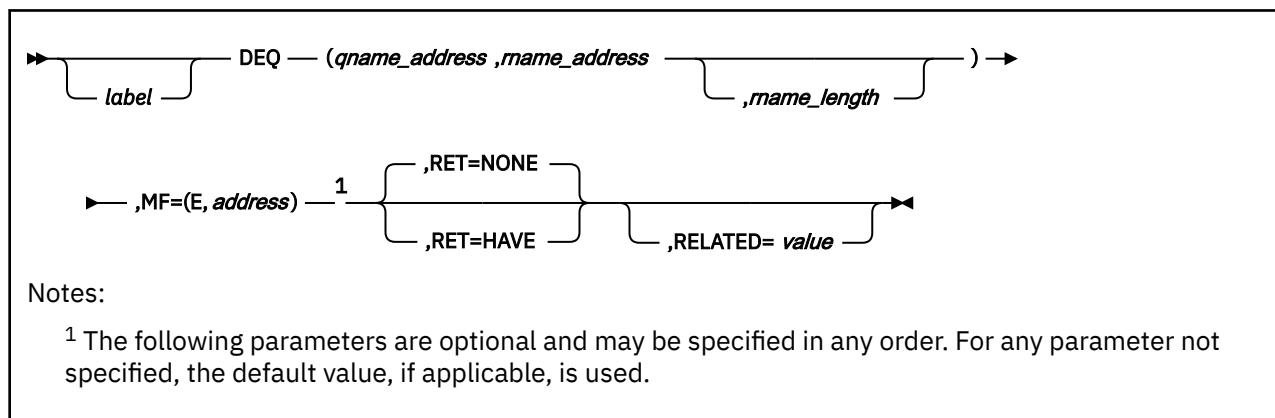
This format of the macro generates an in-line parameter list based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation.

## Added Parameter

### MF=L

Specifies the list format of this macro.

## Execute Format



## Purpose (Execute Format)

This format of the macro generates code that runs the function, using a parameter list whose address you specify.

## Added Parameter (Execute Format)

### MF=(E, address)

address specifies the address of the parameter list to be used by the macro.

You can add or modify values in this parameter list by specifying them in this macro.

# DETACH

## Format



## Purpose

Use the DETACH macro to remove a subtask from your virtual storage.

When you no longer have any use for a subtask for which you issued the ATTACH macro, it should be removed from storage.

Use the DETACH macro to remove a subtask and its task block from storage and to break the logical link between it and its immediate ancestor task.

## Parameters

### *task\_id\_address*

Specifies the address of a fullword that contains the task identifier of the subtask in question.

GCS assigned a task ID to your subtask when you issued the ATTACH macro for it. (If necessary, review the entry titled “[ATTACH](#)” on page 165.) Presumably, you saved the task ID somewhere when the ATTACH macro returned it to you. GCS assumes that the task ID is stored in the 2 low-order bytes at this address. GCS ignores the 2 high-order bytes.

You can write this parameter as an RX-type address or as register (1) through (12).

## Usage

1. The task that issues the DETACH macro for a particular subtask must be the one that issued the ATTACH macro for it first.
2. If a DETACH macro is issued for a subtask that is in mid-execution, then the latter is terminated abnormally. Should the subtask in question have any descendant subtasks of its own, they are also terminated abnormally. If you specified an exit routine for the subtask through the ESTAE macro, then the former is not executed. (If necessary, review the entry titled “[ESTAE](#)” on page 223.) Nor is the routine specified by the ETXR parameter in the ATTACH instruction executed. However, if you specified an event control block (ECB) in the ATTACH macro associated with the subtask, then that ECB is posted. Finally, control is returned to the instruction immediately following the DETACH instruction.

## Return Codes and ABEND Codes

When this macro completes processing, it passes to the caller a return code in register 15.

Hex Code	Decimal Code	Meaning
X'00'	0	Function completed normally.
ABEND Code	Meaning	
13E	This subtask was detached in mid-execution. Therefore, it has terminated abnormally.	

<b>ABEND Code</b>	<b>Meaning</b>
23E	The address of the task ID was invalid.
43E	The ECB address specified in the corresponding ATTACH instruction was invalid.
705	An uncorrectable machine, system, or indeterminate error occurred when GCS issued the FREEMAIN macro.

## DEVTYPE

---

### Format



### Purpose

Use the DEVTYPE macro to request information relating to the characteristics of an I/O device, and to cause this information to be placed into a specified area.

### Parameters

#### *dd\_address*

Specifies the address an 8-byte field that contains the name of the DD statement to which the device is assigned. The name must be left justified in the 8-byte field and must be followed by blanks if the name is fewer than eight characters.

#### *area\_address*

Specifies the address of an area into which the device information is to be placed. The amount of device information returned is dependent on the device type and the usage of the DEVTAB and RPS parameters.

#### DEVTAB

This parameter is only required for direct access devices. If DEVTAB is specified, the following number of words of information is placed in your area: For DASD, 5 words, and for non-DASD, 2 words.

If you do not code DEVTAB, 2 words of information are placed in your area.

#### RPS

If RPS is specified, then DEVTAB must also be specified. The RPS parameter causes an additional word of RPS information to be included with the DEVTAB information.

### Usage

To map this information from the DEVTYPE macro use the IHADVA macro. For more information on the IHADVA macro see "["IHADVA"](#) on page 272.

### Return Codes and ABEND Codes

Control is returned to your program at the next executable instruction following the DEVTYPE macro. Register 15 contains a return code from the DEVTYPE macro. The DEVTYPE macro generates the following return code.

Hex Code	Decimal Code	Meaning
X'00'	0	The information has been successfully stored.
X'04'	4	The DDNAME you specified was not found.

The following are ABEND Codes from DEVTYPE.

<b>ABEND Code</b>	<b>Reason Code</b>	<b>Meaning</b>
118	01	The ddname, whose address was supplied in register 1, is not a valid address.
118	02	The output area, whose address was supplied in register 0, is not a valid user address.

## ECVT

### Format

```
►► ECVT ►►
```

### Purpose

The ECVT macro provides a mapping of the OS extended communication vector table in your virtual machine group's common storage.

### Parameters

The ECVT macro accepts no parameters.

### Usage

1. The system compression routine is found through the CVT and ECVT tables, once the CSRCMPSC macro is invoked. It then branches to the correct service entry point to do the data compression or expansion.
2. The following table shows the format of the ECVT, as simulated by GCS.

Address	Field
0 (X'00')	ECVTECVT -- Table eyecatcher 'ECVT'
4 (X'04')	RESERVED
240 (X'F0')	RESERVED
244 (X'F4')	RESERVED
248 (X'F8')	ECVTCMPS -- Address of Data Compression Services routine
252 (X'FC')	RESERVED

### Return Codes and ABEND Codes

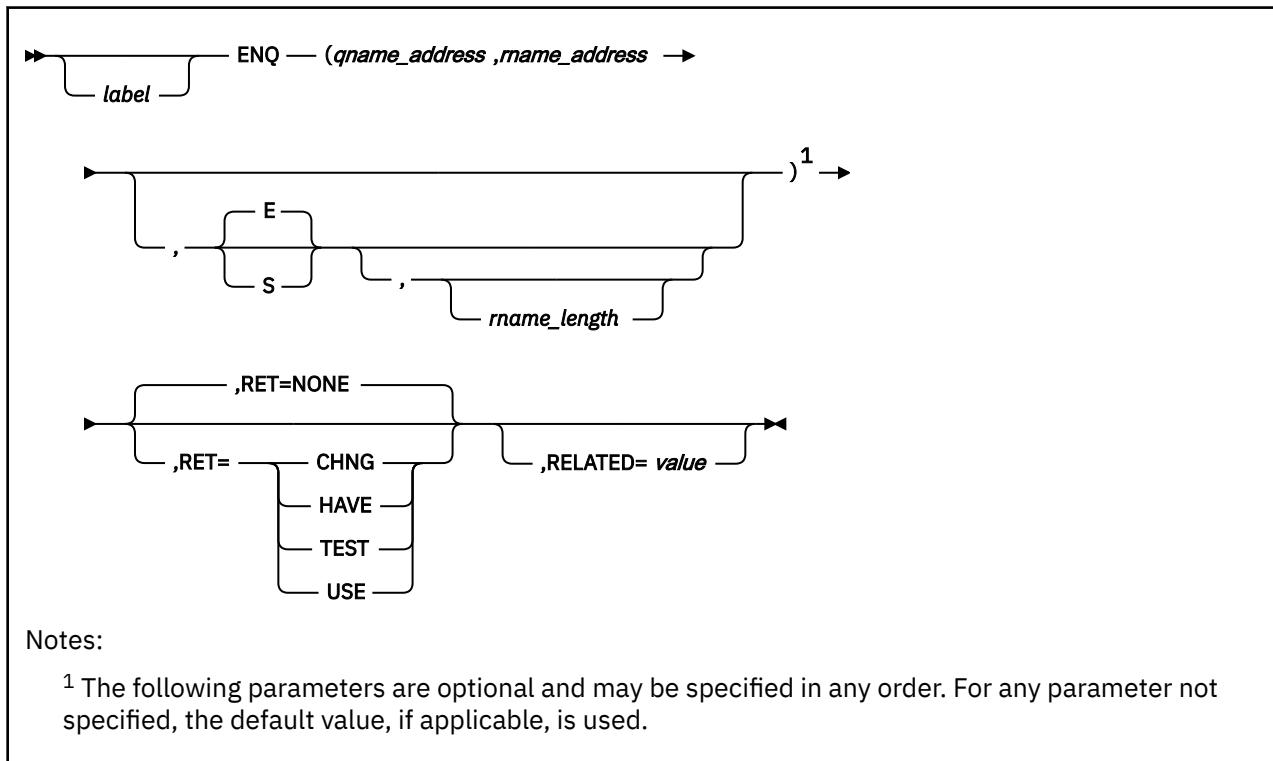
The ECVT macro generates no return codes and no ABEND codes.

## ENQ

The ENQ macro is available in standard, list and execute formats.

### Standard Format

See also “List Format” on page 217 and “Execute Format” on page 217.



### Purpose

Use the ENQ macro to request control of a serially reusable resource.

A serially reusable resource (SRR) is a data resource, local to a virtual machine, that some tasks may want to update and that others may want merely to examine. Use of these SRRs should be coordinated carefully. Two programs may seek to update the resource simultaneously, leading to incorrect results. Meanwhile, another program may be looking at the same data, causing more confusion.

Use the ENQ macro to request control of a serially reusable resource and to define the control sought by your task.

### Parameters

#### *qname\_address*

Specifies the address in virtual storage where the QNAME for the resource in question can be found.

The QNAME is the first of a pair of names that identifies the resource, and must be eight characters long. Your installation has defined the QNAMEs of each serially reusable resource available to you. Each programmer is required to use the proper QNAME to identify an SRR.

You can write this parameter as an assembler program label or as register (2) through (12).

***rname\_address***

Specifies the address in virtual storage where the RNAME of the resource can be found.

The RNAME is the second of a pair of names that identifies the resource. Again, your installation has defined these and they must be used consistently. The name can be qualified and be from 1 to 255 characters long.

You can write this parameter as an assembler program label or as register (2) through (12).

**E**

Indicates that you want your task to have exclusive control over the serially reusable resource. That is, while your task has control over the resource, no other task can use it.

You must request exclusive control if your task is to modify the serially reusable resource in any way.

**S**

Indicates that your task can share control of the resource with other tasks that are also willing to share.

If two or more tasks are sharing a serially reusable resource, then none is permitted to change the contents of that resource.

***rname\_length***

Specifies the length of the RNAME, in bytes.

If you omit this parameter, then the RNAME is considered by default to be the assembled length. If you wish, you may override its assembled length with another within the range 1 through 255. If you specify 0 as the length, then the ENQ macro assumes that the first byte at the address specified for the RNAME ADDRESS contains the RNAME's correct length.

You must specify this parameter if there is no length associated with the RNAME itself. For example, you may specify the RNAME by using a register or by using a name appearing in an EQU assembler instruction to specify the RNAME.

You can write this parameter as a number from 0 to 255.

**RET**

Indicates the condition under which your request for control of the resource will be honored. If you omit this parameter, then the request is considered unconditional.

**TEST**

Tests the availability of the resource specified. It does not turn control of the resource over to your task.

**CHNG**

Indicates that the shared control your task now has over the resource is to change to exclusive control.

This request will be honored if no other tasks are sharing the same resource with your task.

**HAVE**

Indicates that your task wants control of the resource only if it has not requested control of it before.

**NONE**

Indicates that your task requests control of the resource unconditionally.

Your task will not regain control until it obtains control of the resource.

**USE**

Indicates that your task wants immediate control over the resource. If control of the resource is not immediately available, then your task foregoes control and does not wait.

**RELATED**

Specifies documentation data that you are using to relate this macro to a DEQ macro.

The value you assign to this parameter has nothing to do with the execution of the macro itself. It merely relates one macro (ENQ) to a macro that provides an opposite, though related, service (DEQ).

The format and content of this parameter are at your discretion and may be any valid coding values.

## Usage

1. Control of a resource is surrendered under when:
  - A program within the task with control issues the DEQ macro. Review the entry titled “[DEQ](#)” on page [204](#).
  - The task with control ends. The task terminates abnormally, because it did not release the resource itself.
2. After it issues the ENQ macro, your task may be placed in the WAIT state when requesting:
  - Exclusive unconditional control of a resource that is under exclusive or shared control of another task.
  - Control of a resource that is under the exclusive control of another task.
  - Shared control but there is a request for exclusive control ahead of it.
3. The ENQ macro affects only the tasks within the virtual machine which issued it. Tasks in other virtual machines are not constrained from using the serially reusable resource to which the instruction refers. The programmers involved should take steps to assure that this does not create problems.
4. If you choose the TEST parameter, then your task is not given control of the task but merely receives a return code. The same may be true if you choose the HAVE or USE parameter. Return codes are defined in “[Return Codes and ABEND Codes](#)” on page [215](#).

## Examples

```
GETIT ENQ (PATH,(4),E,32)
```

The task is requesting exclusive, unconditional control over a certain serially reusable resource. The resource's QNAME can be found at the address associated with the assembler program label PATH. The RNAME can be found at the address in register 4. Because a register was specified for the RNAME, the length of the RNAME is also specified, as 32. GETIT is the label on this instruction.

```
ENQ ((3),RN,S),RET=USE
```

The task is requesting immediate, shared control of a resource. If that resource is not immediately available, the task does not wish to wait. The QNAME can be found at the address in register 3. The RNAME can be found at the address associated with the label RN. The length of the RNAME will be the assembled length of RN, by default.

## Return Codes and ABEND Codes

A return code is passed to your task only if you choose the TEST, USE, CHNG, or HAVE conditions for the RET parameter.

If register 15 contains 0, then the return code for the resource in question is 0. If register 15 does not contain 0, then it contains the address of the input parameter list of the macro. The ENQ macro places all nonzero return codes in byte-3 of the input parameter list.

For all 08 return codes (except when RET=CHNG), you must examine the fourth bit in byte-0 of the input parameter list. If this bit is reset to 0, then the return code means that the task has obtained exclusive control of the resource. If this bit is set to 1, then the return code means that the task has obtained shared control.

The return codes and abend codes are described as follows, according to the condition specified in the RET parameter. **When RET=CHNG:**

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'00'	0	The task now has exclusive control of the resource.
X'04'	4	The task cannot get exclusive control of the resource.
X'08'	8	The resource has not been queued.
X'14'	20	A previous request for control of the same resource was made by this task. The task does not have control of the resource.

**When RET=HAVE:**

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'00'	0	Control of the resource has been given to the task.
X'08'	8	The task has control of this resource by virtue of a previous request. If bit 3 of the first byte in the parameter list is set to 1, then this task has shared control of the resource. If bit 3 is reset to 0, then this task has exclusive control.
X'14'	20	The task has made a previous request for control of this resource. The task is not given control of the resource.

**When RET=TEST:**

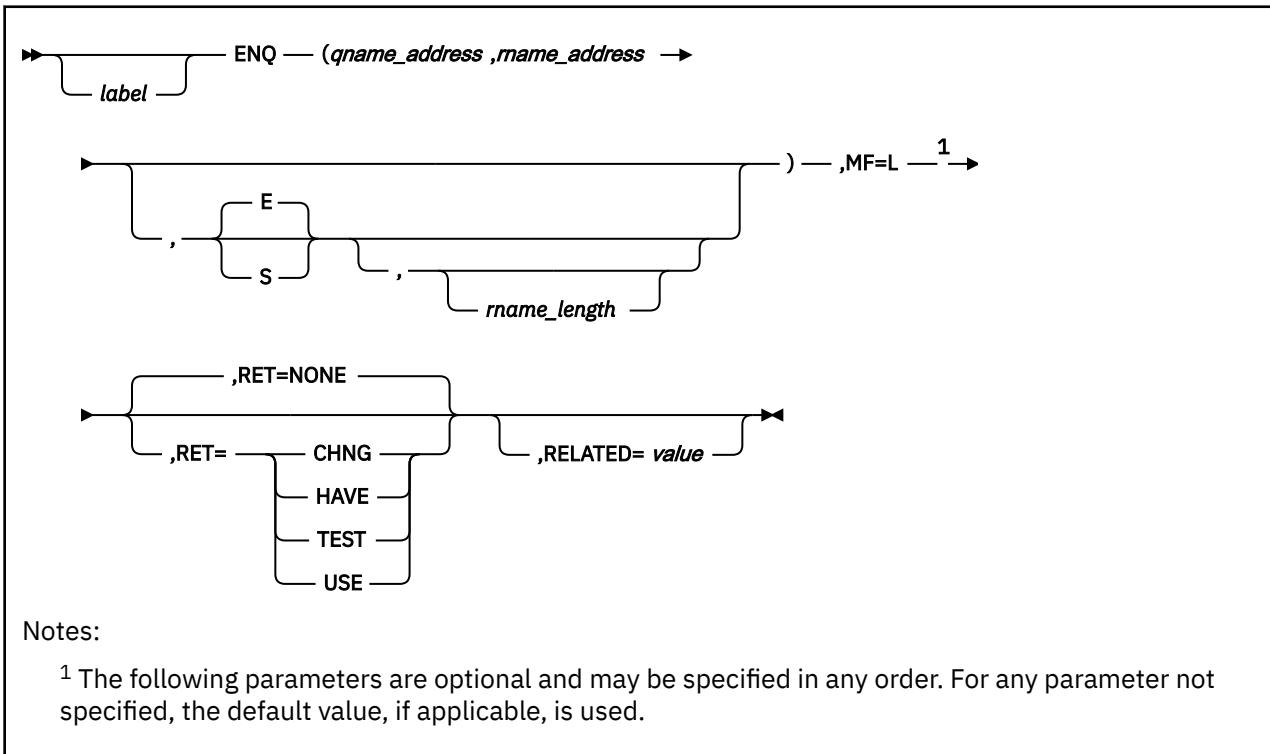
<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'00'	0	The resource is available immediately.
X'04'	4	The resource is not available immediately.
X'08'	8	The task has control of this resource by virtue of a previous request. If bit 3 of the first byte in the parameter list is set to 1, then this task has shared control of the resource. If bit 3 is reset to 0, then this task has exclusive control.
X'14'	20	The task has made a previous request for control of this resource. The task is not given control.

**When RET=USE:**

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'00'	0	Control of the resource has been given to the task.
X'04'	4	The resource is not available immediately.
X'08'	8	The task has control of this resource by virtue of a previous request. If bit 3 of the first byte in the parameter list is set to 1, then this task has shared control of the resource. If bit 3 is reset to 0, then this task has exclusive control.
X'14'	20	The task has made a previous request for control of this resource. The task is not given control.

ABEND Code	Meaning
138	Two ENQ instructions were issued for the same resource by the same task without an intervening DEQ instruction.
238	An invalid length was specified for the RNAME LENGTH parameter.
438	Invalid parameter list.
638	Insufficient storage was available to fulfill your request.
E38	Either your task attempted to make multiple requests with one ENQ instruction, or a parameter that is not supported by GCS was specified in the instruction.

### List Format



### Purpose (List Format)

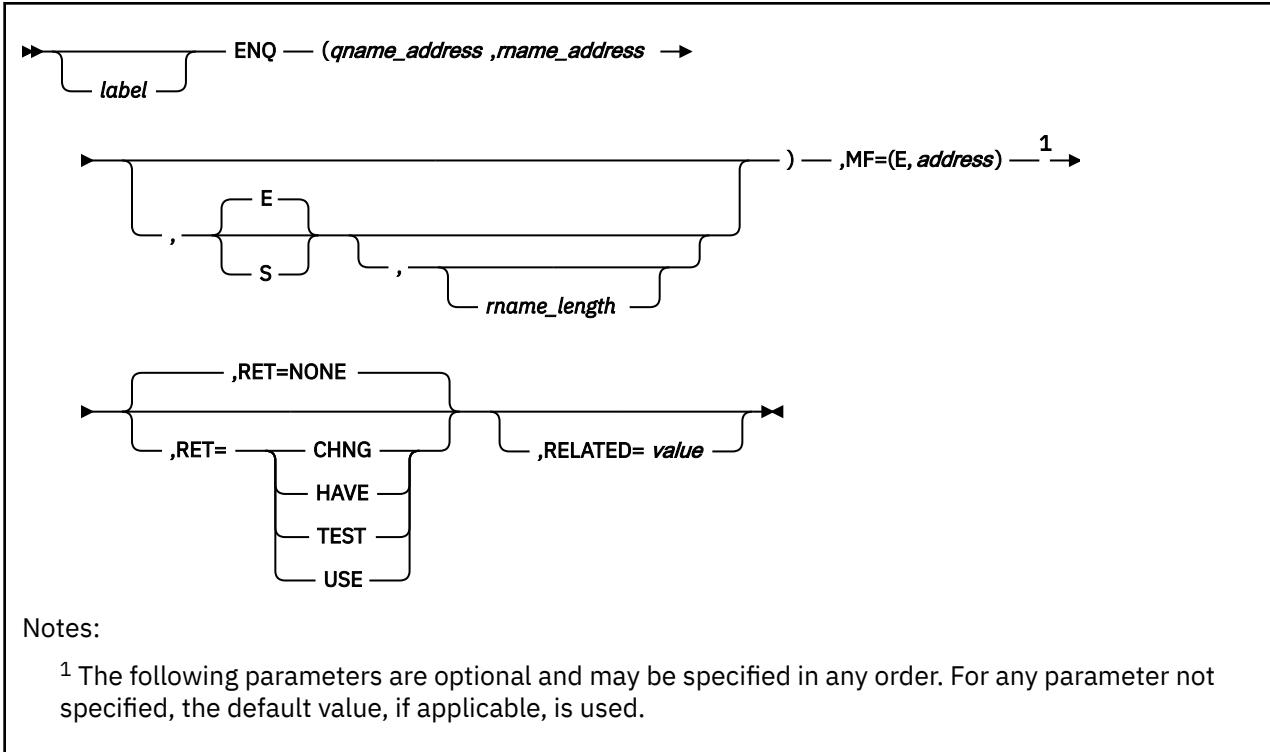
This format of the macro generates an in-line parameter list based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation.

### Added Parameter

#### MF=L

Specifies the list format of this macro.

### Execute Format



## Purpose (Execute Format)

This format of the macro generates code that executes the function, using a parameter list whose address you specify.

## Added Parameter (Execute Format)

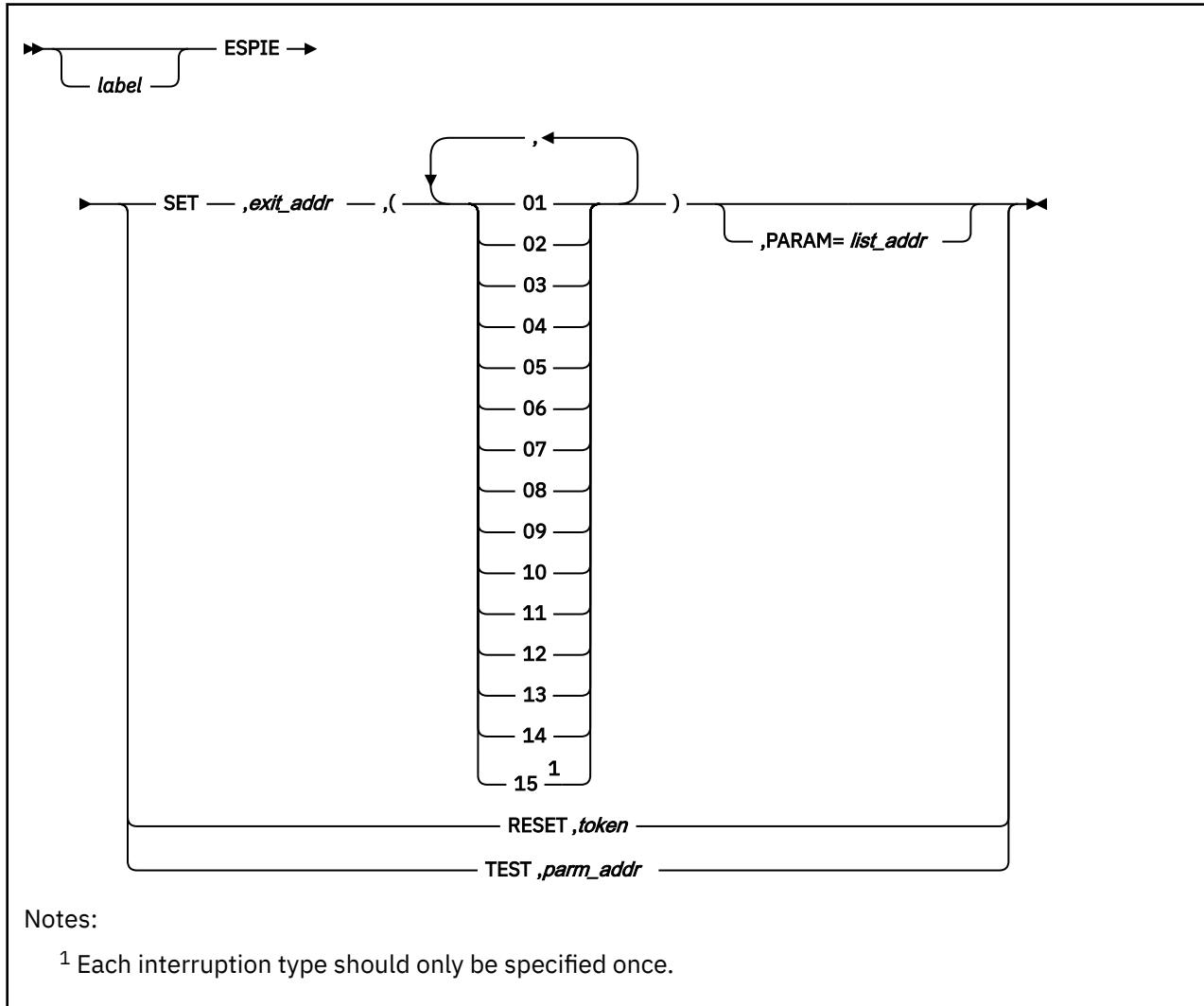
### MF=(E, address)

address specifies the address of the parameter list to be used by the macro.

You can add or modify values in this parameter list by specifying them in this macro.

# ESPIE

## Format



## Purpose

Use the ESPIE macro to specify the address of an interruption exit routine and the program interruptions types that are to cause the exit routine to be given control. If the program interruption type specified can be masked, the corresponding program mask bit in the PSW is set to one. If a maskable interruption is not specified, the corresponding bit in the PSW is set to 0.

## Parameters

### SET

indicates that an ESPIE environment is to be established.

### exit\_addr

specifies the address of the exit routine to be given control when program interruptions of the type specified by 'interruptions' occur. The exit routine receives control in the same addressing mode as the issuer of the ESPIE macro.

***interruptions***

indicates the interruption types that are being trapped. The interruption types are:

- 01** Operation
- 02** Privileged operation
- 03** Execute
- 04** Protection
- 05** Addressing
- 06** Specification
- 07** Data
- 08** Fixed-point overflow (maskable)
- 09** Fixed-point divide
- 10** Decimal overflow (maskable)
- 11** Decimal divide
- 12** Exponent overflow
- 13** Exponent underflow (maskable)
- 14** Significance (maskable)
- 15** Floating-point divide

**PARAM=list\_addr**

specifies the fullword address of a parameter list that is to be passed by the caller to the exit routine.

**RESET**

indicates that the current ESPIE environment is to be deleted and the previously active ESPIE environment specified by *token* is to be re-established.

***token***

specifies a fullword that contains a token representing the previously active ESPIE environment. This is the same token that ESPIE processing returned to the caller when the ESPIE environment was established using the SET option of the ESPIE macro.

**TEST**

indicates a request for information concerning the active or current ESPIE environment. EPSIE processing returns this information to the caller in a four-word parameter list located at *parm\_addr*.

**parm\_addr**

specifies the address of a four-word parameter list.

The parameter list has the following form:

**Word 0**

Address of the user-exit routine.

**Word 1**

Address of the user-defined parameter list

**Word 2**

Mask of program interruption types

**Word 3**

Zero

## Results

1. The program issuing the ESPIE SET macro, receives the following information in its registers.

Register	Contents
01	Token representing the previously active ESPIE environment.
15	Return code 0

2. The program issuing the ESPIE RESET macro, receives the following information in its registers.

Register	Contents
01	Token identifying the new ESPIE environment.
15	Return code 0

3. The program issuing the ESPIE TEST macro, receives the following information in register 15.

Return Codes	Description
0	An ESPIE exit is active and the parameter list is complete.
8	An ESPIE exit is not active.

## The Extended Program Interruption Element (EPIE)

The control program creates an EPIE the first time you enter an ESPIE macro during the performance of a task or whenever you enter an ESPIE macro and no EPIE exists. The EPIE is freed when you eliminate the ESPIE environment.

The EPIE contains the information that the ESPIE service routine passes to the ESPIE exit routine when it receives control. When the exit routine receives control, register 1 contains the address of the EPIE. The format of the EPIE is:

Hex Location	Contents
X'00'	'EPIE'
X'04'	Address of user-supplied parameter list.
X'08'	Contents of the general purpose registers at the time of the interruption. The registers are stored in order of register 0 to register 15.
X'48'	Old program status word in EC mode.
X'50'	Program interruption information consisting of the 2-byte ILC followed by the 2-byte interruption code.
X'54'	Reserved
X'58'	Contents of the access registers at the time of the interruption. The registers are stored in the order of register 0 to register 15.

## Register Contents Upon Entry to User's Exit Routine

When control is passed to your routine, the register contents are as follows:

Register	Contents
0	Not significant to exit.
1	Address of EPIE.
2-12	Same as when program interruption occurred.
13	Address of save area for the main program. The exit routine cannot use this area.
14	Return address to the control program.
15	Address of the exit routine. The exit routine must be in virtual storage when it is required, and must return control to the control program using the address in register 14. The control program restores all 16 registers from the EPIE.

## Messages

The following are ABEND codes from ESPIE.

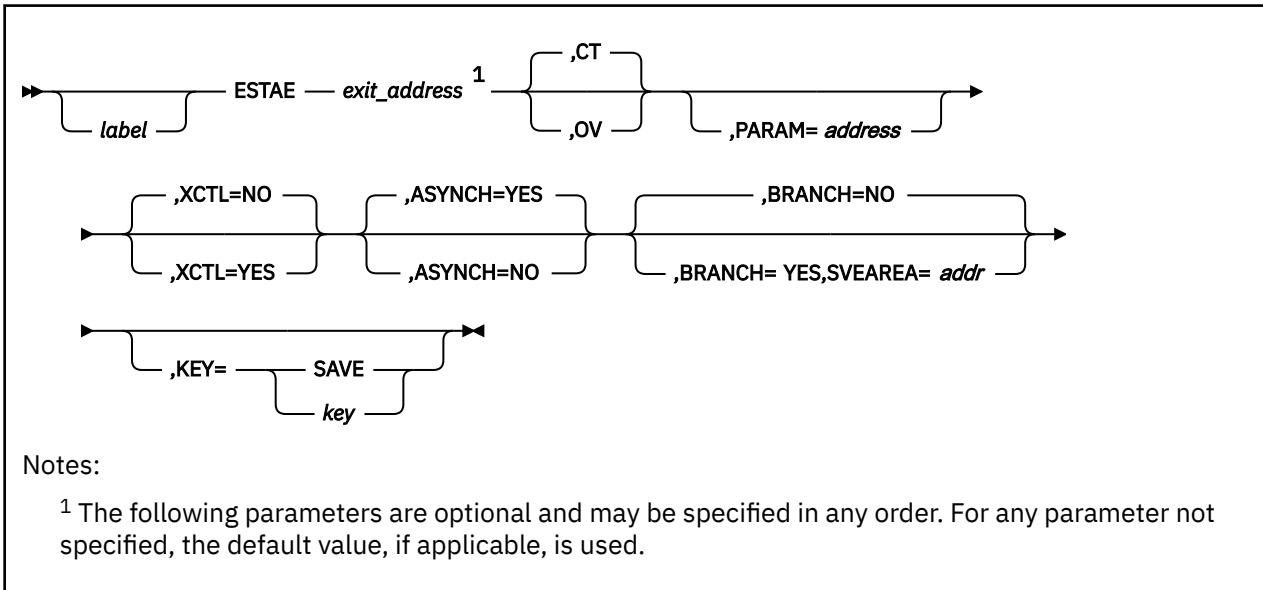
ABEND Code	Reason Code	Meaning
46D	04	Invalid function code
46D	08	Invalid parameter list address
46D	12	Invalid exit address
46D	20	Invalid reset token

## ESTAE

The ESTAE macro is available in standard, list and execute formats.

### Standard Format

See also “List Format” on page 227 and “Execute Format” on page 228.



### Purpose

Use the ESTAE macro to specify an exit routine for a task that will gain control if the task abends.

When a task ends abnormally, GCS usually carries out task termination activities for the task. These activities include the release of locks, storage, and other resources associated with the task.

However, you may wish to provide your task with your own exit routine that receives control if an abend occurs. This exit routine can be designed to find a solution to the problem, try the task again, or allow task termination to continue. Use the ESTAE macro to specify and describe this exit routine.

#### ***exit\_address***

Specifies the address of the exit routine that is to gain control if your task ends abnormally.

If you specify an address of zero, then the exit routine you most recently defined through the ESTAE macro is canceled.

You can write this parameter as an assembler program label or as register (2) through (12).

#### **CT**

Indicates that you are specifying a new exit routine for the active task.

Because you may have several exit routines, this new exit routine will supplement any that may currently be defined for the task.

If neither the CT nor the OV parameter is specified, then CT is assumed, by default.

#### **OV**

Indicates that you wish to modify (overlay) certain parameters that you specified in your last ESTAE instruction, yet maintain the status of the current exit routine as the current exit routine.

Specify only those parameters that you want overlaid, along with any necessary values.

**PARAM**

Specifies the address of a parameter list that is to be passed to your exit routine, should it ever gain control.

It is your responsibility to provide this parameter list.

You can write this parameter as an assembler program label or as register (2) through (12).

**XCTL**

Indicates whether your exit routine will maintain its status as current exit routine if your task transfers control to a module through the XCTL macro. See “[XCTL](#)” on page 373.

**NO**

Indicates that if your task transfers control to a module through the XCTL instruction, and the module abends, then the exit routine in question will not gain control. This is the default.

**YES**

Indicates that if your task transfers control to a module through the XCTL instruction, and the module abends, then the exit routine in question will gain control.

**ASYNCH**

Indicates whether asynchronous exits will be allowed while your exit routine is running.

**YES**

Indicates that you will allow asynchronous exits while your exit routine is running. This is the default.

You must specify ASYNCH=YES if your exit routine requests supervisor services that require such interrupts. These supervisor services include general I/O, ATTACH ETXR, IUCV, STIMER, and SCHEDEX.

**NO**

Indicates that you will allow no asynchronous exits while your exit routine is running.

**BRANCH**

Specifies whether your task should branch directly to the ESTAE service routine.

**YES**

Specifies that the task should branch directly to the ESTAE service routine.

**NO**

Specifies that you want to use the customary SVC interface. This option is the default.

**SVEAREA**

Specifies the address of a 72-byte register save area that you must reserve when BRANCH=YES. You can write this parameter as an assembler program label or as a register number.

**KEY**

If BRANCH=YES and your task is not running in key 0, you must supply the KEY parameter.

**SAVE**

Causes the system to save the current program status word (PSW) protection key in register 2. The system then issues a set PSW key (SPKA) instruction, changing the key to zero. When the ESTAE service routine returns control, it restores the original PSW key from register 2. You may want to save and, later, restore the contents of register 2.

**key**

If you know the PSW protection key of your task, then specify it as a number from 0 to 15.

**Usage**

1. Your task may use the ESTAE macro many times while processing. However, only the latest exit routine specified remains current. Any others are pushed down in a stack. If the current exit routine is canceled, then the next one in the stack moves to the top, becoming the current exit routine. Conversely, if you specify a new exit routine, then any others in the stack move down one position and the new one becomes the current exit routine.

2. The current exit routine loses its status as the current exit routine under one of these conditions the:
  - Module that defined it, through the ESTAE macro, ends
  - Task issues the ESTAE macro, specifying zero as the EXIT ADDRESS
  - Exit routine ends abnormally
  - Exit routine allows termination of the task that defined it to continue
  - Task attempts to transfer control using the XCTL macro when XCTL=YES is not specified.

In each case, the exit routine defined by the previous ESTAE instruction moves to the top of the stack and assumes the role of current exit routine.

3. ESTAE instructions that cancel the current exit routine or overlay parameters must be issued by the same program that defined the current exit routine.
4. Your exit routine can diagnose the cause of the abend, and then retry the task at some entry point. Or, it can simply allow GCS to perform usual termination activities and shut the task down.
5. Whenever a task abends, GCS attempts to build a system diagnostic work area (SDWA), as described in [“IHASDWA” on page 273](#).
6. When you branch directly to the ESTAE service routine, your task must be in supervisor state and disabled for interrupts.
7. An interrupt handler cannot use the branch interface to the ESTAE service routine.
8. Because this method of invoking the ESTAE macro avoids the supervisor call, no trace entry for the macro is generated.
9. The CVT mapping macro must be assembled as a DSECT into your program.
10. The AMODE of the exit will always be considered the AMODE of the caller.
11. If storage was available for the SDWA, then when your exit routine receives control, the registers contain the following:

<b>Register</b>	<b>Contents</b>
0	A return code of 16(10), signifying that no I/O processing was performed.
1	Address of the SDWA.
2-12	Unpredictable.
13	Address of a register save area.
14	A return address.
15	Address of the current exit routine.

Here, the SETRP macro should be issued to notify the GCS supervisor of the action that is to be taken. See [“SETRP” on page 340](#).

12. If storage was not available for the SDWA, then when your exit routine receives control, the registers contain the following:

<b>Register</b>	<b>Contents</b>
0	A return code of 12(C), signifying that no SDWA was obtained.
1	The completion code passed by the ABEND macro. See <a href="#">“ABEND” on page 162</a> .
2	The address of the parameter list intended for the exit routine. Or, if none was intended, zero.
3-13	Unpredictable.
14	Address of an SVC 3 instruction.
15	Address of the current exit routine.

13. If no SDWA was obtained, then your exit routine must set the registers in the following manner just before returning control to the GCS supervisor.

<b>Register</b>	<b>Contents</b>
0	The address of a recovery routine, if one is to be scheduled.
15	A return code. Specifically:
0	Termination should be continued. Any previously defined exit routines will move toward the top of the stack.
4	A recovery routine is to be scheduled. The address of this routine can be found in register 0.

14. An exit routine always runs in the same key as the task that defined it and is enabled for the same interrupts. The same holds true for any retry routine.

15. If storage was available for an SDWA, then when the recovery routine gains control, the registers contain the following:

<b>Register</b>	<b>Contents</b>
0	Zero, indicating that storage for the SDWA was available.
1	Address of the SDWA.
2-13	Unpredictable.
14	Address of an SVC 3 instruction.
15	Address of the recovery routine.

16. If storage was not available for an SDWA, then when the recovery routine gains control, the registers contain the following:

<b>Register</b>	<b>Contents</b>
0	12(C), indicating that storage for the SDWA was not available.
1	The value of the PARAM parameter that was specified in the ESTAE instruction associated with the current exit routine.
2	Zero.
3-13	Unpredictable.
14	Address of an SVC 3 instruction.
15	Address of the recovery routine.

17. The SPLEVEL macro need not be issued unless you want an ESTAE macro used by GCS that has an expanded parameter list, which is designed for use in the 31-bit addressing mode. A 31-bit parameter list is incompatible if you are running under the 370 Accommodation Facility. However the SPLEVEL macro lets you select either the 24-bit version or the 31-bit version

18. This macro supports both 24 and 31 bit address expansions of the parameter list. The macro expansion is controlled by the internal macro SPLEVEL. The default value is 31.

### Examples

```
DEFEXT ESTAE (4),CT,PARAM=PLIST3
```

The task defines an exit routine that will gain control in case of an abend. Register 4 contains the address of the exit routine in question. The CT parameter indicates that this exit routine is new. The parameter list

at the address associated with the label PLIST3 will be passed to the exit routine if it ever gains control. DEFEXT is the label on this instruction.

## Return Codes and ABEND Codes

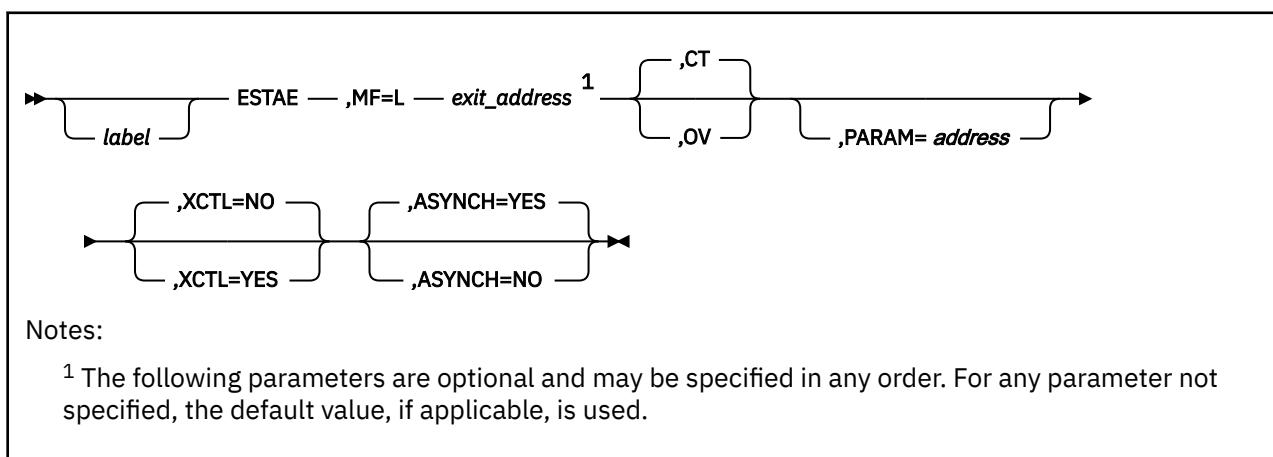
When this macro completes processing, it passes to the caller a return code in register 15.

Hex Code	Decimal Code	Meaning
X'00'	0	Function completed successfully.
X'04'	4	The OV parameter was specified along with a valid EXIT ADDRESS. However, either there is no current exit routine defined, or the ESTAE macro was not issued by the same active program module that defined the current exit routine.
X'OC'	12	An attempt was made to cancel the current exit routine. However, either no current exit routine is defined, or the ESTAE instruction was not issued by the same active program module that defined the current exit routine.
X'14'	20	The ESTAE macro was unable to acquire the storage necessary for it to process.

ABEND Code	Meaning
0F8	The GCS supervisor was called in access register mode.
13C	An invalid ESTAE request was made.

## List Format



## Purpose (List Format)

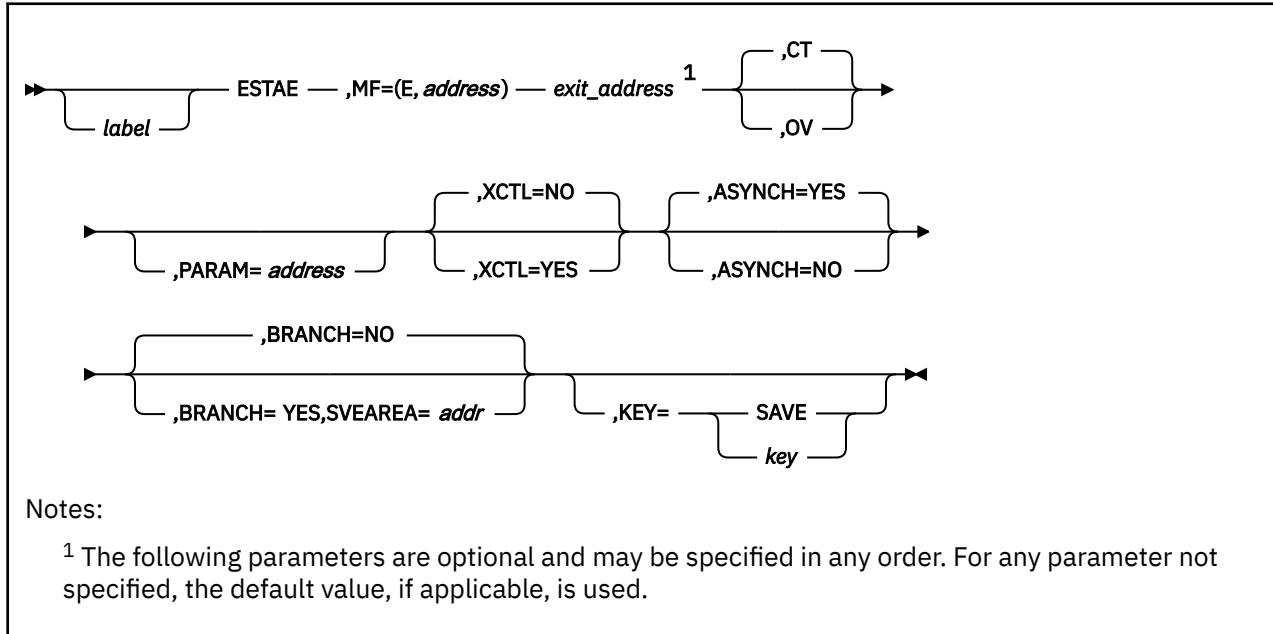
This format of the macro generates an in-line parameter list based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation.

## Added Parameter

### MF=L

Specifies the list format of this macro.

## Execute Format



## Purpose (Execute Format)

This format of the macro generates code that runs the function, using a parameter list whose address you specify.

## Added Parameter (Execute Format)

### MF=(E, *address*)

*address* specifies the address of the parameter list to be used by the macro.

You can add or modify values in this parameter list by specifying them in this macro.

# EXECCOMM

---

## Format



## Purpose

Use the EXECCOMM macro to set up the interface that allows a program to gain access to the variables within the EXEC that started it.

EXECs running under GCS frequently call other programs, such as commands and subcommands. Often these programs need access to the variables within the EXEC that called them.

## Parameters

### REQLIST

Specifies the address of the first (or only) shared variable request block in a chain of such blocks.

A shared variable request block is a control block that defines an EXEC variable to which your program wants access. This will describe how the variable will be used. Your program must create one shared variable request block for each variable to which it wants access. If there is more than one request block, they must be strung together in a chain.

Detailed information on the EXECCOMM facility and shared variable request block formatting is provided in the [z/VM: REXX/VM Reference](#)

You can write this parameter as an RX-type address or as register (2) through (12).

## Usage

1. The EXECCOMM macro stores the address of the first (or only) request block in the chain in a register. This is then passed to the REXX/VM interpreter, which processes your request. The EXECCOMM macro then passes a return code back to your program that describes if and how the function was completed.
2. EXEC variables may be inspected, modified, or deleted by a program that gains access to them.
3. For a program within a specific task to issue the EXECCOMM instruction, an EXEC must be active within that task.

## Return Codes and ABEND Codes

When this macro completes processing, it passes to the caller a return code in register 15.

Hex Code	Decimal Code	Meaning
X'FFFFFFFF'	-1	No EXEC was active within the task.
X'FFFFFFF'E'	-2	Insufficient storage is available to process your request.
0 or any positive number except 8	0 or any positive number except 8	Function completed successfully.
X'00000008'	8	Invalid variable name.

<b>ABEND Code</b>	<b>Reason Code</b>	<b>Meaning</b>
FCB	0D01	An invalid address exists in a shared variable request block, or the address of the block itself is invalid.

# FLS

---

## Format

►► FLS ►►

## Purpose

Use the FLS macro to gain access to certain fields in your virtual machine's low storage.

There are several fields within your virtual machine's low storage (page 0) to which you can gain access.

## Parameters

The FLS macro accepts no parameters.

## Usage

1. The FLS macro gives you access to the following fields residing in page 0 of your virtual storage:

### **FLSVMID**

The user ID associated with your virtual machine.

### **FLSLVL**

A fullword that contains the component level and service level of your GCS system. The GCSLEVEL macro can be used to map this field. See “[GCSLEVEL](#)” on page 238.

### **FLSRLVL**

Second byte of FLSLVL. It contains the component level.

### **FLSSLVL**

Second halfword of FLSLVL. It contains the service level.

### **FLSID**

A fullword that contains the signal services machine ID and the task ID of the active task.

### **FLSPOST**

The branch entry address for the POST macro. See “[POST](#)” on page 314.

### **FLSDUMP**

A pointer to the dump receiver. This is the virtual machine in the GCS group that is to receive all the dumps generated in the group.

### **FLSFLG**

A fullword that contains two flags to let applications running on GCS know if the machine is in XC mode and if hardware compression is supported.

### **FLSFLGX**

This flag will be on if the machine is running in XC mode.

### **FLSFLGH**

This flag will be on if hardware compression is supported.

2. The following table shows the format of the FLS fields:

Address	Field
512 (X'200')	RESERVED

<b>Address</b>	<b>Field</b>
516 (X'204')	FLSVMID
524 (X'20C')	FLSLVL -- FLSRLVL FLSSLVL
528 (X'210')	FLSIDS
532 (X'214')	RESERVED
536 (X'218')	FLSPOST
540 (X'21C')	RESERVED
544 (X'220')	RESERVED
548 (X'224')	RESERVED
552 (X'228')	RESERVED
556 (X'22C')	RESERVED
560 (X'230')	RESERVED
564 (X'234')	RESERVED
568 (X'238')	RESERVED
572 (X'23C')	FLSDUMP
580 (X'244')	FLSFLG - - FLSFLGXC FLSFLGHC
652 (X'28C')	RESERVED
664 (X'298')	END

### Return Codes and ABEND Codes

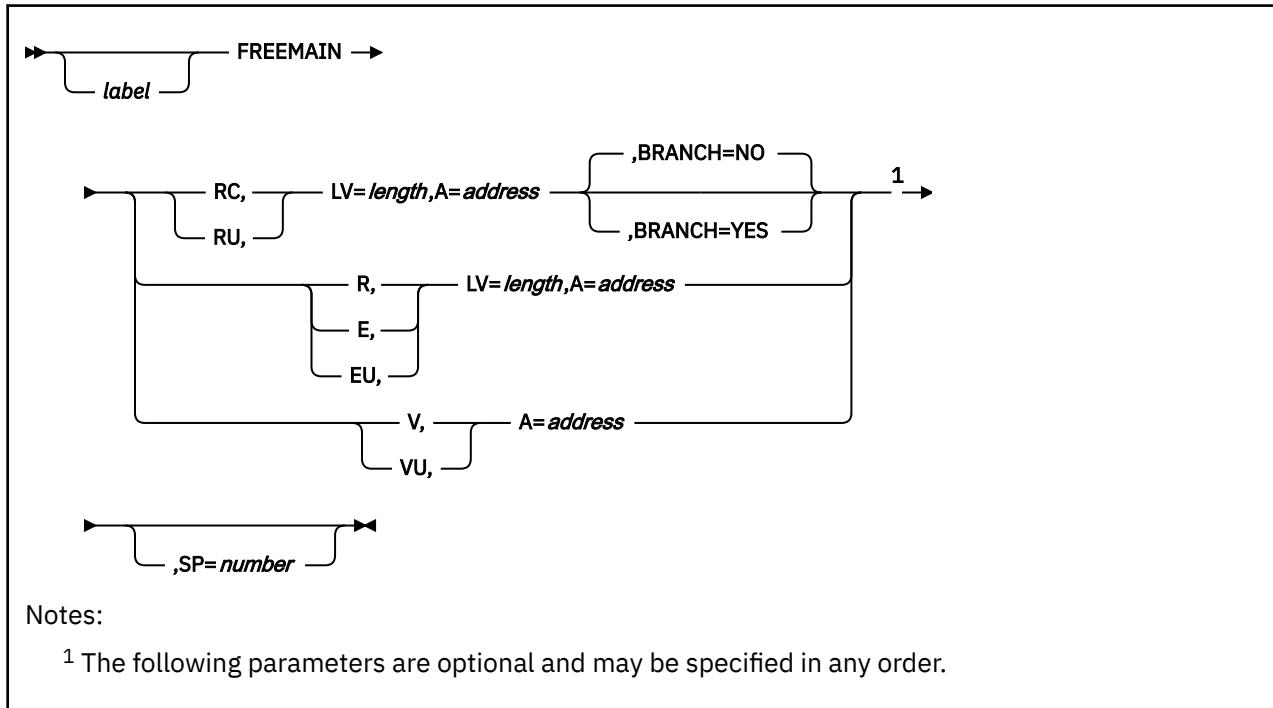
The FLS macro generates no return codes and no ABEND codes.

## FREEMAIN

The FREEMAIN macro is available in standard, list and execute formats.

### Standard Format

See also [“List Format” on page 236](#) and [“Execute Format” on page 237](#).



### Purpose

Use the FREEMAIN macro to free a contiguous block of storage.

The storage management function of GCS enables a task to dynamically obtain and free contiguous blocks of storage as required.

### Parameters

#### RC

Indicates that your register request to free the storage is conditional.

#### RU

Indicates that your register request to free the storage is unconditional.

#### BRANCH

Specifies whether your task should branch directly to the FREEMAIN service routine.

#### YES

Specifies that your task should branch directly to the FREEMAIN service routine.

#### NO

Specifies that you want to use the customary SVC interface. This option is the default.

#### R

Indicates that your register request to free the storage is unconditional.

**E or  
EU**

Indicates that this is an unconditional request to free a certain element of storage.

**V or  
VU**

Indicates that your request to free the storage is unconditional.

This storage was originally obtained by using the VC or VU parameter on the GETMAIN instruction. Hence, it was a request for a variable amount of storage.

**LV**

Specifies the length, in bytes, of the storage block you want to free.

This length should be a multiple of eight. If it is not, then GCS rounds it up to the nearest multiple of eight.

If the R parameter is specified, then LV=(0) can be coded as well. If it is, then the high-order byte of register 0 must contain the storage block's subpool number and the 3 low-order bytes must contain the length of the storage block.

You can write this parameter as an assembler program label or as register (2) through (12).

**A**

Specifies the address of a one or two-word list, starting on a fullword boundary.

If you select the E, EU, R, RC, or RU parameter, then this list need contain only one fullword. This word must contain the address of the block of storage to be freed.

If you select the V or VU parameter, then this list must contain two fullwords. The first word must contain the address of the block of storage you want to free. The second word must contain the length of this block, in bytes.

The storage block must begin on a doubleword boundary. Its length must be a multiple of eight. If it is not, then GCS rounds the length up to the nearest multiple of eight.

You can write this parameter as register (2) through (12) or as an assembler program label. If you express it as a register, and if you select the R, RC, or RU parameter, then the register must contain the address of the block you want to free, not the address of any fullword that contains that address. Here, you may also use register (1) to specify the address.

**SP**

Specifies the subpool associated with the storage block you want to free.

A subpool is identified by a number from 0 to 255. A subpool number describes the characteristics of the block of storage to which it is assigned. The subpool number that you specify (explicitly or by default) must be the same as you specified in the corresponding GETMAIN macro.

For a definition of all subpool numbers, see ["GETMAIN" on page 257](#).

If you omit this parameter, the subpool number is 0, by default. You can write it as an assembler program label or as register (2) through (12). Or, if the R parameter is specified, then LV=(0) can be coded as well. If it is, then the high-order byte of register 0 must contain the storage block's subpool number and the 3 low-order bytes must contain the length of the storage block.

## Usage

1. Callers in either 24-bit or 31-bit addressing mode must use only RC or RU to free storage above the 16MB line.
2. If you specify BRANCH=YES your task must be in supervisor state, key 0, and disabled for interrupts.

You can include BRANCH=YES only with the RC and RU parameters of the FREEMAIN macro.

The macro destroys the contents of register 3. You may want to save and, later, restore the contents of register 3.

Before the branch, register 13 must contain the address of a 72-byte register save area. You can obtain this save area by using the GCSSAVE macro.

The GCSSAVI macro must be used in place of the GCSSAVE macro to obtain the save area if the branch entry to FREEMAIN is from an exit defined by GENIO with EXITBR=YES, or from an exit defined by IUCVCOM with BRANCH=YES.

The CVT mapping macro must be assembled as a DSECT into your program.

## Examples

```
FREEMAIN RC, LV=400, A=(2), SP=10
```

The task requests that 400 bytes of storage in subpool 10 be freed. Register 2 contains the address of this storage block. This is a conditional request, a return code of 0 would result if the storage were in fact freed. If it were not, then a return code of 4 would result and the storage in question would remain unchanged.

```
GETMAIN VC LA=RANGE, A=DBLWD
.
.
.
FREEMAIN V, A=DBLWD
```

The task requested a variable amount of storage within a certain range. This range was specified in the two-word list at the address associated with the label RANGE. The task provided a two-word list at the address associated with the label DBLWD. When GCS gave the storage to the task, it stored the address of the storage block in the first word of this list. It then stored the actual length of the storage block in the second word. The task retained the values in this two-word list and later requested that the same storage block be freed.

## Return Codes and ABEND Codes

When this macro completes processing a conditional request, it passes to the caller a return code in register 15. **For the RC parameter only:**

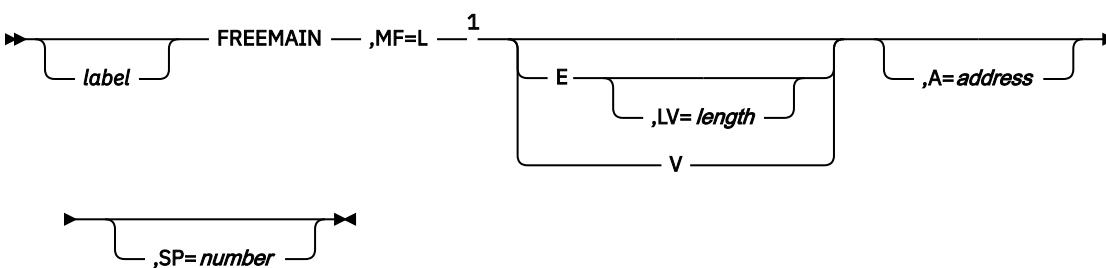
Hex Code	Decimal Code	Meaning
X'00'	0	Function completed successfully.
X'04'	4	Function was not completed.

ABEND Code	Meaning
0F8	The GCS supervisor was called in access register mode.
305	A FREEMAIN macro contained a subpool specification error.
605	Either a FREEMAIN macro contained an invalid address in the A parameter or an invalid parameter list address was passed to the macro.
705	An irrecoverable machine, system, or other error occurred while processing the FREEMAIN macro.
905	The address of the storage area specified in a FREEMAIN macro was not on a doubleword boundary.
A05	Either the area you tried to free overlapped into an already free area, or it has been locked through the PGLOCK macro.

ABEND Code	Meaning
D05	<p>One of several things happened:</p> <ul style="list-style-type: none"> <li>The FREEMAIN macro attempted to free an area of storage not allocated to your task.</li> <li>You specified zero or a negative number in the LV parameter.</li> <li>The key is different from what it was when the storage was allocated.</li> </ul>
E05	You specified a parameter that GCS does not support.
30A	A FREEMAIN macro, with the R parameter specified, contained a subpool specification error.
70A	An irrecoverable machine, system, or other error occurred while processing the FREEMAIN macro with the R parameter specified.
90A	The address of the storage area specified in a FREEMAIN instruction, with the R parameter specified, was not on a doubleword boundary.
A0A	Either the area to be freed by a FREEMAIN instruction, with the R parameter specified, overlapped into an already free area or was locked through the PGLOCK macro and never unlocked.
D0A	<p>One of several things happened:</p> <ul style="list-style-type: none"> <li>The FREEMAIN macro, with the R parameter specified, attempted to free an area of storage not allocated to your task.</li> <li>You specified zero or a negative number in the LV parameter.</li> <li>The key is different from what it was when the storage was allocated.</li> </ul>
E0A	A FREEMAIN instruction, with the R parameter specified, specified another parameter that GCS does not support.
378	A FREEMAIN macro, with the RU parameter specified, contained a subpool specification error.
778	An irrecoverable machine, system, or other error occurred while processing the FREEMAIN macro with the RU parameter specified. It may also be that an error, involving the release of free storage, occurred within the GCS supervisor.
978	The address of the storage area specified in a FREEMAIN macro, with the RU parameter specified, was not on a doubleword boundary.
A78	The area to be freed by the FREEMAIN macro, with the RU parameter specified, overlapped a free area of storage or is an area that was locked through the PGLOCK instruction.
D78	<p>One of several things happened:</p> <ul style="list-style-type: none"> <li>The FREEMAIN macro, with the RU parameter specified, attempted to free an area of storage not allocated to your task.</li> <li>You specified zero or a negative number in the LV parameter.</li> <li>The key is different from what it was when the storage was allocated.</li> </ul>
E78	A FREEMAIN macro, with the RU parameter specified, specified another parameter that is not supported by GCS.

## List Format

**Notes:**

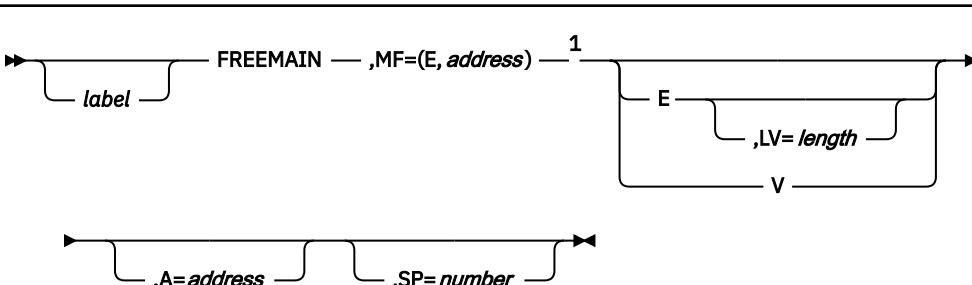
- <sup>1</sup> The following parameters are optional and may be specified in any order. For any parameter not specified, the default value, if applicable, is used.

**Purpose (List Format)**

This format of the macro generates an in-line parameter list based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation. Only the preceding parameters listed are valid in the list format of this macro.

**Added Parameter****MF=L**

Specifies the list format of this macro.

**Execute Format****Notes:**

- <sup>1</sup> The following parameters are optional and may be specified in any order. For any parameter not specified, the default value, if applicable, is used.

**Purpose (Execute Format)**

This format of the macro generates code that runs the function, using a parameter list whose address you specify.

Only the preceding parameters listed are valid in the execute format of this macro.

**Added Parameter(Execute Format)****MF=(E, address)**

*address* specifies the address of the parameter list to be used by the macro.

You can add or modify values in this parameter list by specifying them in this macro.

# GCSLEVEL

---

## Format

►► GCSLEVEL ►►

## Purpose

Use the GCSLEVEL macro to map the component level of your GCS system from the FLSLVL field found in your virtual machine's low storage.

There are several fields in the low storage of your virtual machine (page 0) to which you can gain access. One of these fields, the FLSLVL field, accommodates the component level of the GCS system you are using. See “[FLSLVL](#)” on page 231.

The GCSLEVEL macro generates equates for use in determining the component level of GCS.

## Parameters

The GCSLEVEL macro accepts no parameters.

## Usage

1. The number of bytes for each subfield are in parentheses. The format of the FLSLVL field is:

RESERVED (1)	Component Level (1)	Service Level (2)
--------------	---------------------	-------------------

2. The equates for the current and previous GCS component levels are:

Label	Release	Component of
GCSL	EQU X'01'	VM/SP Release 4
GCS5	EQU X'02'	VM/SP Release 5
GCS6	EQU X'03'	VM/SP Release 6
GCS370	EQU X'04'	VM/ESA Release 1.0 - 370 Feature
GCSESA1	EQU X'05'	VM/ESA Release 1.0
GCSESA11	EQU X'06'	VM/ESA Release 1.1
GCSESA2	EQU X'07'	VM/ESA Release 2.0
GCSESA21	EQU X'08'	VM/ESA Release 2.1
GCSESA22	EQU X'09'	VM/ESA Release 2.2
GCSV2R1	EQU X'0A'	VM/ESA Version 2, Release 1.0
GCSV2R2	EQU X'0B'	VM/ESA Version 2, Release 2.0
GCSV2R3	EQU X'0C'	VM/ESA Version 2, Release 3.0
GCSV2R4	EQU X'0D'	VM/ESA Version 2, Release 4.0
GCSV3R1	EQU X'0E'	z/VM Version 3, Release 1.0
GCSV4R1	EQU X'0F'	z/VM Version 4, Release 1.0
GCSV4R2	EQU X'10'	z/VM Version 4, Release 2.0
GCSV4R3	EQU X'11'	z/VM Version 4, Release 3.0
GCSV4R4	EQU X'12'	z/VM Version 4, Release 4.0
GCSV5R1	EQU X'13'	z/VM Version 5, Release 1.0
GCSV5R2	EQU X'14'	z/VM Version 5, Release 2.0
GCSV5R3	EQU X'15'	z/VM Version 5, Release 3.0
GCSV5R4	EQU X'16'	z/VM Version 5, Release 4.0
GCSV6R1	EQU X'17'	z/VM Version 6, Release 1.0
GCSV6R2	EQU X'18'	z/VM Version 6, Release 2.0
GCSV6R3	EQU X'19'	z/VM Version 6, Release 3.0
GCSV6R4	EQU X'1A'	z/VM Version 6, Release 4.0
GCSV7R1	EQU X'1B'	z/VM Version 7, Release 1.0
GCSV7R2	EQU X'1C'	z/VM Version 7, Release 2.0
GCSV7R3	EQU X'1D'	z/VM Version 7, Release 3.0
GCSV7R4	EQU X'1E'	z/VM Version 7, Release 4.0

3. The SERVICE LEVEL information is a halfword, stored in binary format.

## **Return Codes and ABEND Codes**

The GCSLEVEL macro generates no return codes and no ABEND codes.

## GCSSAVE

---

### Format



### Purpose

Use the GCSSAVE macro to create a register save area when branching directly.

GCSSAVE allocates space for and returns the address of a register save area. This macro can be used only when your task branches directly to one of the following service routines: ESTAE, FREEMAIN, GENIO, GETMAIN, IUCVCOM, SCHEDEX, or WAIT.

### Parameters

#### GET

Indicates that you want to allocate space for a 72-byte register save area and return its address.

#### FREE

Indicates that you want to release the space allocated by the GET option.

### Usage

1. Use the GCSSAVE macro only when you intend to branch directly to the ESTAE, FREEMAIN, GENIO, GETMAIN, IUCVCOM, SCHEDEX or WAIT service routine. In all other instances, create your own register save area.
2. The caller must be disabled for interrupts, in supervisor state, and in key 0.
3. After GCSSAVE GET, you must use GCSSAVE FREE to free the allocated space.
4. Use of the GCSSAVE macro is optional; you may give the address of your own register save area to the branch entries instead.
5. The program issuing the GCSSAVE macro, receives the following information in its registers.

Register	Contents
13	The address of the register save area.

### Return Codes and ABEND Codes

The GCSSAVE macro generates no return codes or abend codes.

# GCSSAVI

---

## Format



## Purpose

Use the GCSSAVI macro to create a register save area when a branch entry to GETMAIN or FREEMAIN is issued from an exit which was defined for GENIO with EXITBR=YES, or for IUCVCOM with BRANCH=YES. GCSSAVI allocates space for and returns the address of a register save area.

## Parameters

### GET

Indicates that you want to allocate space for a 72-byte register save area and return its address.

### FREE

Indicates that you want to release the space allocated by the GET option.

## Usage

1. Use the GCSSAVI macro only when a branch entry to GETMAIN or FREEMAIN is issued from an exit which was defined for GENIO with EXITBR=YES, or for IUCVCOM with BRANCH=YES.
2. The caller must be disabled for interrupts, in supervisor state, and in key 0.
3. After GCSSAVI GET, you must use GCSSAVI FREE to free the allocated space.
4. Use of the GCSSAVI macro is optional; you may give the address of your own register save area to the branch entries instead.
5. The program issuing the GCSSAVI macro, receives the following information in its registers.

Register	Contents
13	The address of the register save area.

## Return Codes and ABEND Codes

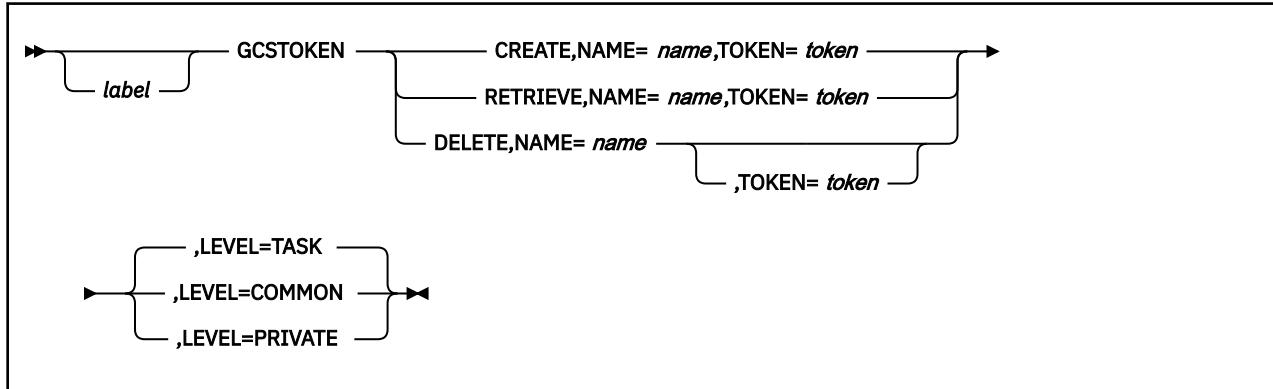
The GCSSAVI macro generates no return codes or abend codes.

## GCSTOKEN

The GCSTOKEN macro is available in standard, list, list address, and execute formats.

### Standard Format

See also [“List Format” on page 244](#), [“List Address Format” on page 245](#) and [“Execute Format” on page 245](#).



### Purpose

Use the GCSTOKEN macro to allow two or more programs to share data. The programs can be running under the same task, or different tasks, or in different virtual machines in the GCS group.

Programs often need to maintain pointers to control blocks or data. The pointers are persistent over termination of a task in the case of private level Name/Token pairs, and over the reset of a single virtual machine in the case of common level pairs.

Use the GCSTOKEN macro to create, retrieve, or delete, a Name/Token pair.

### Parameters

#### **CREATE**

Indicates that a Name/Token pair is to be established. Both NAME and TOKEN must be specified. If LEVEL is not specified the default is TASK. Programs running in problem state can only create a Name/Token pair at the task level.

#### **RETRIEVE**

Indicates that a token is to be located and placed into a 16 byte buffer provided by the caller. The search is based on the name, and level provided as input. If LEVEL is not specified the default is TASK.

#### **DELETE**

Indicates that a Name/Token pair is to be deleted from a given level. For both private and common level Name/Token pairs, the task that deletes a pair must be running in supervisor state. In the case of a Name/Token pair created at the common level, the task that deletes a pair must also be running in the virtual machine where the Name/Token pair was created. If LEVEL is not specified the default is TASK.

#### **NAME**

Is the address of a unique name. The buffer which contains the name must be 16 bytes in length. All 16 characters in the buffer will be used as input to the requested function. For the CREATE and DELETE functions, names cannot start with the characters "GCT", or a X"00" (null).

If the name is defined on the private level it must be unique for the virtual machine, and if defined on the common level it must be unique to the GCS group.

The NAME parameter must be specified on a CREATE, RETRIEVE, and DELETE.

You can write this parameter as an assembler program label or as register (2) through (12).

#### TOKEN

Is the address of a data buffer which is 16 bytes in length. For CREATE this buffer can contain any data. It can be in any format desired by the application creating the Name/Token pair.

This field must be specified on a CREATE and will be filled in on a RETRIEVE by the system. It need not be specified on a DELETE. If specified on a DELETE it will be ignored.

You can write this parameter as an assembler program label or as register (2) through (12).

#### LEVEL

Specifies the level of access you desire for the Name/Token pair.

If LEVEL is not specified the default is TASK.

If LEVEL=PRIVATE is specified the Name/Token pair is accessible to programs running on tasks in the virtual machine where the Name/Token pair is created. The Name/Token pair can be accessed only if the 16 byte name is known to the application. The Name/Token pair will be deleted only if it is explicitly deleted by an authorized user or the virtual machine resets.

If LEVEL=COMMON is specified the Name/Token pair is accessible to programs running on tasks in any virtual machine in the GCS group. The Name/Token pair can be accessed only if the 16 byte name is known to the application. The Name/Token pair will be deleted only if it is explicitly deleted by an authorized user in the virtual machine that created it or if the recovery machine resets. When running in a single user group environment, common level Name/Token pairs cannot be shared because no other machines are in the group.

If LEVEL=TASK is specified the Name/Token pair will be obtained and available on a task level and associated with the independent task if one exists and will be automatically deleted when the task terminates. If one does not exist it will be associated with the command's task, and deleted when the command completes. Problem programs can only obtain task level Name/Token pairs and thus cannot share Name/Token pairs they create with other independent tasks.

#### Examples

Examples of STANDARD formats of the GCSTOKEN macro

Create a Name/Token pair at the private level.

```
GCSTOKEN CREATE,NAME=TOKEN001,TOKEN=TOKEN,LEVEL=PRIVATE
LTR   R15,R15           Check the return code
...
```

Retrieve a Token using the NAME and LEVEL.

```
GCSTOKEN RETRIEVE,NAME=TOKEN001,TOKEN=BUFF01,LEVEL=PRIVATE
LTR   R15,R15           Check the return code
...
```

Delete a Name/Token pair.

```
GCSTOKEN DELETE,NAME=TOKEN001,LEVEL=PRIVATE
LTR   R15,R15           Check the return code
...
```

Data areas:

TOKEN001	DS	0D
	DC	CL16 'TOKENNAME-0000001'
TOKEN	DC	XL16 '00EA24580000100000013A0400000348'
BUFF01	DS	XL16

#### Return Codes and ABEND Codes

When this macro completes processing, it passes a return code to the caller in register 15. For return codes 12 (x'0C') and higher, an error message will be displayed giving details of the error condition.

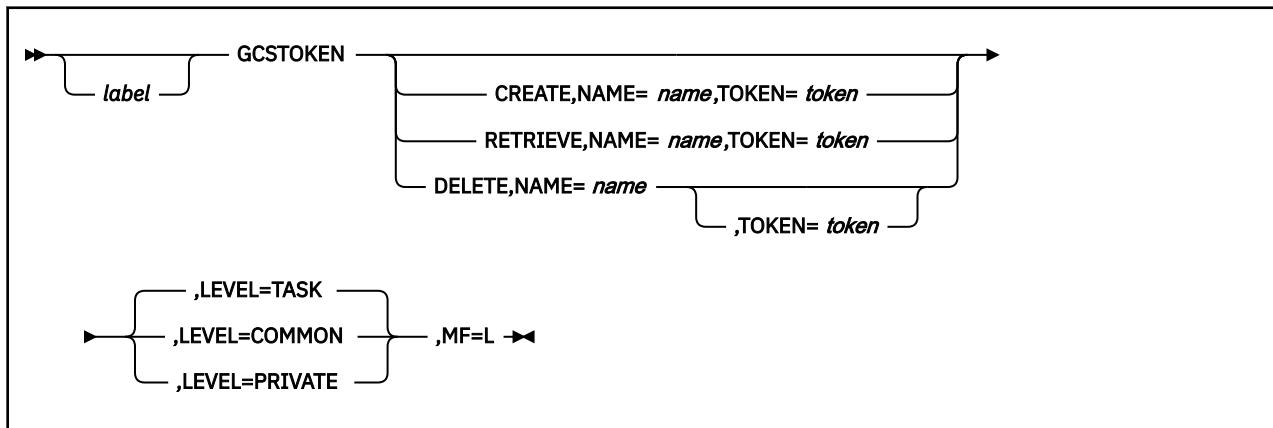
Hex Code	Decimal Code	Meaning
X'00'	0	Function completed successfully.
X'04'	4	NAME requested was not found; RETRIEVE or DELETE failed.
X'08'	8	NAME already exists; CREATE failed.
X'0C'	12	Problem state programs can only issue {CREATE   DELETE} with LEVEL=TASK.
X'10'	16	Only the virtual machine that created a Name/Token pair can DELETE it with LEVEL=COMMON.
X'14'	20	Names cannot begin with {'GCT'   X'00'}; {CREATE   DELETE} failed.
X'26'	38	GCSTOKEN parameter list (plist) does not start with 'GCSTOKEN'.
X'28'	40	GCSTOKEN function requested was not CREATE, RETRIEVE, or DELETE.
X'2A'	42	GCSTOKEN LEVEL specified was not COMMON, PRIVATE, or TASK.
X'68'	104	Insufficient free storage available for GCSTOKEN CREATE function with LEVEL={COMMON   PRIVATE   TASK}.

ABEND Code	Reason Code	Meaning
FCB	C00	No read access to the parameter list or the address in NAME or TOKEN.
FCB	C0A	No write access to the address contained in TOKEN in the parameter list.
FCB	C0B	The GCSTOKEN parameter list contained an address of zero for either the NAME or TOKEN or both.
FCB	C1F	Freemain of private storage failed during delete.
FCB	C2F	Freemain of common storage failed during delete.
FCB	C3F	GCS internal error.

**Note:** See the z/VM: System Messages and Codes section "GCS Abend Codes" for additional information.

## List Format



- If Function is not specified then it must be specified on the Execute Format.
  - If Token is not specified then it must be specified on the Execute Format.
  - If Name is not specified then it must be specified on the Execute Format.

## Purpose (List Format)

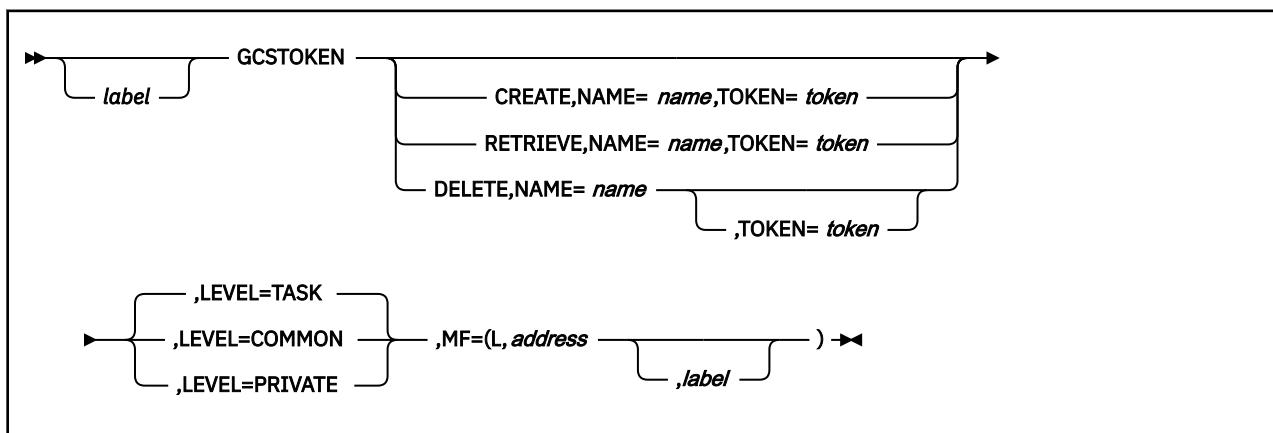
This format of the macro generates an in-line parameter list based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation.

## Added Parameter

$$MF=L$$

Specifies the list format of this macro.

## List Address Format



- If Function is not specified then it must be specified on the Execute Format.
  - If Token is not specified then it must be specified on the Execute Format.
  - If Name is not specified then it must be specified on the Execute Format.

## Purpose (List Address Format)

This format of the macro does not produce any executable code that invokes the function. However, it does produce executable code that moves the parameter values that you specify into a certain parameter list. If you enter the macro using this format, then you must do so before any related invocation of the macro using the execute format.

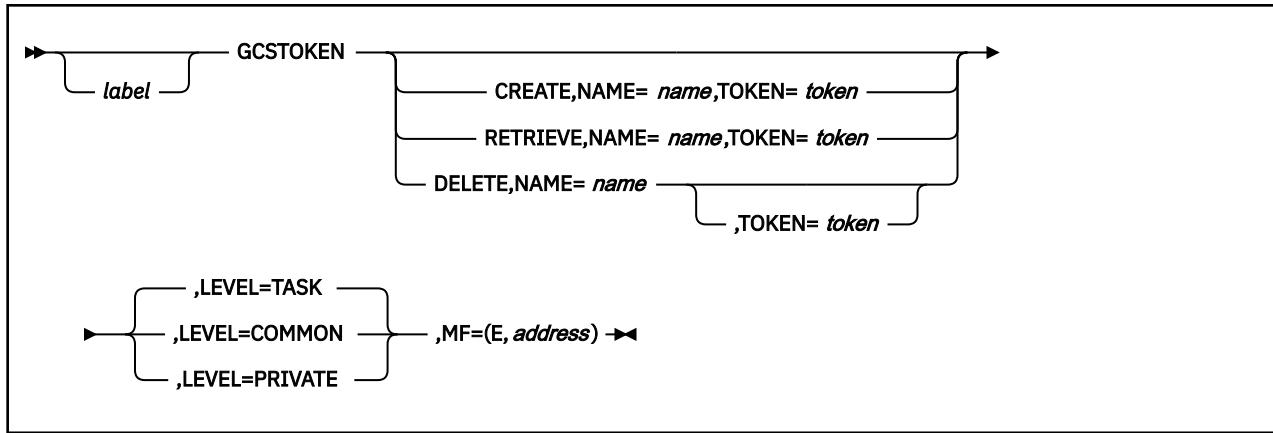
## **Added Parameter (List Address Format)**

**MF=(L, address, label)**

*address* specifies the address of the parameter list into which you want the parameter values you mention placed. This address can be within your program or somewhere in free storage.

*label* is optional and is a user-specified label, indicating that you want to determine the length of the parameter list. The macro expansion equates the label you specify with the length of the parameter list.

## Execute Format



### Purpose (Execute Format)

This format of the macro generates code that executes the function, using a parameter list whose address you specify.

### Added Parameter (Execute Format)

#### **MF=(E, address)**

*address* specifies the address of the parameter list to be used by the macro.

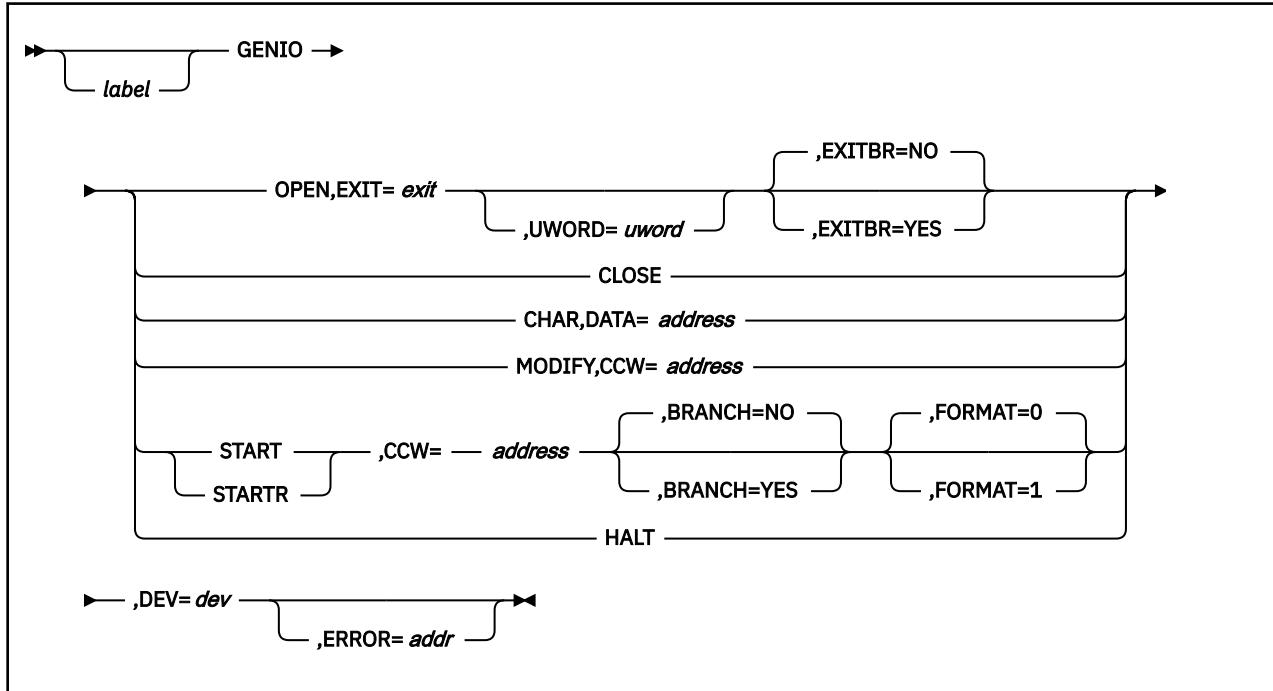
You can add or modify values in this parameter list by specifying them in this format of the macro.

# GENIO

The GENIO macro is available in standard, list, list address and execute formats.

## Standard Format

See also “List Format” on page 254, “List Address Format” on page 255 and “Execute Format” on page 256.



## Purpose

Use the GENIO macro to use general input/output devices.

The GENIO macro allows a program to obtain, use, and release any I/O device, except for DASD devices and the virtual machine console. It is an unauthorized GCS function, except for GENIO STARTR, which is an authorized function.

## Parameters

### OPEN

Indicates that the device specified in the macro should be opened for use by your program.

In doing so, an entry is placed in the GCS general I/O table containing information about the device and your program. Among the information included in the table entry are the device's address, its characteristics, the address in your program to which control is given when an interrupt occurs on the device, and the UWORLD.

No other program may open a device that has been opened by another program. In opening a device, a program obtains exclusive use of it until it closes the device.

The OPEN parameter requires that the address of an exit routine be specified for the device.

### EXIT

Specifies the address of the exit routine for the specified device.

This routine receives control under:

- I/O interrupt occurs on the device that was opened, signalling the end of an I/O operation.
- I/O operation ends because of error.
- Asynchronous interrupt occurs.

This exit routine will handle all interrupts occurring on the specified device.

The exit routine will always be run in the AMODE of the caller.

You can write this parameter as an assembler program label or as register (2) through (12). If you write it as a register, then the register must contain the address of the exit routine.

**UWORD**

An optional fullword parameter that will be passed to the exit routine. It can contain any value you wish.

You can write this parameter as an assembler program label or as register (2) through (12). If you write it as a label, the address of the label is passed to the routine. If you write it as a register, the contents of the register are passed to the routine.

**EXITBR**

Specifies whether the system should branch to your exit routine directly from the I/O interrupt handler.

**YES**

Specifies that you want to branch to your exit routine directly from the I/O interrupt handler.

**NO**

Specifies that you want to schedule your exit routine in the usual fashion. This option is the default.

**CLOSE**

Indicates that the program no longer needs the device specified in the instruction and relinquishes control of it. After this, any program may obtain control over the device.

The program issuing the GENIO macro with this parameter had to have opened the device initially. This parameter clears the entry that was placed in the GCS general I/O table when the device was opened. Any pending I/O requests for the device are deleted from the virtual channel queue, all I/O activity for the device is ended.

Remember that your exit routine cannot receive control resulting from an interrupt occurring on a closed device. Also, remember that the GENIO macro, with the CLOSE parameter specified, cannot be issued from an I/O exit routine.

**CHAR**

Indicates that the characteristics of the specified virtual device and its corresponding real device, if any, should be returned to your program.

Bytes 4-11 of the VRDCBLOK returned by diagnose X'210' are placed in an 8-byte area provided by your program. For details on how to interpret this information, see the description of diagnose X'210' in the *z/VM: CP Programming Services*.

It is not necessary that the device be opened for the program to request this information. The device's characteristics are placed in two consecutive fullwords that your program should reserve for this purpose.

**DATA**

Specifies the address of the data area into which the characteristics of the device are to be placed. Your program must reserve two consecutive fullwords for this purpose.

The first word will contain the characteristics of the virtual device. The second word will contain the characteristics of the real device. If no real device is associated with the virtual device, then the second word will be reset to zero.

You may write this as an assembler program label or as register (2) through (12). If you write it as a register, then that register must contain the address of this data area.

#### **MODIFY**

Indicates that you wish to modify a real CCW (channel control word) after the I/O operation has begun but before it has finished.

First, modify the virtual CCW. Then, enter the GENIO macro with the MODIFY parameter to apply the modification to the real CCW.

Remember that you are allowed to make only the following changes to any CCW:

- A TIC instruction to a NOP instruction
- A NOP instruction to a TIC instruction
- The address in a TIC instruction.

#### **START**

Indicates that a virtual channel program should be started on the specified opened device.

This program is a set of channel control words that instructs the channel which I/O operation to perform. Only one I/O operation can be performed by a single device at one time. Another I/O operation is not accepted by GCS until the previous I/O operation is complete. The latter ends either when a DEVICE END interrupt occurs, or when an error condition arises. The I/O operation is performed in the same key as the program requesting the operation.

#### **STARTR**

Indicates that a real channel program should be started on the specified opened device.

This program is a set of channel control words that instructs the channel which I/O operation to perform. The device in question must be a real device.

The program issuing the GENIO macro with the STARTR parameter must be running in supervisor state in a key other than key 0. And, the DIAG98 parameter must be in the OPTION control statement in the virtual machine's directory entry. Then, the program will build the channel control program in real storage using real addresses. To do this, the program should take advantage of the page-locking and unlocking capabilities of the PGLOCK and PGULOCK macros. See “[PGLOCK](#)” on page 310 and “[PGULOCK](#)” on page 312.

#### **CCW**

If you select the STARTR parameter, then CCW specifies the real address of the first channel control word of the real channel program.

If you select the START parameter, then CCW specifies the virtual address of the first channel control word of the virtual channel program.

If you select the MODIFY parameter, then CCW specifies the virtual address of the channel control word that will be modified.

You can write this parameter as an assembler program label or as register (2) through (12). If you write it as a register, then that register must contain the address of the first CCW.

#### **BRANCH**

Specifies whether your task should branch directly to the GENIO service routine.

##### **YES**

Specifies that your task should branch directly to the GENIO service routine.

##### **NO**

Specifies that you want to use the customary SVC interface. This option is the default.

#### **HALT**

Indicates that the active I/O operation of the specified device is to stop immediately. GCS will issue a HDV (HALT DEVICE) instruction to effect this.

#### **FORMAT**

Specifies the format of the CCW:

**0**

Indicates a format 0 CCW.

**1**

Indicates a format 1 CCW.

**DEV**

Specifies the virtual address of the I/O device that the GENIO macro is to affect.

You can write this parameter as an assembler program label or as register (2) through (12). If you write it as an assembler program label, the address of the device must be in the halfword at that address. If you write it as a register, the address of the device must be in the low-order 2 bytes of the register.

**ERROR**

Specifies the address of an error routine that is to receive control if an error in the GENIO macro occurs.

If you omit this parameter, control will return to the instruction immediately following the GENIO instruction, just as it would were there no error. In such a case you should analyze the return code before proceeding further.

**Usage**

1. It is an error if you enter the GENIO macro with the START, STARTR, HALT, MODIFY, or CLOSE parameter specified before the device has been opened.
2. Only an authorized supervisor state program can issue the GENIO macro with the STARTR parameter specified. This allows an authorized program to use real channel programs to control real I/O devices directly. The CP channel program translation, which is a necessary middle step when using a *virtual* channel program, is bypassed.

An unauthorized program must use the START parameter.

3. If you specify EXITBR=YES, your interrupt handler calls your exit routine. This means that there may not be an active task when the branch takes place. Therefore, be certain that your exit routine contains no supervisor calls.

You can issue branch entries to GETMAIN and FREEMAIN for subpools for persistant private storage only and you can issue a Name/Token CREATE for level=private.

Your exit routine is called in key 0, in supervisor state, and disabled for interrupts. It must remain disabled for interrupts and must return control in supervisor state and key 0.

Your exit routine is responsible for saving and restoring registers in the save area whose address it receives in register 13.

4. The exit routine receives control in the same state and key as the program that opened the device. If the program is authorized, then the exit is disabled, meaning it cannot be interrupted. If the program is unauthorized, then the exit routine is enabled. I/O requests can be issued only by an exit routine that is disabled. I/O interrupts are handled after the exit routine ends. If in XA mode, the exit will be run in the AMODE of the caller.
5. A distinction must be made between errors occurring in the GENIO macro and errors occurring during the I/O operation.

If an error is found in the GENIO macro before the I/O operation has actually been started, a return code is placed in register 15. If you specified an address through the ERROR parameter, then control is passed, along with the return code, to the routine at that address. If you specified no error routine address, then control is passed to the instruction immediately following the GENIO macro.

If an error results from an I/O operation that was initiated through the START or STARTR parameter, then the exit routine specified when the device was opened receives control. All I/O error recovery is the responsibility of the program that opened the device.

6. The CLOSE parameter completely cuts the program off from the device specified and makes the device generally available. This includes deactivating the exit routine, which cannot receive control resulting from an interrupt from a closed device.
7. GCS does not support program controlled interrupts (PCIs). If a task receives a PCI, then the interrupt is saved in the interrupt control block. However, it will not be passed to the task's exit until the I/O operation is complete. And, although the byte-count in the CSW is unpredictable when a PCI interrupt occurs, the byte-count is also passed to the task's exit.
8. If you specify BRANCH=YES, your task must be in supervisor state, key 0, and disabled for interrupts.

Input and output are performed in the key of the task that issued the GENIO OPEN instruction, **not** in the key of the caller.

An interrupt handler may use the branch interface to the GENIO START and STARTR service routines.

Before you branch to the GENIO service routine, register 13 must contain the address of a 72-byte register save area.

When you branch directly to the service routine, no trace entry for the macro is generated.

9. The GENIO macro passes the following information to the exit routine.

Register 0	The UWORLD parameter, as specified when the device in question was opened.
Register 1	The address of the interrupt control block, defined below.

### The Interrupt Control Block

0 (0)	Flag byte Synchronous Interrupt = 00 Asynchronous Interrupt = 01	X
1 (1)	Reserved	3X
4 (4)	Device address	F
8 (8)	Channel status word (CSW) (370 accommodation)	D
10 (16)	Sense bytes	8F
30 (48)	Reserved	4F
40 (64)	Subchannel status word (SCSW)	3F
4C (76)	End	

If there was a unit check and the sense data could not be obtained, then the first 2 bytes of the sense data will contain X'107E'

Although it may be a condition code 3 (DEVICE NOT OPERATIONAL), the condition code from the I/O operation will be in byte-0 of the interrupt control block's SCSW or CSW.

If the STARTR parameter was specified, then the CCW address in the channel status word will be a real address.

Program controlled interrupts (PCIs) do not result in the scheduling of a user's exit routine. Rather, the SCSW or CSW stored as the result of a PCI will be saved in the interrupt control block.

In the case of a format 0 CCW, the second word of the CSW will be loaded with the third word of the SCSW.

### Examples

The following three GENIO macros are issued by the same program, affecting the same device.

```
GENIO OPEN,DEV=(2),EXIT=GOODBYE
```

The program requests that a certain device be opened. The address of the device can be found in register 2. When an interrupt occurs on this device, the exit routine at the address associated with the label GOODBYE is to receive control.

```
GENIO START,DEV=(2),CCW=(3)
```

The program now asks that the device it just opened be started. Register 3 contains the address of the first CCW in the channel control program to be processed. If the device is not busy, then the I/O operation begins. When the operation is finished, the exit program at the address associated with the label GOODBYE receives control.

```
GENIO CLOSE,DEV=DEVADDR
```

The program no longer needs the device, it asks that it be closed. The address of the device can be found at the address associated with the label DEVADDR.

### Return Codes and ABEND Codes

When the GENIO macro completes execution, it passes to the caller a return code in register 15.

#### General Return Codes:

Hex Code	Decimal Code	Meaning
X'00'	0	Function completed successfully.
X'30'	48	An invalid function was requested.
X'34'	52	No device address was specified.

#### For the OPEN function:

Hex Code	Decimal Code	Meaning
X'04'	4	DASD devices cannot be opened as general I/O devices.
X'08'	8	The specified I/O device does not exist.
X'0C'	12	The specified device is already opened.
X'10'	16	You did not specify the address of an exit routine.

#### For the CLOSE function:

Hex Code	Decimal Code	Meaning
X'04'	4	The specified device is not opened.

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'08'	8	The program that closes a device must be the same as the one that opened it.
X'0C'	12	An I/O exit routine cannot issue the GENIO macro with the CLOSE parameter specified.

**For the CHAR function:**

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'08'	8	The device specified does not exist.
X'0C'	12	The data area for storage of the device characteristics was not specified.

**For the MODIFY function:**

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'04'	4	No I/O is active on the device.
X'08'	8	The device is not open or not operational.
X'0C'	12	The specified CCW address is not accessible to you.
X'10'	16	The specified CCW address does not fall on a doubleword boundary.
X'14'	20	No CCW could be found that corresponds with the specified address or device.
X'18'	24	The CCW is neither a TIC nor a NOP instruction.
X'1C'	28	The new address of the modified CCW TIC instruction is not accessible to you.
X'20'	32	The new address of the modified CCW TIC instruction does not fall on a doubleword boundary.
X'24'	36	DEVICE END and CHANNEL END have already occurred.
X'2C'	44	The modified CCW cannot be a NOP instruction with command chaining if it is the last CCW in a real channel control program.
X'38'	56	Since the I/O is queued, there is no reason to enter a GENIO MODIFY instruction.
X'3C'	60	No CCW address was specified.

**For the START or STARTR functions:**

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'04'	4	Your virtual machine is not authorized for real I/O.
X'08'	8	The specified I/O device is not open.

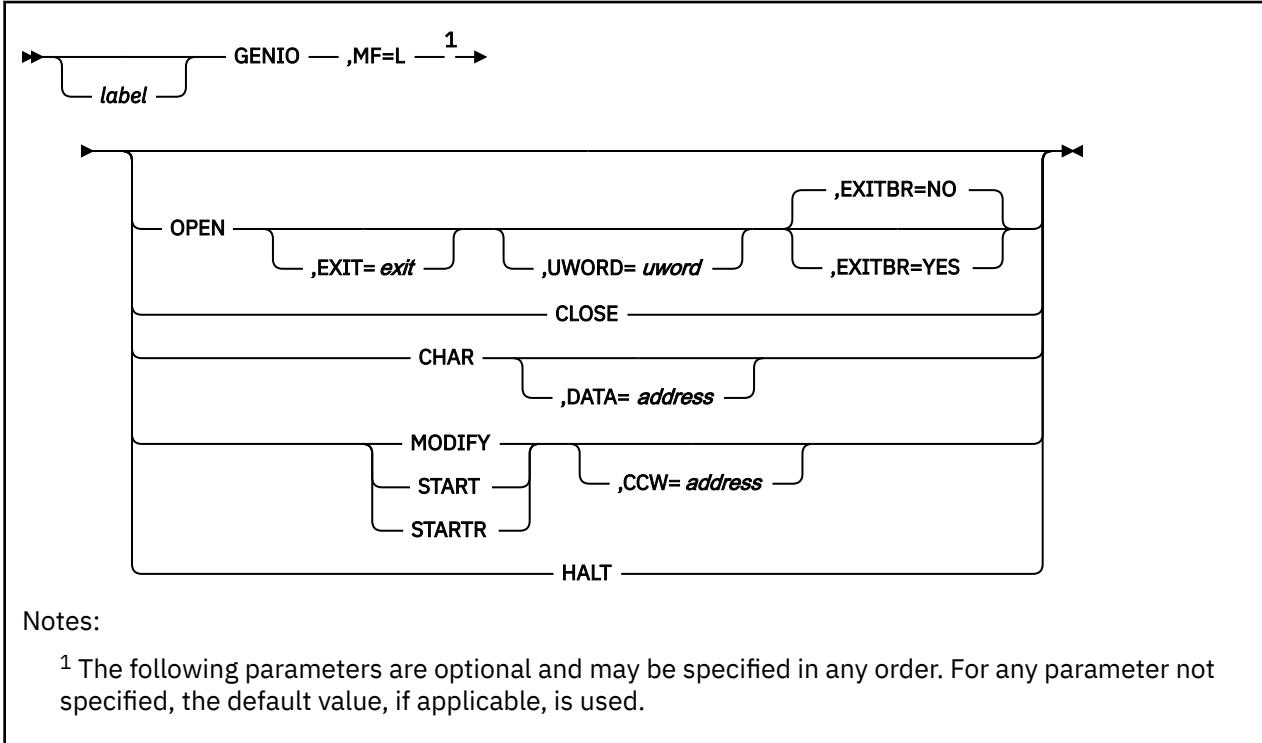
Hex Code	Decimal Code	Meaning
X'0C'	12	The specified I/O device is busy.
X'10'	16	Channel control word (CCW) address was not specified.
X'14'	20	You cannot perform real I/O functions while in key 0.
X'18'	24	A real I/O device is required for the STARTR function.

**For the HALT function:**

Hex Code	Decimal Code	Meaning
X'08'	8	The specified I/O device is not open.
X'0C'	12	The I/O activities of the device could not be halted.
X'10'	16	The specified device is not operational.

**ABEND Code Reason Code Meaning**

0F8	16	The GCS supervisor was called in access register mode.
FCA	0500	The specified parameter list is invalid.
FCA	0501	Your task is not authorized to perform real I/O functions.

**List Format**

## Purpose (List Format)

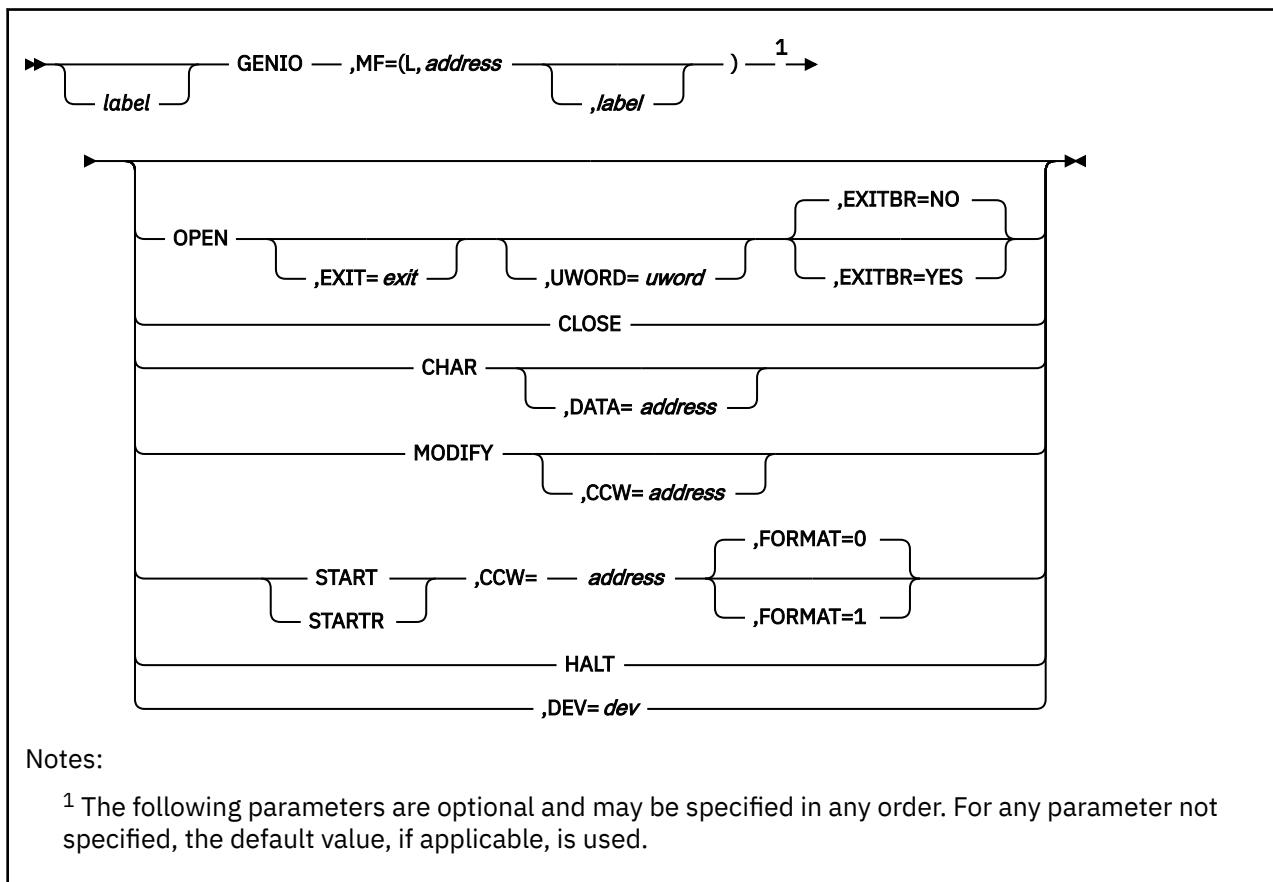
This format of the macro generates an in-line parameter list, based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation. Also, note that only the preceding parameters listed are valid in the list format of this macro.

## Added Parameter

### **MF=L**

Specifies the list format of this macro.

## List Address Format



## Purpose (List Address Format)

This format of the macro does not produce any executable code that starts the function. However, it does produce executable code that moves the parameter values that you specify into a certain parameter list. If you enter the macro using this format, then you must do so before any related invocation of the macro using the execute format.

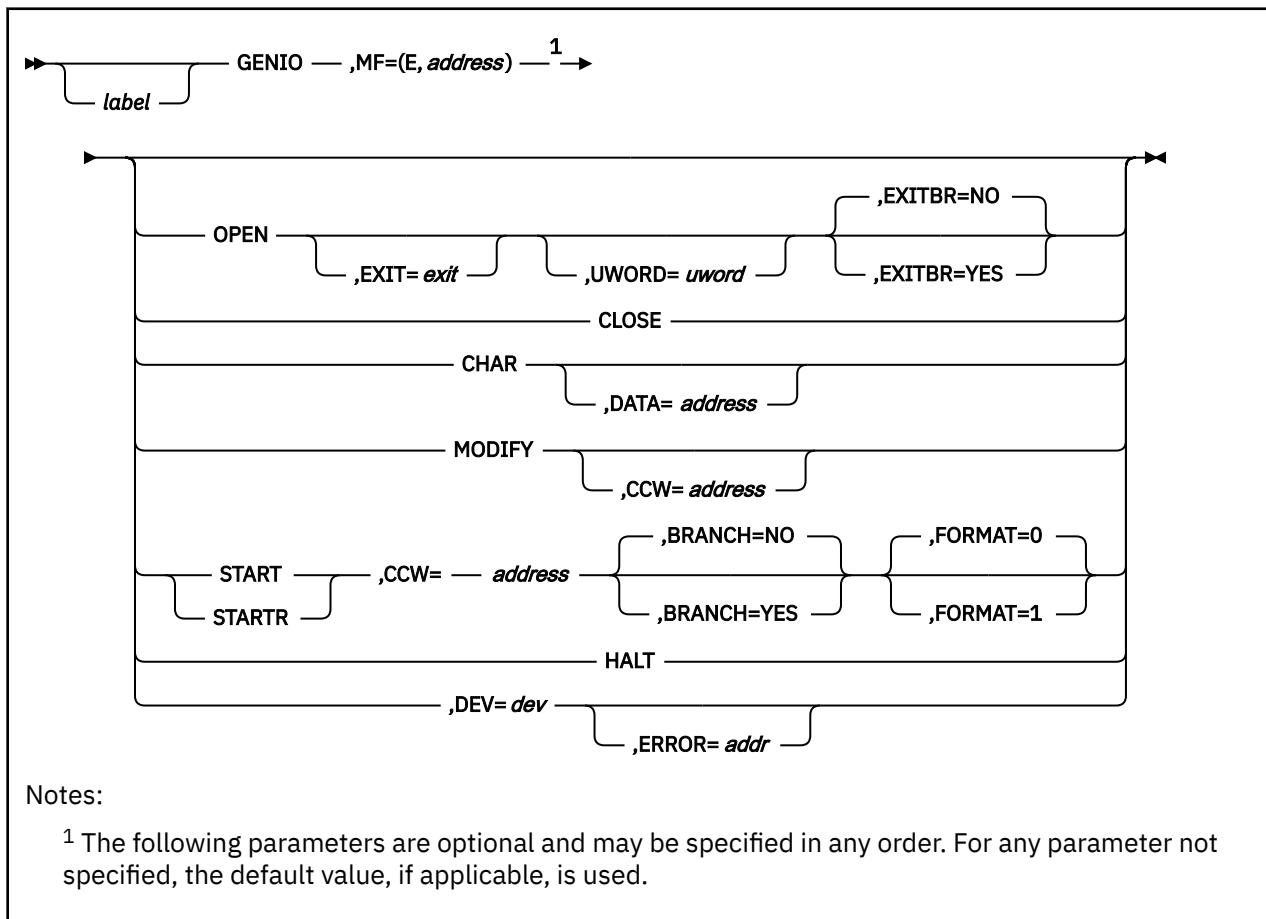
## Added Parameter (List Address Format)

### **MF=(L, address,label)**

*address* specifies the address of the parameter list where you want the parameter values placed. This address can be within your program or somewhere in free storage.

*label* is optional and is a user-specified label, indicating that you want to determine the length of the parameter list. The macro expansion equates the label you specify with the length of the parameter list.

## Execute Format



## Purpose (Execute Format)

This format of the macro generates code that runs the function, using a parameter list whose address you specify. Only the preceding parameters listed are valid in the execute format of this macro.

## Added Parameter (Execute Format)

### MF=(E, *address*)

*address* specifies the address of the parameter list used by the macro.

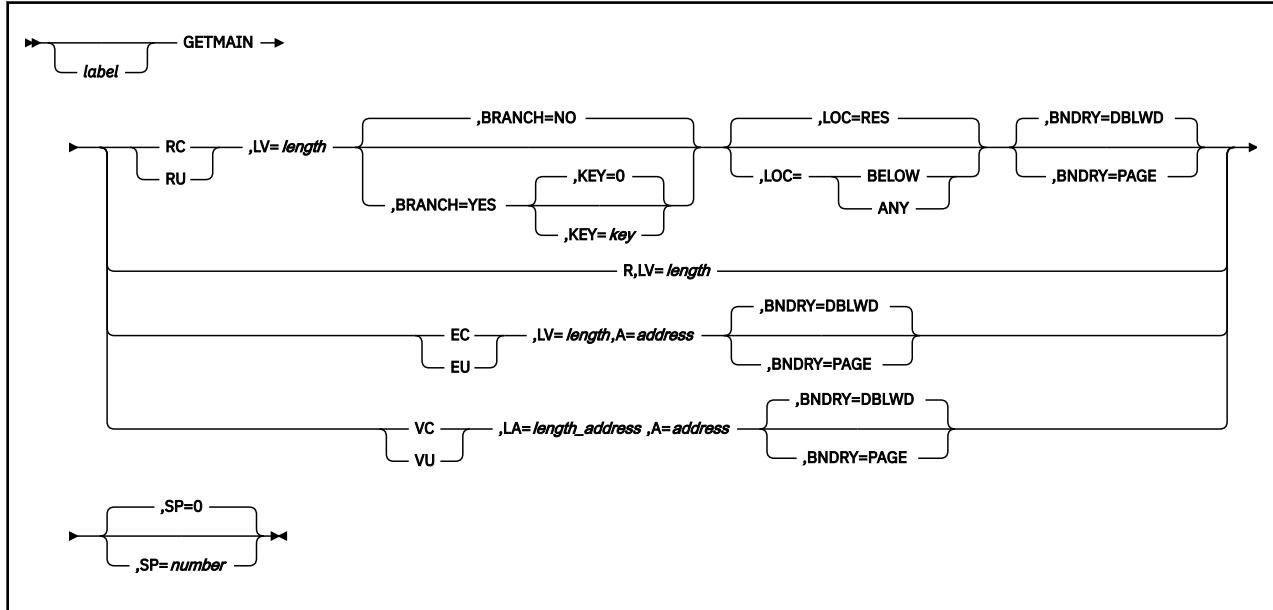
You can add or modify values in this parameter list by specifying them in this macro.

## GETMAIN

The GETMAIN macro is available in standard, list and execute formats.

### Standard Format

See also “List Format” on page 262 and “Execute Format” on page 263.



### Purpose

Use the GETMAIN macro to obtain a contiguous block of virtual storage.

The storage management function of GCS enables a task to dynamically obtain and free contiguous blocks of virtual storage as required.

The options R, VC, VU, EC, or EU can be used by callers in either 24-bit or 31-bit addressing mode. If one of these options is specified, storage area addresses and lengths will be treated as 24-bit addresses and values. The parameter list addresses and pointers to the length and address lists in the parameter lists (if present) will be treated as 31-bit addresses if the caller's addressing mode is 31-bit; otherwise, they will be treated as 24-bit addresses.

The options RU and RC can be used by callers in either 24-bit or 31-bit addressing mode. However, all values and addresses will be treated as 31-bit values and addresses.

### Parameters

#### RC

Indicates that your register request for storage is conditional. Your task will be able to continue, even if the storage you ask for is not immediately available.

Express the amount of storage you need in the LV parameter. If the storage is available, then you will receive its address in register 1. If it is not available, then you will receive a return code to that effect in register 15.

**RU**

Indicates that your register request for storage is unconditional. That is, your task will be unable to continue unless the storage you ask for is available immediately.

Express the amount of storage you need in the LV parameter. If the storage is available, then you will receive its address in register 1. If it is not available, then your task is abnormally terminated and you will receive an ABEND code.

**BRANCH**

Specifies whether your task should branch directly to the GETMAIN service routine.

**YES**

Specifies that your task should branch directly to the GETMAIN service routine.

**NO**

Specifies that you want to use the customary SVC interface. This option is the default.

**KEY**

KEY specifies the key which the storage is to be obtained; it is a key number from 0 through 15 or a register number (2) through (12) containing the key number in bits 24 through 27. The default is key 0.

**R**

Indicates that your register request for storage is unconditional.

Express the amount of storage you need in the LV parameter. If the storage is available, then you will receive its address in register 1. If it is not available, then your task is abnormally terminated and you will receive an ABEND code. Note that the BNDRY parameter cannot be used with the R parameter.

**EC**

Indicates that your request for storage is conditional.

Express the amount of storage you need in the LV parameter. If the storage is available, then you will receive its address in the word specified by the A parameter. If it is not available, then you will receive a return code to that effect in register 15.

**EU**

Indicates that your request for storage is unconditional.

Express the amount of storage you need in the LV parameter. If the storage is available, then you will receive its address in the word specified by the A parameter. If it is not, then your task is terminated abnormally and you will receive an ABEND code.

**VC**

Indicates that your request for a variable amount of storage is conditional.

Express the acceptable size range in the LA parameter.

If the storage is available, then you will receive the address of the storage block in the first word of the area specified by the A parameter. The second word of that area will contain the length of the storage block. If it is not available, then you will receive a return code to that effect in register 15.

**VU**

Indicates that your request for a variable amount of storage is unconditional.

Express the acceptable size range in the LA parameter.

If the storage is available, then you will receive its address in the first word of the area specified by the A parameter. The second word of that area will contain the length of the storage block. If it is not available, then your task is terminated abnormally and you receive an ABEND code.

**LV**

Specifies the length, in bytes, of the storage block you need.

This number should be a multiple of eight. If it is not, then GCS rounds it up to the nearest multiple of eight.

If the R parameter is used, then you can code LV=(0) as well. If it is, then the high-order byte of register 0 must contain the subpool number and the 3 low-order bytes must contain the length of the requested storage block.

You can write this parameter as an assembler program label or as register (2) through (12).

#### **LA**

Specifies the address of a two-word list that defines the acceptable size range into which the requested variable length storage block may fall.

The first word in the list must contain the minimum acceptable length of the block. The second word must contain its maximum acceptable length. These numbers should be multiples of eight. If they are not, then GCS rounds them up to the nearest multiples of eight.

On a variable request GETMAIN, if GCS cannot obtain the maximum size requested, it obtains the largest block of storage available that is greater than the minimum acceptable length.

You can write this parameter as an assembler program label or as register (2) through (12).

#### **A**

Specifies the address of a one or two word list.

If the EC, EU, VC, or VU parameter is specified, then GCS will store the address of the storage block in the first word of this list.

If the VC or VU parameter is specified, then GCS will store the length of the variable length storage block in the second word of this list.

You can write this parameter as an assembler program label or as register (2) through (12).

#### **LOC**

Specifies the location of the requested block of storage.

The location of virtual storage is allocated below or above the 16MB line as follows:

##### **BELOW**

Specifies that the requested virtual storage is to be allocated entirely below 16MB.

##### **ANY**

Specifies that the requested virtual storage can be located anywhere.

##### **RES**

Specifies that the location of the virtual storage requested is to be allocated based on the location of the requester. If the requester resides below 16MB, storage is to be allocated below 16MB, if the requester resides above 16MB, virtual storage may be located anywhere.

**Note:** This parameter can only be used with RC and RU. On all other forms, LOC=BELOW is used.

#### **BNDRY**

Specifies the boundary alignment of the requested storage block.

If you omit this parameter, then the block is aligned on a doubleword boundary, by default. In fact, you must omit this parameter if you use the R parameter.

Include one of the following with the BNDRY parameter.

##### **PAGE**

Indicates that the storage block is to begin on a 4KB page boundary.

##### **DBLWD**

Indicates that the storage block is to begin on a doubleword boundary. This option is the default.

#### **SP**

Specifies the subpool associated with the requested block of storage.

A subpool is a number from 0 to 255 that is assigned to a block of storage to describe its characteristics.

You can write this parameter as an assembler program label or as register (2) through (12). If the R parameter is used, then LV=(0) can be coded as well. If it is, then the high-order byte of register 0 must contain the subpool number and the 3 low-order bytes must contain the length of the requested storage block.

Subpool numbers are as follows:

**0**

Specifies private, fetch-protected storage. If the main task issued the GETMAIN macro, then GCS automatically frees the storage when the task ends. This is also true for a subtask that was attached to a main task with the SZERO=NO parameter specified in an ATTACH macro. See “[ATTACH](#)” on page [165](#).

However, if the subtask was attached with the SZERO=YES parameter specified (or defaulted), then GCS associates the storage with the oldest ancestor task with which this subtask is sharing the subpool. The storage block is not automatically freed by GCS when the subtask ends. The storage is freed only when the oldest ancestor task ends.

Any program can obtain storage from this subpool.

This option is the default.

**1 - 127**

Specifies private, fetch-protected storage. If the main task issued the GETMAIN macro, then GCS automatically frees the storage when the task ends. This is also true for a subtask that was attached to a main task without this subpool having been specified in the SHSPV or SHSPL parameter in the ATTACH macro.

However, if the subtask was attached with this subpool specified in the SHSPV or SHSPL parameter in the ATTACH macro, then GCS associates the storage with the oldest ancestor task with which this subtask is sharing the subpool. The storage is not automatically freed by GCS when the subtask ends. The storage is freed only when the oldest ancestor task ends.

Any program can obtain storage from these subpools.

**229**

Specifies private, fetch-protected storage. GCS will automatically free the storage when the task ends.

Only privileged programs can obtain storage from this subpool.

**230**

Specifies private, nonfetch-protected storage. GCS will automatically free the storage when the task ends.

Only privileged programs can obtain storage from this subpool.

**231**

Specifies common, fetch-protected storage. GCS does not free the storage when the task that acquired it ends. This storage must be explicitly freed by some privileged program.

Only privileged programs can obtain storage from this subpool.

**241**

Specifies common, nonfetch-protected storage. GCS does not free the storage when the task that acquired it ends. This storage must be explicitly freed by some privileged program.

Only privileged programs can obtain storage from this subpool.

**243**

Specifies private, fetch-protected storage. GCS does not free the storage when the task that acquired it ends. This storage must be explicitly freed by some privileged program.

Only privileged programs can obtain storage from this subpool.

**244**

Specifies private, nonfetch-protected storage. GCS does not free the storage when the task that acquired it ends. This storage is persistent in that it must be explicitly freed by some privileged program.

Only privileged programs can obtain storage from this subpool.

If you specify a subpool number that is not listed above or one which you are not authorized to use, and if your request was unconditional, then GCS will end your program abnormally. If your request were conditional, then you will receive a return code of 4.

<b>Subpool</b>	<b>Private</b>	<b>Fetch-protected</b>	<b>Privileged</b>	<b>Persistent</b>
0	X	X		
1 - 127	X	X		
229	X	X	X	
230	X		X	
231		X	X	X
241			X	X
243	X	X	X	X
244	X		X	X

**Usage**

1. GCS sets the key of the requested storage block to the PSW key of the task issuing the GETMAIN macro.
2. You can include BRANCH=YES only with the RC and RU parameters of the GETMAIN macro.

Your task must be in supervisor state, key 0, and disabled for interrupts when you use this service.

The macro destroys the contents of register 3 when BRANCH=YES. You may want to save and, later, restore the contents of register 3.

Before branching directly to the GETMAIN service routine, register 13 must contain the address of a 72-byte save area. You can obtain this save area by using the GCSSAVE macro.

The GCSSAVI macro must be used in place of the GCSSAVE macro to obtain the save area if the branch entry to GETMAIN is from an exit defined by GENIO with EXITBR=YES, or from an exit defined by IUCVCOM with BRANCH=YES.

The CVT mapping macro must be assembled as a DSECT into your program.

**Examples**

```
GETMAIN RU,LV=(5),SP=0,BNDRY=PAGE
```

The task requests a certain amount of storage space. This amount has previously been stored in register 5. If the task cannot get the storage, it will not continue processing, because this is an unconditional request. Furthermore, the task requests that the subpool number 0 be assigned to the storage and that it begin on a page boundary.

```
GETMAIN EC,LV=STORE,A=BLOCK
```

The task requests a certain amount of storage space. This amount is stored at the address associated with the label STORE. The address of the storage space is to be stored at the address associated with the label BLOCK.

## Return Codes and ABEND Codes

When this macro completes processing, it passes to the caller a return code in register 15.

**For CONDITIONAL requests only:**

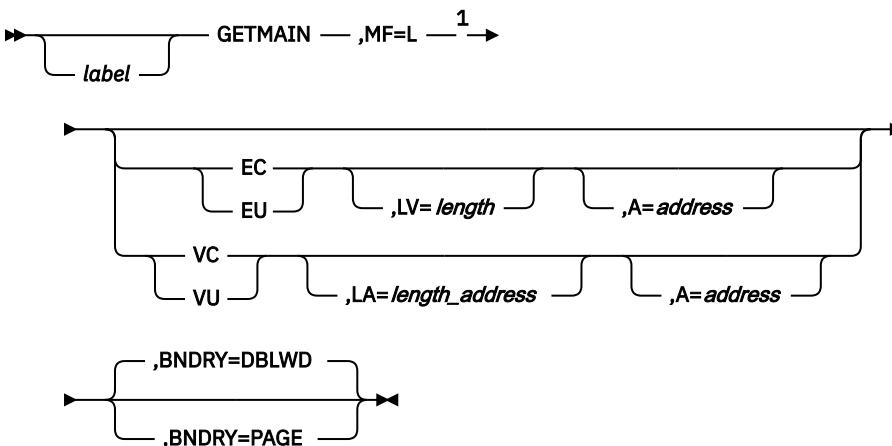
Hex Code	Decimal Code	Meaning
X'00'	0	Function completed successfully.
X'04'	4	Function was not completed.

When BRANCH=YES, the following additional information is returned:

- Register 0 contains the number of bytes in the storage block obtained.
- Register 1 contains the address of the storage block obtained.
- Register 15 contains a return code for the conditional call.

ABEND Code	Meaning
0F8	The GCS supervisor was called in access register mode.
604	Either an invalid address was specified in the A or LA parameter, or the macro itself received an invalid parameter list address.
704	An irrecoverable machine, system, or other error occurred while processing the GETMAIN macro.
804	Either there was insufficient virtual storage to execute the GETMAIN macro, or the LV parameter specified zero or a negative number.
B04	A GETMAIN macro contained an error in the specification of the subpool.
70A	An irrecoverable machine, system, or other error occurred while processing the GETMAIN macro with the R parameter specified.
80A	Either there was insufficient virtual storage to execute the GETMAIN macro with the R parameter specified, or a length of zero was specified.
B0A	A GETMAIN macro, with the R parameter specified, contained an error in the specification of the subpool.
778	An irrecoverable machine, system, or other error occurred while processing the GETMAIN macro with the RU parameter specified.
878	Either there was insufficient virtual storage to execute the GETMAIN macro with the RU parameter specified, or the LV parameter specified a zero or a negative number.
B78	A GETMAIN macro, with the RU parameter specified, contained an error in the specification of the subpool.
E04	A GETMAIN macro specified a parameter that GCS does not support.

## List Format



Notes:

- <sup>1</sup> The following parameters are optional and may be specified in any order. For any parameter not specified, the default value, if applicable, is used.

## Purpose (List Format)

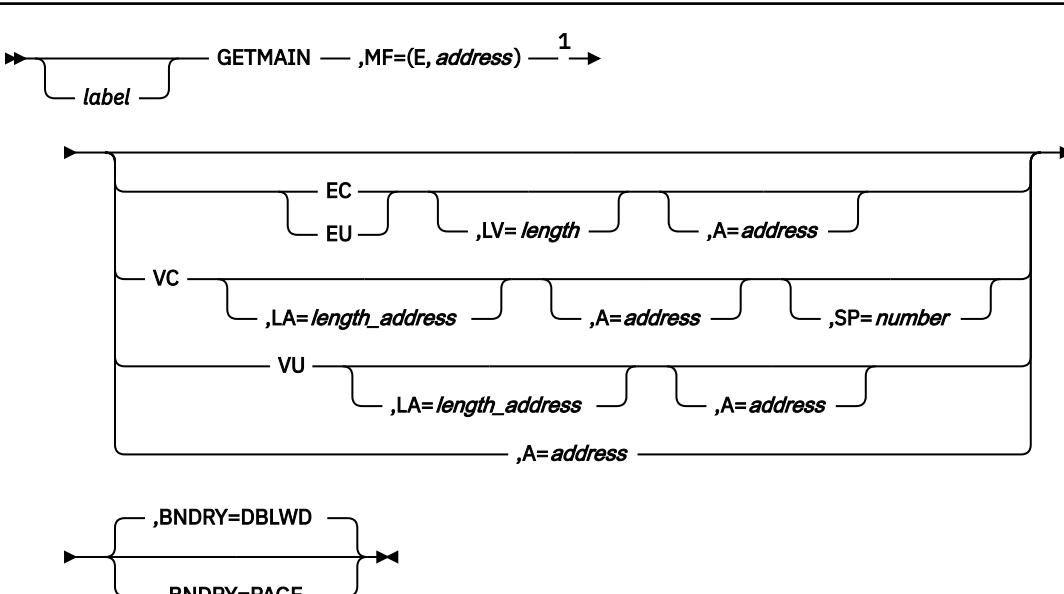
This format of the macro generates an in-line parameter list based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation. Also, note that only the preceding parameters listed are valid in the list format of this macro.

## Added Parameter

### MF=L

Specifies the list format of this macro.

## Execute Format



Notes:

<sup>1</sup> The following parameters are optional and may be specified in any order. For any parameter not specified, the default value, if applicable, is used.

## Purpose (Execute Format)

This format of the macro generates code that executes the function, using a parameter list whose address you specify. Only the preceding parameters listed are valid in the execute format of this macro.

## Added Parameter (Execute Format)

### **MF=(E, *address*)**

*address* specifies the address of the parameter list to be used by the macro.

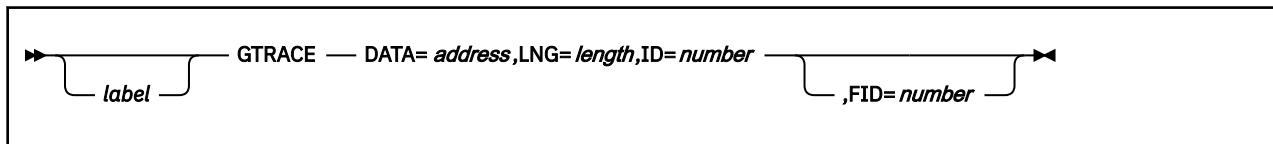
You can add or modify values in this parameter list by specifying them in this macro.

# GTRACE

The GTRACE macro is available in standard, list and execute formats.

## Standard Format

See also “List Format” on page 266 and “Execute Format” on page 267.



## Purpose

Use the GTRACE macro to record user data in the GCS trace table.

Sometimes, you will need certain user data recorded for you in the GCS trace table. Any type of data you wish can be recorded in the GCS trace table, for example an instruction or the result of some calculation.

## Parameters

### DATA

Specifies the address in your virtual storage where the data you want recorded begins.

You can write this parameter as an assembler program label or as register (2) through (12).

### LNG

Specifies the number of bytes to be recorded starting at the address you specified in the DATA parameter.

You can write this parameter as any decimal number from 1 to 8192, as a hexadecimal number from X'00' to X'2000', or as register (2) through (12).

### ID

Specifies an identifier you want associated with the recorded data, which you can use for documentation purposes.

This identifier will be recorded along with the specified data to make it easier for you to find a trace entry on a terminal screen or in a printed dump. Valid identifier values are as follows:

0 through 1023	FOR GENERAL USERS
1024 through 4095	FOR IBM USE ONLY

### FID

Specifies the last two characters in the name of one of your formatting routines.

A formatting routine processes the externally traced data for printing. Because you define the data to be recorded by the trace facility, it is your responsibility to provide any routine that may be required to interpret and format it.

For more information on formatting and printing trace data, see the TRACERED utility in the [z/VM: CP Commands and Utilities Reference](#).

Each formatting routine must have a name that is eight characters long. The first six of these characters must be:

CSIYTX

The last two characters of the name can be any two-digit hexadecimal number from X'00' to X'FF'. These last two characters must be used as follows:

X'00'	The data is to be dumped in hexadecimal form (IBM USE ONLY).
X'01' through X'50'	FOR GENERAL USERS
X'51' through X'FF'	FOR IBM USE ONLY

Because the first six characters of the routine's name are known, you need only specify the last two characters in the FID parameter. If you omit this parameter, GCS assumes X'00', by default.

See “[Coding User Formatting Routines](#)” on page 267 for more information about the FID parameter and how it is used when you define your own formatting routines.

## Usage

For the information given to the GTRACE macro to be recorded, you must have previously issued the ETRACE or ITRACE commands. See “[ETRACE](#)” on page 73 and “[ITRACE](#)” on page 108.

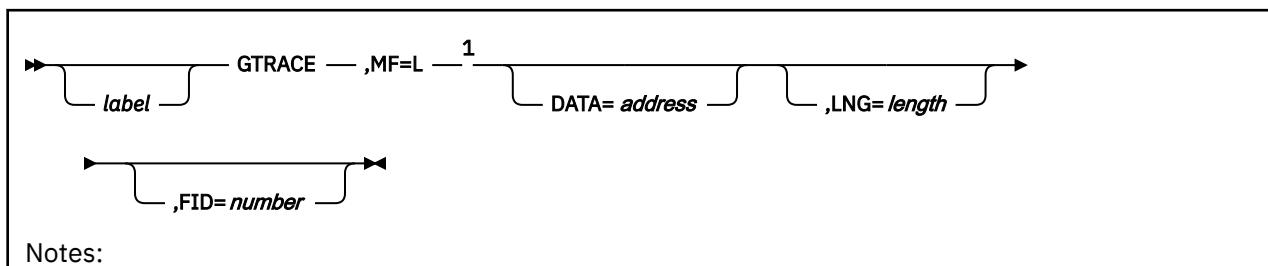
## Messages

The GTRACE macro generates no ABEND codes.

When this macro completes processing, it passes to the caller a return code in register 15.

Hex Code	Decimal Code	Meaning
X'00'	0	Function completed successfully.
X'04'	4	The GTRACE facility (monitor class 14) is not enabled.
X'08'	8	You specified an invalid value for the LNG parameter. It was less than 1 or greater than 8192.
X'0C'	12	You specified an invalid address for the DATA parameter.
X'10'	16	You specified an invalid value for the FID parameter. It was less than 0 or greater than 255.
X'14'	20	You specified an invalid value for the ID parameter. It may have been less than 0 or greater than 4095. Or, you may have specified a value from X'01' to X'50' for the FID parameter. This requires that the value specified for the ID parameter be from 0 to 1023.
X'1C'	28	Invalid parameter list address.

## List Format



<sup>1</sup> The following parameters are optional and may be specified in any order. For any parameter not specified, the default value, if applicable, is used.

## Purpose (List Format)

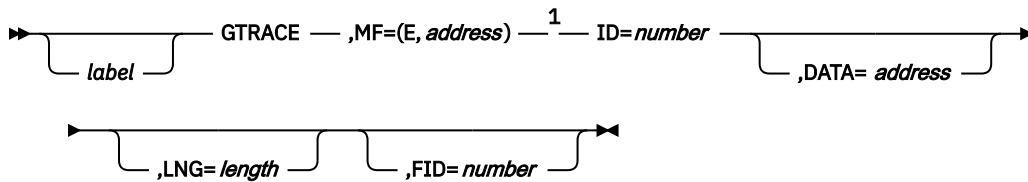
This format of the macro generates an in-line parameter list based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation.

### Added Parameter

#### MF=L

Specifies the list format of this macro.

### Execute Format



Notes:

<sup>1</sup> The following parameters may be specified in any order.

## Purpose (Execute Format)

This format of the macro generates code that executes the function, using a parameter list whose address you specify.

### Added Parameter (Execute Format)

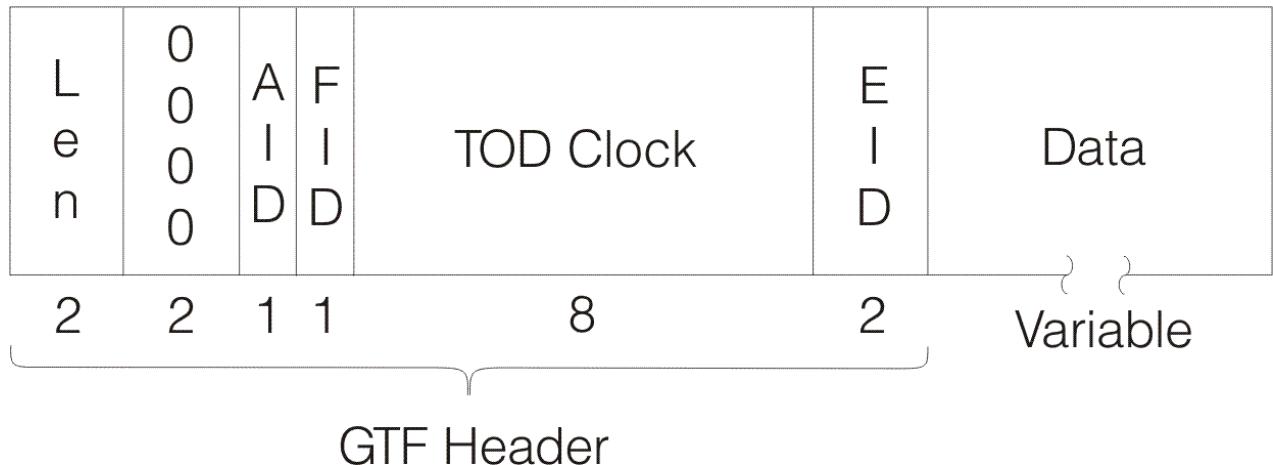
#### MF=(E, address)

*address* specifies the address of the parameter list to be used by the macro.

You can add or modify values in this parameter list by specifying them in this macro.

## Coding User Formatting Routines

Dump viewing facility formatting routines are invoked to format the header portion (including the GTF header) of GTRACE entries. But another formatting routine, one supplied by the user, is called to finish the data portion of the record. The format for the GTF header and data looks like this:



where:

**Len**

Contains the length of the entry, including the GTF header.

**0000**

Is a reserved field of the GTF header.

**AID**

Always contains X'FF', indicating that this is a data record.

**FID**

(Format ID) identifies the formatting module used for this record.

**TOD Clock**

Tells when the record was built, in time-of-day format.

**EID**

Contains information from the GTRACE macro's ID parameter.

**Data**

Contains the internal trace entry without the internal header (up to 256 bytes).

The 1-byte FID field shown in the GTF header locates the user-supplied formatting routine. The routine's name must be CSIYTxX, with "xx" being the same two characters that were specified in the GTRACE macro's FID parameter. Also, the routine must have a file type of TEXT.

If the GCS program cannot find a user-supplied routine to handle data formatting, the data portions of the records get printed unformatted, in hexadecimal. If the program does find a CSIYTxX TEXT, it calls that routine. At that time, the registers will contain:

**R15**

The CSIYTxX routine's entry point

**R14**

The return address

**R13**

A 72-byte save area

**R1**

A parameter list with the following format:

**Bytes**

**Content**

**0-3**

The address of the trace record, a standard VS1 GTF prefix followed by the trace data

**4-7**

The address of an output buffer, cleared to blanks before the call. The buffer is 80 bytes if the output is to be displayed at the terminal and 132 bytes if the output is to be printed. CSIYTXxx puts the formatted trace entry here.

**8-11**

All zeros. (Not used with GCS, but put here to maintain compatibility with VS1. In VS1, this shows the address of GTF options in effect.)

**12-15**

The address of the GTF Event Identifier (EID)

**16-19**

The address of the trace record's data portion

**20-23**

The address of the end of the trace record's data portion + 1

**24**

A flag for the following options and information:

**X'40'**

Format the record for display on a printer

**X'80'**

Format the record for display on a terminal

**25**

A flag byte for trace format processing:

**X'01'**

Do not reload the formatting module for the next entry. If this bit is off, the formatting module will be reloaded on each trace entry.

**26-30**

Reserved for future use

**31**

A byte containing the GCS type code

**32-63**

32-byte work area for use by the called formatting routine

When formatting the GTRACE entries, user routine CSIYTXxx should fill the output buffer at the address found in bytes 4 through 7 in the input parameter list (the address of this input parameter list is in register 1). Then it should return to the calling routine with one of the following return codes in register 15:

**RC****Description****0**

The user has printed the buffer and continues processing on the same GTRACE record.

**4**

The user has printed the buffer and is finished processing the GTRACE record.

**8**

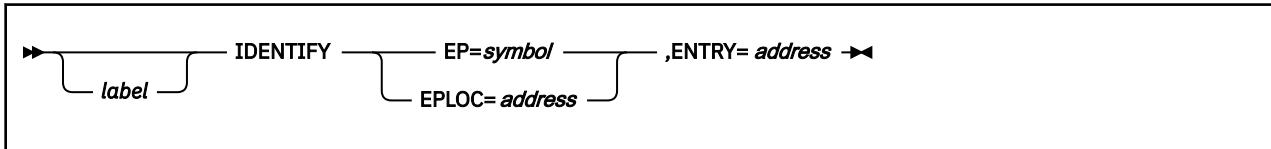
Do not print the buffer, and the GTRACE record is done.

**Other**

Print the record in hexadecimal.

## **IDENTIFY**

### **Format**



### **Purpose**

Use the IDENTIFY macro to define an entry point within a load module.

At times you may find it necessary to add an entry point to a load module where none had previously existed.

### **Parameters**

#### **EP**

Specifies the name by which you want the entry point to be known. It is this name that you will use in your program to refer to the entry point.

This name need not correspond to any name or symbol within the load module, though it can. It must not, however, correspond to any name, alias, or entry point name known to the system.

This name can be up to 8 bytes long.

#### **EPLOC**

Specifies the address where you have stored the name of the entry point in your program.

This name can be up to 8 bytes long. If it is less than 8 bytes long, it must be padded on the right with blanks.

You can write this parameter as an RX-type address, as register (0), or as register (2) through (12).

#### **ENTRY**

Specifies the address within the load module of the entry point you wish to identify.

You can write this parameter as an RX-type address or as register (1) through (12).

### **Usage**

1. The copy of the load module containing the entry point in question must be one of the following:
  - A copy of the load module for which your task previously issued a LOAD macro. See “[LOAD](#)” on page [298](#).
  - The last load module given control through the OSRUN command, or the ATTACH, LINK, or XCTL macro. See “[ATTACH](#)” on page [165](#), “[LINK](#)” on page [293](#), “[XCTL](#)” on page [373](#), or “[OSRUN](#)” on page [116](#).
2. The IDENTIFY macro cannot be issued by any asynchronous exit routine.
3. You cannot use the IDENTIFY macro to define an entry point that has been declared through the CONTENTS macro. See “[CONTENTS](#)” on page [197](#).
4. All Program Management Macros consider the code at entry points that are defined through the IDENTIFY macro to be reentrant. You must be certain that this code is in fact reentrant; otherwise, unpredictable results are possible.

## Examples

```
NAMEIT IDENTIFY EP=ABC3,ENTRY=(6)
```

Define a new entry point within a certain load module. The name of this entry point will be ABC3. The address of the entry point within the load module can be found in register 6. NAMEIT is the label on this instruction.

## Return Codes and ABEND Codes

When this macro completes processing, it passes to the caller a return code in register 15.

Hex Code	Decimal Code	Meaning
X'00'	0	Function successfully completed.
X'04'	4	The nonmain entry point name you specified is already assigned to the address.
X'08'	8	The entry point name you specified duplicates the name of a load module currently in virtual storage. The entry point name was not assigned to the address you specified.
X'0C'	12	The entry point address you specified is not within an eligible load module. The entry point name was not assigned to the address you specified.
X'10'	16	The IDENTIFY macro was issued by an asynchronous exit routine. The entry name was not assigned to the address you specified.
X'14'	20	An IDENTIFY macro was previously issued defining the same nonmain entry point name but at a different address. The entry point name specified in the present IDENTIFY macro was not assigned to the address specified.
X'18'	24	The parameter list was invalid. The entry point name was not assigned to the address you specified.
X'28'	40	The address specified by the EPLOC parameter is fetch-protected and the calling program is in a different key. Therefore, the calling program cannot access the storage. So, the entry point name was not assigned to the address that you specified.

## IHADVA

---

### Format

```
►► label — IHADVA ►►
```

### Purpose

Use the IHADVA macro to map the data returned by the DEVTYPE macro.

### Operands

#### IHADVA

Specifies that you want to map the data returned by the DEVTYPE macro.

### Response

The following information is placed into your area:

*Table 15. Supported Device Characteristics Information*

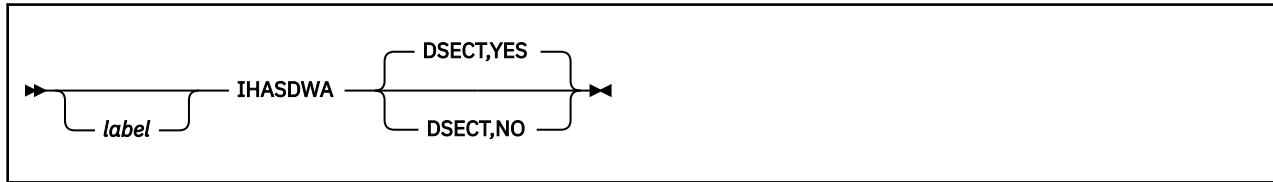
IBM Device	Device code (Hex)	Maximum Record (Dec)	DEVTAB (Hex)	RPS (Hex)
DUMMY	00000000	00000000	N/A	N/A
1403 Printer	10800808	120	N/A	N/A
2540 Reader	10000801	80	N/A	N/A
2540 Punch	10000802	80	N/A	N/A
Not Supported	00000000	0	00000000000000000000000000000000	00000000

For a detailed description of these fields, see the *MVS/DFP System Data Administration* book.

# IHASDWA

---

## Format



## Purpose

Use the IHASDWA macro to get a symbolic name for each field in the system diagnostic work area.

Often an application identifies an exit routine for each task that will receive control if the task terminates abnormally. See “[ESTAE](#)” on page 223.

When the ABEND macro is issued for a specific task, a system diagnostic work area (SDWA) is created. See “[ABEND](#)” on page 162.

The SDWA is an area of storage that contains important information about the task that has just terminated abnormally. (Study the following format of the SDWA.) The exit routine uses this information to analyze the problem.

Use the IHASDWA macro to produce a template of the system diagnostic work area that will make programming your exit routine much easier. The IHASDWA macro assigns symbolic names to each field of the template. Each symbolic name can be used as a displacement in an assembler language instruction in your exit routine to gain access to the corresponding field in the SDWA.

## Parameters

### DSECT

Indicates that you are about to specify whether the template produced will be a DSECT (dummy control section).

### YES

Indicates that the template will be created as a DSECT. If you omit the DSECT parameter altogether, then the template is produced as a DSECT. This is the default.

### NO

Indicates that the DSECT card should not be generated and that the SDWA should be a continuation of defined storage.

## Usage

1. To use the DSECT you have created to find your way around the SDWA, simply assign the address of the latter to a base register. Then, use the symbolic name of a field in the DSECT as the displacement to the corresponding field in the SDWA.
2. The template is created as part of the expansion of the IHASDWA macro as follows:

0	SDWAPARM	ESTAE parameter list address
4	SDWACMPF	Flags: 80 ---> Dump requested 40 ---> STEP parameter specified in ABEND instruction
5	SDWAREQ	Completion code (see note)
8	SDWACMPC	BC mode PSW at entry to ABEND macro
16(10)	SDWACTL1	Reserved
24(18)	SDWAGRSV	General registers 0-15 at entry to ABEND macro
88(58)	SDWANAME	Name of module that terminated abnormally
92(5C)		Reserved
96(60)	SDWAEPA	Entry point address of module that terminated abnormally
100(64)		Reserved
104(68)	SDWAEC1	Extended control PSW at time of error (ABEND)
200(C8)	SDWASPID	Number of the subpool containing SDWA
201(C9)	SDWALNTH	Length of SDWA (in bytes)
204(CC)		Reserved
232(E8)	SDWAFLGS	Flags
232(E8)		Reserved
234(EA)	SDWAERRC	Flags: 10 ---> Recovery routine percolated
235(EB)	SDWAERRD	Flags: 40 ---> State block associated with this ESTAE exit at time of error
236(EC)		Reserved
240(F0)	SDWARTYA	Address of recovery routine
244(F4)		Reserved
252(FC)	SDWARCDE	Return code from recovery routine: 0 ---> Continue with termination 4 ---> Retry using recovery at address in SDWARTYA
253(FD)		Reserved
368(170)	SDWAXPAD	Address of extention pointers SDWAPTRS
663(297)		END SDWA
0	SDWAPTRS	Extention pointers
4	SDWASRVP	Address of extention SDWARC1
0	SDWARC1	Extention
216(D8)	SDWAARER	Access registers 0-15 at entry to ABEND macro

**Note:** The SDWACMPC field contains the completion code specified in the ABEND macro. The SETRP macro may modify this field through its COMPCOD parameter.

3. Below the 16MB line, both BC mode and EC mode PSWs are filled in.

## Return Codes and ABEND Codes

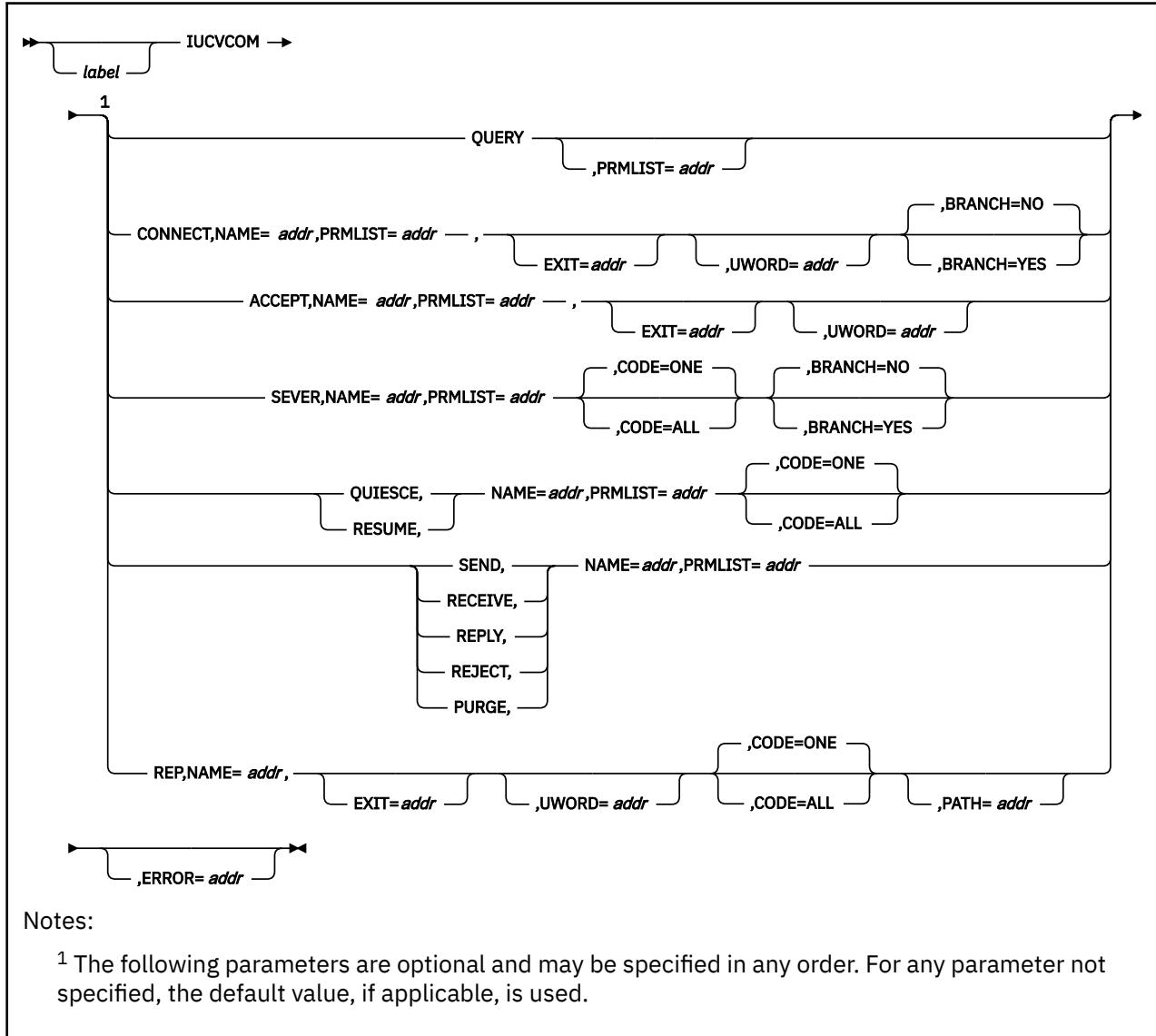
The IHASDWA macro generates no return codes and no abend codes.

# IUCVCOM

The IUCVCOM macro is available in standard, list, list address and execute formats.

## Standard Format

See also “List Format” on page 283, “List Address Format” on page 283 and “Execute Format” on page 284.



## Purpose

Use the IUCVCOM macro to coordinate communication among users within the IUCV and APPC/VM environment.

The Inter-User Communications Vehicle (IUCV) is a CP facility that allows a virtual machine to send information to or receive information from other virtual machines, a CP system service, or itself.

Advanced Program-to-Program Communication/VM (APPC/VM) is an application program interface (API) for communicating between two virtual machines that is mappable to the SNA LU 6.2 APPC interface and is based on IUCV functions. For more information on APPC/VM, see [z/VM: CP Programming Services](#).

By using the GCS IUCV support, communications can take place among several users. APPC/VM, used with the Transparent Services Access Facility (TSAF) virtual machine component, allows these communications to span several systems.

When the word *user* appears, you should take it to mean any supervisor or problem program.

This treatment of the IUCVCOM macro assumes that you are already familiar with the section dealing with IUCV in the [z/VM: CP Programming Services](#). For more information on IUCV, see “[IUCVINI](#)” on page 286.

## Parameters

### **QUERY**

Indicates that the user wants to know the size of the external interrupt buffer, and the maximum number of communication paths that can be established in the user's virtual machine.

GCS returns the size, in bytes, of the external interrupt buffer in register 0. It returns the maximum number of connections possible in register 1.

You can start the QUERY function without first having entered the IUCVINI SET instruction.

### **CONNECT**

Indicates that the user requests a communication path be established between it and the party the user is trying to communicate with.

The user must identify the party it wishes to communicate with through the CP IUCV or APPC/VM parameter list. The user should place the virtual machine identifier of the particular machine or CP system service it desires in the IPVMID field of the parameter list. Then, in the first 8 bytes of the IPUSER field, it should identify the particular user it wishes to contact in that machine.

Remember that all IUCV and APPC/VM users, including privileged ones, must use the IUCVCOM CONNECT instruction to establish an IUCV or APPC/VM path.

### **ACCEPT**

Indicates that the user wants to complete the communication path initiated by another party trying to communicate with it.

Remember that all IUCV and APPC/VM users, including privileged ones, must use the IUCVCOM ACCEPT instruction to complete an IUCV or APPC/VM path.

### **SEVER**

Indicates that the user wishes to terminate communication over the path in question.

The user cannot request that all paths into its virtual machine be severed by setting to 1 the IPALL bit in the CP IUCV or APPC/VM parameter list.

However, if the user enters an IUCVCOM SEVER instruction specifying the CODE=ALL parameter, then GCS issues an IUCV SEVER instruction for each of the user's paths, (both the IUCV and the APPC/VM paths). This happens because CP allows both types of paths to be severed regardless of its state.

If the user specifies CODE=ONE (or allows it to default), then only one specific path shall be severed. The path must be specified in the CP IUCV or APPC/VM parameter list.

Remember that all IUCV and APPC/VM users, including privileged ones, must use the IUCVCOM SEVER instruction to sever an IUCV or APPC/VM path.

### **QUIESCE**

Indicates that, while the user does not want the path in question severed, it does not wish to accept any incoming messages over it. Incoming communication over the path is temporarily suspended.

The user cannot request that all paths into its virtual machine be quiesced by setting to 1 the IPALL bit in the CP IUCV parameter list.

However, if the user enters an IUCVCOM QUIESCE instruction, specifying the CODE=ALL parameter, then GCS examines each of the user's paths to determine its current state. If a path is in a state which CP permits a quiesce to take place, then the path is quiesced. Otherwise, it is not. For example, CP does not permit a path to be quiesced if its owner has not completed a pending connection through an IUCVCOM ACCEPT instruction.

If the user specifies CODE=ONE (or allows it to default) then only one specific path shall be quiesced. The path must be specified in the CP IUCV parameter list.

Although incoming communication on the path in question is temporarily suspended, the user may still use the path to communicate out.

#### **RESUME**

Indicates that the user wants a quiesced path restored to full use.

The user cannot request that all paths into its virtual machine be resumed by setting to 1 the IPALL bit in the CP IUCV parameter list.

However, if the user enters an IUCVCOM RESUME instruction, specifying the CODE=ALL parameter, then GCS examines each of the user's paths to determine its current state. If a path is in a state which CP permits this function to take place, then the path is resumed; otherwise, it is not. For example, CP does not permit a path to resume if its owner has not completed a pending connection through an IUCVCOM ACCEPT instruction.

If the user specifies CODE=ONE (or allows it to default) then only one specific path shall be resumed. The path must be specified in the CP IUCV parameter list.

#### **SEND**

Send the user's message to the party at the other end of the specified path.

Presumably the party at the other end of the path has consented to communicate through the IUCVCOM ACCEPT instruction.

#### **RECEIVE**

Indicates that the user accepts the data that was passed through an IUCV or APPC/VM SEND function or through a connection parameter list extension on an APPC/VM path.

In all likelihood, there are other paths into the virtual machine that are owned by other users. Therefore, the RECEIVE function requires that the user identify the path through the IPPATHID field in the CP IUCV or APPC/VM parameter list.

#### **REPLY**

Conveys the user's response to a message sent to it by another party through the SEND function.

#### **REJECT**

Indicates that the user refuses to receive a specific message that some party sent to it through the SEND function.

The path in question must be identified in the IPPATHID field of the CP IUCV parameter list. Unless the user identifies the specific message it rejects, then the first message found on the path is rejected. The user can identify the message in question in the IPMSGID parameter of the CP IUCV parameter list.

#### **PURGE**

Indicates that the user wishes to terminate or cancel a specific message that it sent to another party. Whether the other party received the message or not, the message is canceled.

The path in question must be identified in the IPPATHID field of the CP IUCV parameter list. Unless the user identifies the specific message it wants to purge, then the first message found on the path is purged. The user can identify the message in question in the IPMSGID field of the CP IUCV parameter list.

#### **REP**

Indicates that the exit routine or the UWORD for the specified path (or all the user's paths that were set up under the current task) are to be changed.

If only one specified path's exit routine or UWORD are to be changed, then the REP function must be requested from the same task that established the path.

If you omit the CODE parameter (or specify CODE=ONE), then you can use the PATH parameter to identify the single path to be affected. If the user specifies CODE=ALL, then all the paths the user set up under the current task are affected.

Remember, that the IUCVCOM REP instruction cannot be entered by a privileged user.

## **EXIT**

Specifies the address of an exit routine that is to receive control when an IUCV external interrupt occurs on the path in question.

If you omit this parameter from the CONNECT or ACCEPT function (or if you specify it as a register containing zero), then the exit routine you specified in the IUCVINI SET instruction becomes the exit routine associated with this path and the AMODE will be the AMODE of that exit routine.

When an external interrupt occurs involving an unauthorized user, the exit routine gains control in the same state and key as the user. The exit runs enabled for all interrupts.

The exit will be run in the AMODE of the caller for the CONNECT, ACCEPT and REP parameters.

External interrupts can occur at any time after the IUCVINI or IUCVCOM macro completes execution. Sometimes they occur even before the user's program reaches its next executable statement. Therefore, a user must be ready to handle such interrupts whenever they occur.

When an external interrupt occurs involving an authorized user, the exit routine gains control in supervisor state in key 0. The exit routine is disabled and because of the way that the INTERRUPT HANDLER gives control to the EXIT, no SVC calls can be issued by the EXIT.

Upon entry to the exit routine, the registers contain the following:

<b>Register</b>	<b>Contents</b>
0	The UWORD.
1	Unpredictable.
2	The address of the external interrupt buffer.
3	The address of the external interrupt buffer extension for APPC Connection Pending or APPC Connection Complete. At other times the register content is unpredictable.
4 - 12	Unpredictable.
13	The address of a user save area when an external interrupt occurs involving an unauthorized user, or the address of the 72-byte register save area when an external interrupt occurs involving an authorized user.
14	The address to which control must be returned after the exit routine completes execution.
15	The address of the exit routine.

Upon return from the exit routine, register 15 must contain a return code of either 0 (normal completion) or 4 (error). (The latter, GCS will sever the path involved in the error.) Registers 0 through 14 must contain the same values they contained when the exit routine received control.

You can write this parameter as an assembler program label or as register (2) through (12). If you write it as a label, then the address associated with that label must be the address of the exit routine. If you write it as a register, then the register must contain the address of the exit routine.

## **UWORD**

Specifies a fullword that will be passed to the path's exit routine in register 0 whenever it receives control.

This fullword can contain anything you wish. If you omit this parameter, then the UWORD specified in the IUCVINI SET instruction is passed.

You can write this parameter as an assembler program label or as register (2) through (12). If you write it as a label, then the address corresponding to the label is passed as the UWORD. If you write it as a register, then the contents of that register are passed as the UWORD.

#### **PRMLIST**

Specifies the address of the CP IUCV or APPC/VM parameter list associated with the function the user wishes to perform.

Remember that every function in the IUCVCOM macro (except QUERY and REP) requires such a parameter list. You are expected to provide one yourself. The List Format of the IUCV and APPC/VM macro is a convenient way to create it.

When the PRMLIST parameter is specified with the QUERY function, the external interrupt buffer extension length is returned in the parameter list.

#### **CODE**

Indicates the scope of the IUCVCOM function that the user wishes to perform.

The IUCVCOM functions SEVER, QUIESCE, RESUME, and REP require that to one specific path or allowed to affect all the user's paths. If you omit this parameter altogether, then GCS assumes CODE=ONE, by default.

#### **ALL**

Indicates that the function will affect all paths owned by the user.

#### **ONE**

Indicates that the function will affect only one specific path.

**Note:** For the SEVER function, this path is the one specified in the CP IUCV or CP APPC/VM parameter list. For the QUIESCE and RESUME functions, this path is the one specified in the CP IUCV parameter list. For the REP function, this path is the one specified by the PATH parameter in the IUCVCOM macro.

#### **NAME**

Specifies the address of the name by which the user is known within the IUCV or APPC/VM environment.

This name corresponds exactly with the name the user declared for the user in the IUCVINI SET instruction.

You can write this parameter as an assembler program label or as register (2) through (12). If you write it as a label, then this eight-character name must be stored at the address associated with that label. If you write it as a register, then the address of the name must be stored in the register.

#### **PATH**

Identifies the specific path that is to have its exit routine or UWORD changed through the REP function.

The PATH parameter must never be included if CODE=ALL is specified. However, the PATH parameter must be included for the REP function if CODE=ONE is specified or allowed to default.

You can write this parameter as an assembler program label or as register (2) through (12). If you write it as a label, then the halfword at the address associated with that label must contain the path identifier. If you write it as a register, then the register must contain the address of the halfword where the path id is stored.

#### **ERROR**

Specifies the address of an error routine that is to gain control if an error is found in the IUCVCOM macro.

If you omit this parameter and an error occurs, then control returns to the instruction following the IUCVCOM instruction, just as it would were there no error.

You can write this parameter as an assembler program label or as register (2) through (12).

**BRANCH**

Specifies whether your task should branch directly to the IUCVCOM service routine.

**YES**

Specifies that your task should branch directly to the IUCVCOM service routine.

**NO**

Specifies that you want to use the customary SVC interface. This option is the default.

**Usage**

1. A CP IUCV or CP APPC/VM parameter list must be created for each IUCVCOM function that requires the PRMLIST parameter. [z/VM: CP Programming Services](#) can provide you with more information on this.
2. No user can enter the IUCVCOM macro before it is admitted to the IUCV or APPC/VM environment through the IUCVINI SET instruction. The IUCVCOM QUERY function is the only exception to this.
3. To ensure that no user tries to perform an IUCV or APPC/VM function on a path that another user established, each path is associated with the name of the user that created it. For a user to enter the IUCVCOM macro with the CONNECT or ACCEPT parameter specified is to establish a path and, thereby, ownership of it. For a user to attempt to process a function on a path that does not belong to it is an error.
4. For a function to be processed, the path it is to affect must be in the proper state. The following describes the possible path states.

**CONNECT ISSUED**

A user entered an IUCVCOM CONNECT instruction for a certain path. However, no CONNECT COMPLETE interrupt has yet occurred on that path.

**CONNECT PENDING**

This is the next logical progression from the CONNECT ISSUED state. The CONNECT PENDING interrupt has occurred on the path, though the path is not yet complete. The target user can enter two types of instructions:

- IUCVCOM RECEIVE if there was a connection parameter list extension specified by the CONNECT on a APPC/VM path. The user would remain in a CONNECT PENDING state.
- IUCVCOM ACCEPT which would complete the path and the path would become ACTIVE.

**ACTIVE**

This is the next logical progression from the CONNECT PENDING state. The target user has entered the IUCVCOM ACCEPT instruction, causing a CONNECT COMPLETE interrupt on the path. The path is now complete and communication over it is now possible.

**QUIESCED**

One of the users using an ACTIVE path has entered the IUCVCOM QUIESCE instruction. Therefore, that user will not receive incoming communication over the path, though he can communicate out. For APPC/VM, this path state is incorrect.

**SEVER IN PROGRESS**

One of the users using an ACTIVE or QUIESCED path has entered the IUCVCOM SEVER instruction. No communication over the path is possible. The only logical or useful thing for the other user to do is to enter the same instruction to sever *his half of the path*.

For APPC/VM, this path state will not be monitored.

**INACTIVE**

This state describes a null path. That is, a path that does not exist.

A SEND function cannot be processed if the path is in the CONNECT PENDING state. In a typical scenario, one user (the SOURCE) attempts to establish a connection with another user (the TARGET) through the IUCVCOM CONNECT instruction. This places the source user's *half* of the path in the CONNECT ISSUED state. When a CONNECT PENDING interrupt occurs on the target user's *half* of the path, it is placed in the CONNECT PENDING state. The target user then enters the IUCVCOM ACCEPT instruction, placing its *half* of the path in the ACTIVE state. When a CONNECT COMPLETE interrupt

occurs on the source user's half of the path, it too is placed in the ACTIVE state. Communication between the two users is now possible.

5. If a user loads the SEVER, QUIESCE, or RESUME function with CODE=ALL specified, then all paths associated with that user are affected. When the function ends error-free, the parameter list associated with the function contains data related to the last path it processed. If errors occur, then the data in the parameter list is associated with the path that was being processed when the last error occurred.
6. As with other macros in GCS, the IUCVCOM macro passes return codes in register 15. Other diagnostic information is available in the IPRCODE field of the appropriate CP IUCV or APPC/VM parameter list.
7. You can specify the BRANCH parameter only with the *standard* or the *execute* format of the IUCV macro, not with list or list address formats.
8. If you specify BRANCH=YES, your task must be in supervisor state, key 0, and disabled for interrupts.

You can issue branch entries to GETMAIN and FREEMAIN for subpools for persistant private storage only, and you can issue a Name/Token CREATE for level=private.

An interrupt handler cannot use the branch interface to the IUCVCOM service routine.

Because this method of invoking the IUCVCOM macro avoids the supervisor call, no trace entry for the macro is generated.

The IUCVCOM macro alters the following registers:

Register	Contents
1	The address of the parameter list, as generated by the list form of the IUCVCOM macro. This occurs whether BRANCH is used.
14	The caller's return address, if BRANCH=YES is specified.
15	The address of the IUCVCOM service routine), if BRANCH=YES is specified.

## Return Codes and ABEND Codes

When this macro completes processing, it passes to the caller a return code in register 15.

Hex Code	Decimal Code	Meaning
X'00'	0	Function completed successfully.
X'02'	2	An APPC/VM parameter list was passed as input to IUCVCOM, and the request function has completed immediately. The function complete information is contained in the parameter list. The user's path specific exit will not be driven because no interrupt is reflected to the virtual machine by CP.
X'03'	3	An APPC/VM SENDDATA or RECEIVE function was requested, and has completed immediately, but error information has been stored in the IPAUDIT field of the CP APPC/VM parameter list. The user's path specific exit will not be driven because no interrupt is reflected to the virtual machine by CP.
X'08'	8	The user entered an IUCVCOM macro before admitting itself to the IUCV or APPC/VM environment through the IUCVINI SET instruction.
X'0C'	12	The user does not own the path in question.
X'10'	16	Either the NAME parameter was not specified or its address is zero.

Hex Code	Decimal Code	Meaning
X'18'	24	Either the PRMLIST parameter was not specified or its address is zero.
X'1C'	28	The user cannot process the SEVER, QUIESCE, or RESUME function with the IPALL bit of the CP IUCV or APPC/VM parameter list set to 1.
X'20'	32	The path identifier was not specified in the CP IUCV or APPC/VM parameter list.
X'28'	40	The function name the user specified was not recognizable by GCS. Choose CONNECT, ACCEPT, SEVER, QUERY, QUIESCE, RESUME, SEND, RECEIVE, REPLY, REJECT, PURGE, or REP.
X'2C'	44	Invalid parameter list.
X'30'	48	The state of the path is inconsistent with the function the user requested. For example, the user may have entered an IUCVCOM SEND, RECEIVE, REPLY, REJECT, or PURGE instruction for a path before a CONNECTION COMPLETE interrupt occurred on it. Or, the user may have entered an IUCVCOM QUIESCE, RESUME, SEND, RECEIVE (where a connection parameter list extension is not specified in the CONNECT on an APPC/VM path), REPLY, REJECT, or PURGE instruction for a path that has been only partially completed. That is, a path upon which a PENDING CONNECT interrupt has occurred. In such a case, the user should enter an IUCVCOM ACCEPT or SEVER instruction instead.
X'34'	52	Either the task that entered the IUCVCOM REP instruction is not the same task that established the path in question, or the IUCVCOM REP instruction was entered by a privileged user.
X'38'	56	Invalid APPC/VM parameter list. WAIT=YES can only be specified by privileged IUCV users.  <b>Note:</b> Privileged users must enter the APPC/VM macro directly for synchronous SENDs and RECEIVES.
X'3C'	60	Invalid IUCV connect parameter list. CONTROL=YES can only be specified by the GCS supervisor.
X'4C'	76	An APPC/VM parameter list is not allowed as input on an IUCV SEVER, CODE=ALL.
X'CC'	204	An error occurred in obtaining storage to satisfy the IUCV request. 204 is the return code from the GETMAIN macro.
X'xxx'	1xxx	An error occurred. 'xxx' is the value in the IPRCODE field in the IUCV parameter list that describes the error. See the <a href="#">z/VM: CP Programming Services</a> for information about IUCV parameter lists.  <b>Note:</b> If a RETURN CODE of 1000 is passed this corresponds to a CONDITION CODE of 2, which means "no message found", had the function been entered directly through IUCV. These functions are PURGE, RECEIVE, REJECT, and REPLY.

ABEND Code	Reason Code	Meaning
------------	-------------	---------

0F8	16	The GCS supervisor was called in access register mode.
-----	----	--

---

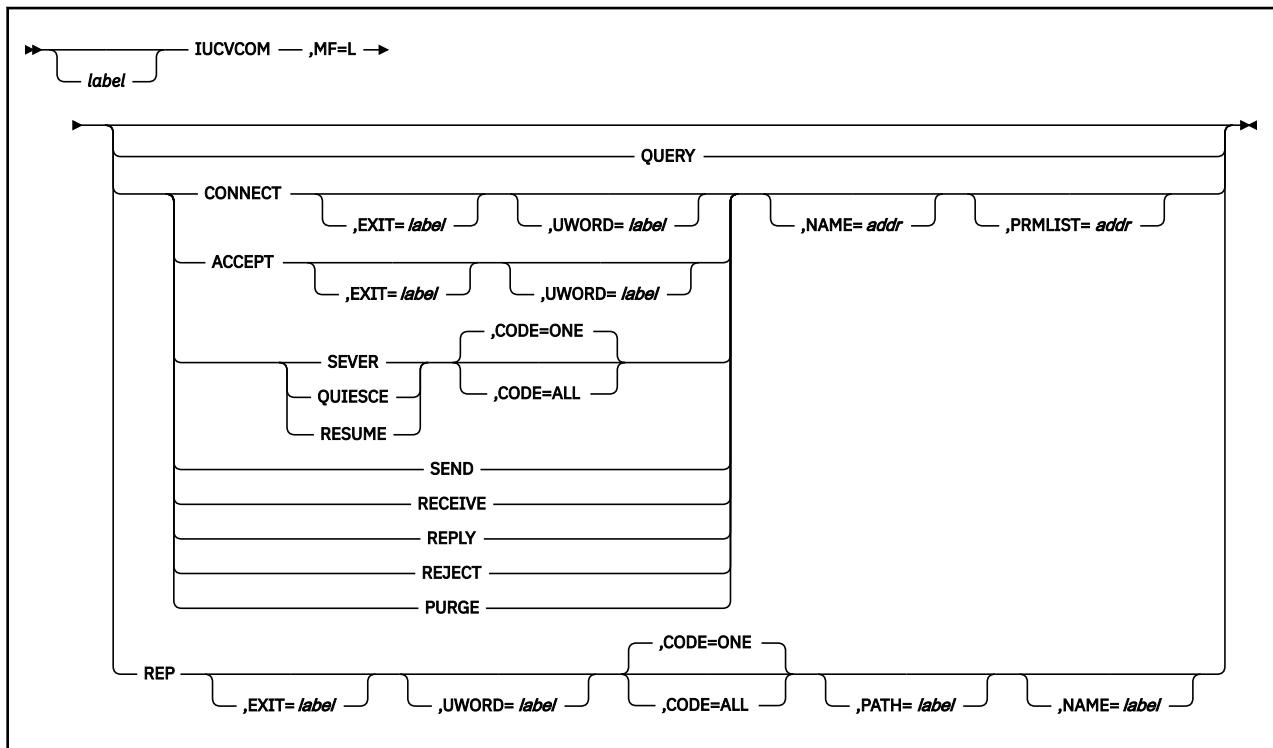
**ABEND Code Reason Code Meaning**


---

FCA	1101	A GETMAIN macro failed when GCS tried to obtain storage for the task that ended abnormally.
-----	------	---

---

## List Format



## Purpose (List Format)

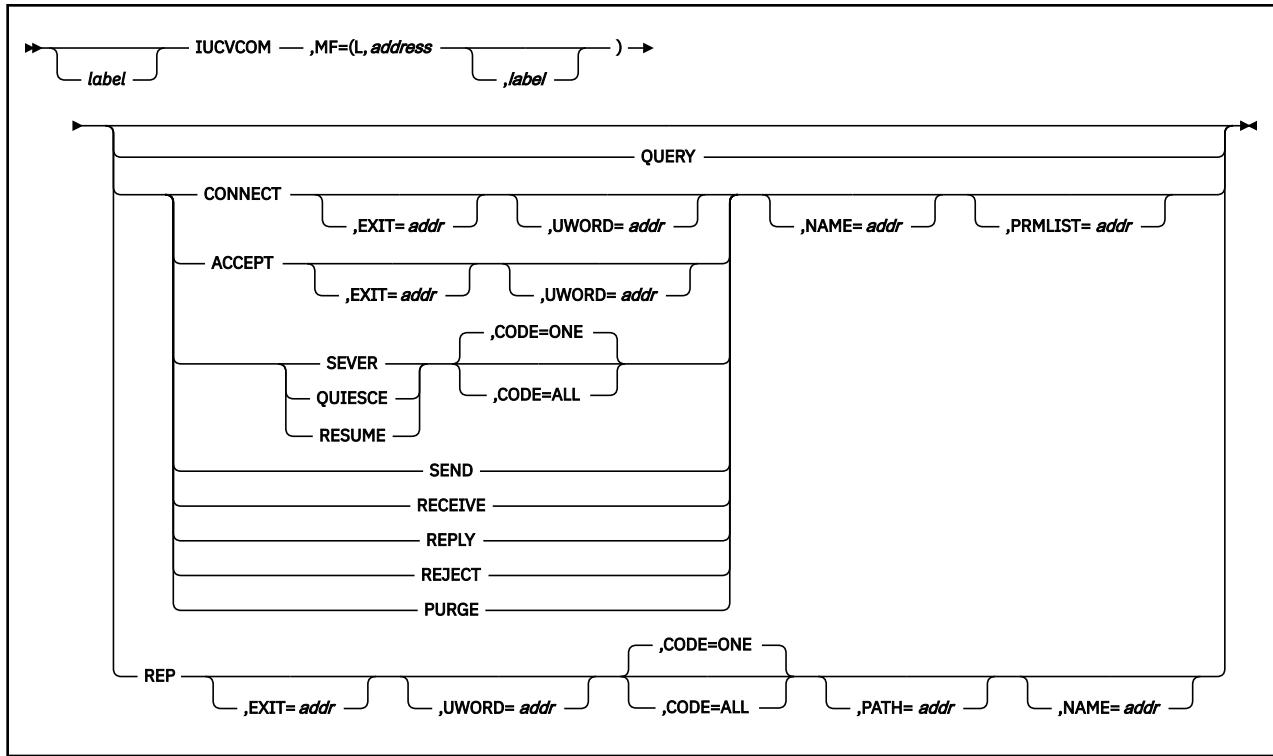
This format of the macro generates an in-line parameter list based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation. Only the preceding parameters listed are valid in the list format of this instruction.

## Added Parameter

### MF=L

Specifies the list format of this macro.

## List Address Format



### Purpose (List Address Format)

This format of the macro does not produce any executable code that runs the function. However, it does produce executable code that moves the parameter values that you specify into a certain parameter list. If you enter the macro using this format, then you must do so before any related invocation of the macro using the execute format. Only the preceding parameters listed are valid in the list address format of this macro.

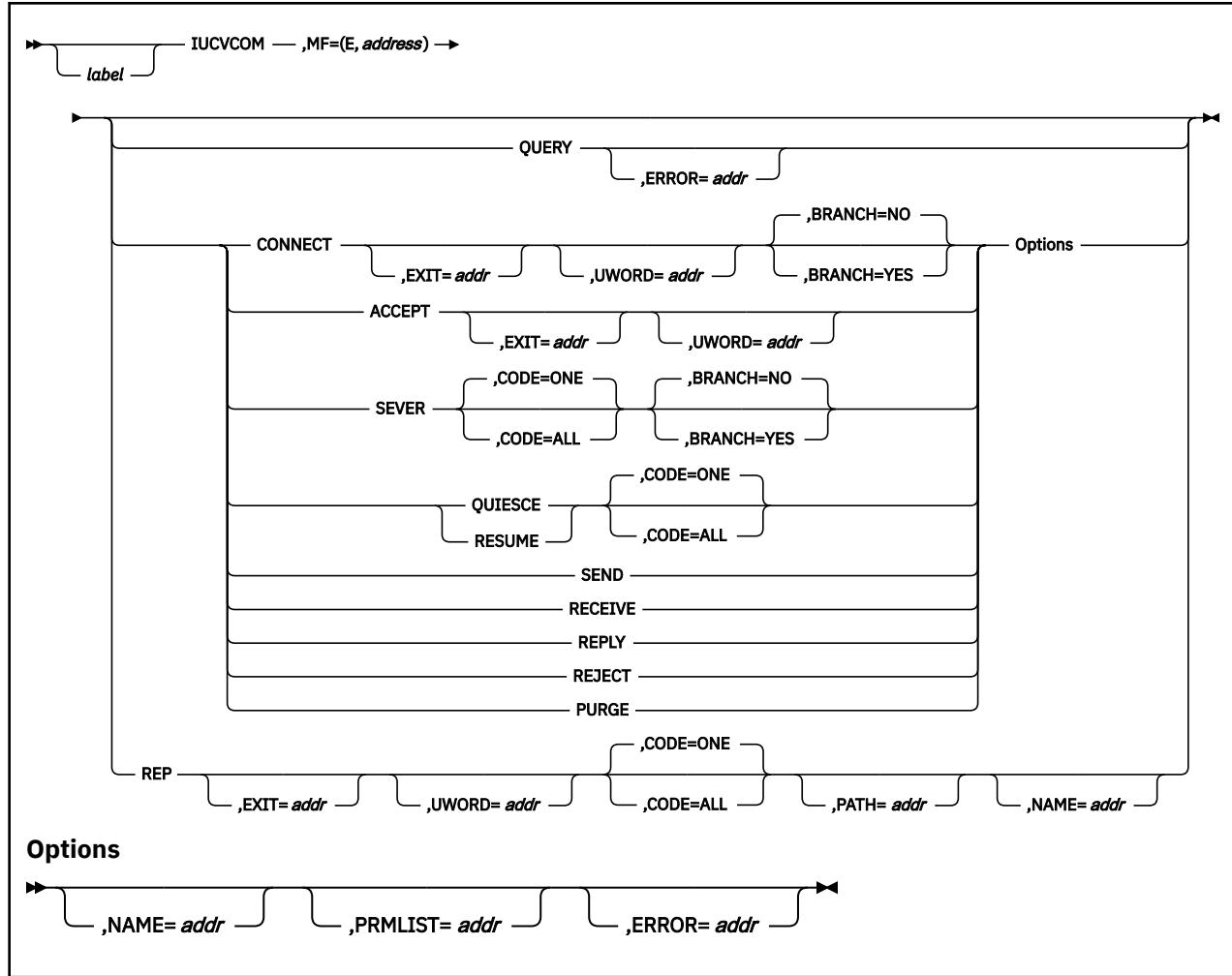
### Added Parameter (List Address Format)

#### **MF=(L, address ,label)**

*address* specifies the address of the parameter list into which you want the parameter values the user mention placed. This address can be within your program or somewhere in free storage.

*label* is optional and is a user specified label, indicating that you want to determine the length of the parameter list. The macro expansion equates the label you specify with the length of the parameter list.

### Execute Format



## Purpose (Execute Format)

This format of the macro generates code that executes the function using a parameter list whose address you specify. Only the preceding parameters listed are valid in the execute format of this macro.

## Added Parameter (Execute Format)

### **MF=(E, *address*)**

*address* specifies the address of the parameter list to be used by the macro.

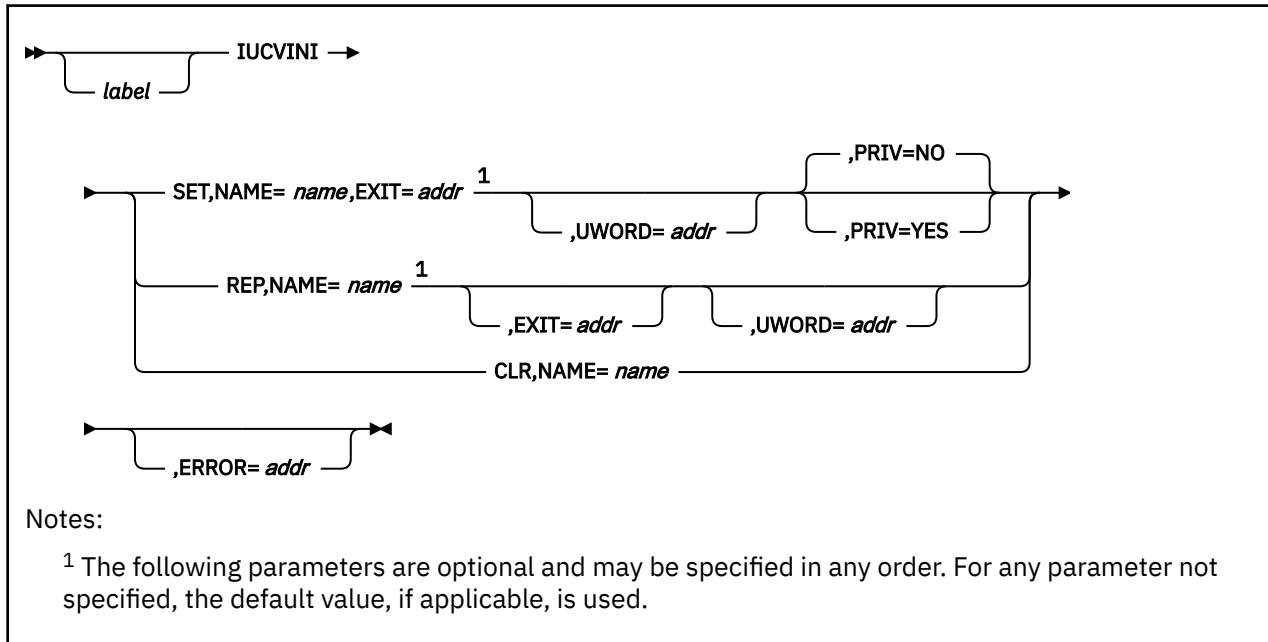
You can add or modify values in this parameter list by specifying them in this macro.

## IUCVINI

The IUCVINI macro is available in standard, list, list address and execute formats.

### Standard Format

See also “List Format” on page 290, “List Address Format” on page 291 and “Execute Format” on page 291.



### Purpose

Use the IUCVINI macro either to admit a user to or withdraw a user from the IUCV and APPC/VM environment.

The Inter-User Communications Vehicle (IUCV) is a CP facility that allows a virtual machine to send information to or receive information from other virtual machines, a CP system service, or itself.

Advanced Program-to-Program Communication/VM (APPC/VM) is an application program interface (API) for communicating between two virtual machines that is mappable to the SNA LU 6.2 APPC interface and is based on IUCV functions. For more information on APPC/VM, see [z/VM: CP Programming Services](#).

By using the GCS IUCV support, communications can take place among several users operating within several tasks operating within several virtual machines. APPC/VM, used with the Transparent Services Access Facility (TSAF) virtual machine component, allows these communications to span several systems.

When the word *user* appears, it should be taken to mean any supervisor or problem program.

This treatment of the IUCVINI macro assumes that you are already familiar with the section dealing with IUCV in the [z/VM: CP Programming Services](#). For more information on IUCV, see the “[IUCVCOM](#)” on page 275.

### Parameters

#### SET

Indicates that you want the user admitted to the IUCV or APPC/VM environment.

When you select this parameter, several things occur. First, an ID BLOCK is created for the user. This block contains the address of the user's general EXIT routine and the address of its general UWORD. Then, this block is associated with the NAME that identifies the IUCV user to GCS. Finally, the user receives permission to establish ownership of the IUCV or APPC/VM paths over which it will send and receive information.

The user must enter the IUCVINI SET instruction once before it attempts to send or receive information through the IUCV or APPC/VM facility. If the SET function completes successfully, then register 0 contains the number of possible IUCV or APPC/VM connections available to the user's virtual machine.

Note that the SET function also provides the PRIV parameter. This parameter allows a task running in supervisor state to establish and terminate a path through the IUCVCOM macro, but to communicate on that path using IUCV or APPC/VM directly, rather than using the GCS IUCV Support. If necessary, review the entry titled "[IUCVCOM](#)" on page 275.

#### **REP**

Indicates that you want to change the address of the user's general exit routine or its UWORD as they are recorded in the ID BLOCK.

This option is provided to allow the user to specify a new general exit routine and UWORD, depending on the situation at the moment. The general exit routine and UWORD specify the manner which the user responds to PENDING CONNECT interrupts (or all interrupts if the exit routine and UWORD are left to default on an ACCEPT or CONNECT function.) The REP function allows the user to change the manner of that response, whenever necessary.

The IUCVINI REP instruction can be issued only by the task that issued the original IUCVINI SET instruction. This function does not affect those paths that are already using the previous general exit routine as the path specific exit. These paths are recorded in the PATH BLOCK. To alter these, the IUCVCOM REP instruction must be used. Remember, though, that IUCVINI REP can never be entered by a user who specified PRIV=YES on an IUCVINI SET instruction.

#### **CLR**

Indicates that you want the user to be removed from the IUCV or APPC/VM environment.

When you select this parameter, the ID BLOCK is released and the user's IUCV or APPC/VM paths are severed.

#### **NAME**

Specifies the address of the symbolic name by which the user shall be known within the IUCV or APPC/VM environment. If the user is connecting to \*IDENT for resource identification, the NAME field must be equal to the resource name that is being identified.

This name was declared when the IUCVINI SET instruction was issued for the user. From that time to the time the IUCVINI CLR instruction is issued, this name must be consistently used to identify the user to GCS IUCV or APPC/VM.

For APPC/VM users, this name is the transaction program name (TPN). For IUCV users, this name corresponds to the first eight bytes of the IPUSER field in the IUCV parameter list.

The name must be eight characters long and can be any string of characters.

You can write this parameter as an assembler program label or as register (2) through (12). If you write it as a label, then the name must be stored at the address associated with that label. If you write it as a register, then the register must contain the address of the name.

#### **EXIT**

Specifies the address of the user's general IUCV or APPC/VM exit routine.

This general exit routine will receive control each time an IUCV or APPC/VM pending connect interrupt occurs on a path associated with this name. An IUCV or APPC/VM PENDING CONNECT interrupt occurs on a path when some other user enters a request to communicate with the user through the CONNECT function. CP then assigns the path to the user. The general exit routine is responsible for reacting to this request.

This exit routine is also considered the routine that receives control by default when any external interrupt occurs on a path for which the user has not established a path specific exit. This can happen under two sets of circumstances:

1. When a PENDING CONNECT interrupt had previously occurred on a path for which the user entered no IUCVCOM ACCEPT instruction.
2. When no exit routine was specified on the IUCVCOM CONNECT or IUCVCOM ACCEPT instruction that established the path.

When an external interrupt occurs involving an unprivileged user, the exit routine gains control in the same state and key as the user. Furthermore, the exit runs enabled for all interrupts if the user is running problem state.

The exit will be run in the AMODE of the caller for the SET or REP parameters.

External interrupts can occur at any time after the IUCVINI or IUCVCOM macro completes execution. Sometimes they occur even before the user's program reaches its next executable statement. Therefore, a user must be ready to handle such interrupts whenever they occur.

When an external interrupt occurs, involving a privileged user, the exit routine gains control in supervisor state, in key 0, and is disabled. The exit cannot issue any SVC calls. The only call allowed to the GCS supervisor is the branch entry to POST.

Upon entry to the exit routine, the registers contain the following:

<b>Register</b>	<b>Contents</b>
0	The UWORD.
1	Unpredictable.
2	The address of the external interrupt buffer.
3	The address of the external interrupt buffer extension for APPC Connection Pending or APPC Connection Complete. At other times the register content is unpredictable.
4-12	Unpredictable
13	The address of a user save area when an external interrupt occurs involving an unprivileged user, or the address of the 72-byte register save area when an external interrupt occurs involving a privileged user.
14	The address to which control must be returned after the exit routine completes execution.
15	The address of the exit routine.

Upon return from the exit routine, register 15 must contain a return code of either 0 (normal completion) or 4 (error). (The latter, GCS will sever the path involved in the error.) Registers 0 through 14 must contain the same values they contained when the exit routine received control.

You can write this parameter as an assembler program label or as register (2) through (12). If you write it as a label, then the exit routine must begin at the address associated with that label. If you write it as a register, then the register must contain the address of the exit routine.

## Defaults 1

### UWORD

Specifies a fullword that will be passed to the general exit routine in register 0, whenever the routine gains control. This parameter also specifies the value to be assigned to the UWORD parameter by default if none is specified on an IUCVCOM CONNECT or ACCEPT instruction

The UWORD can contain any type of information that you wish. But, if you omit this parameter, a value of zero is passed as the UWORD, by default.

You can write this parameter as an assembler program label or as register (2) through (12). If you write it as a label, then the UWORD must be stored at the address associated with that label. If you write it as a register, then the contents of the register are passed as the UWORD.

#### **PRIV**

Indicates whether the user will be privileged or nonprivileged. If you omit this parameter, then the user is considered nonprivileged, by default. This parameter is valid only when the user issuing this instruction is in supervisor state.

##### **NO**

Indicates that the user will be nonprivileged.

This means that the user must use the GCS Support Macros for all IUCV and APPC/VM activities.

##### **YES**

Indicates that the user will be privileged.

This means that the user has the authority to communicate over a path using IUCV or APPC/VM directly, rather than through the IUCVCOM macro. However, the user must establish and end the path using the IUCVCOM macro. This ensures a proper match between the GCS IUCV and APPC/VM path table and the CP IUCV and APPC/VM path table.

## **Defaults 2**

#### **ERROR**

Specifies the address of an error routine that is to gain control if an error is found in the IUCVINI macro.

If you omit this parameter and an error occurs, then control passes to the instruction following the IUCVINI macro, just as it would were there no error.

You can write this parameter as an assembler program label or as register (2) through (12). If you write it as a label, then the error routine must begin at the address associated with that label. If you write it as a register, then the register must contain the address of the error routine.

## **Return Codes and ABEND Codes**

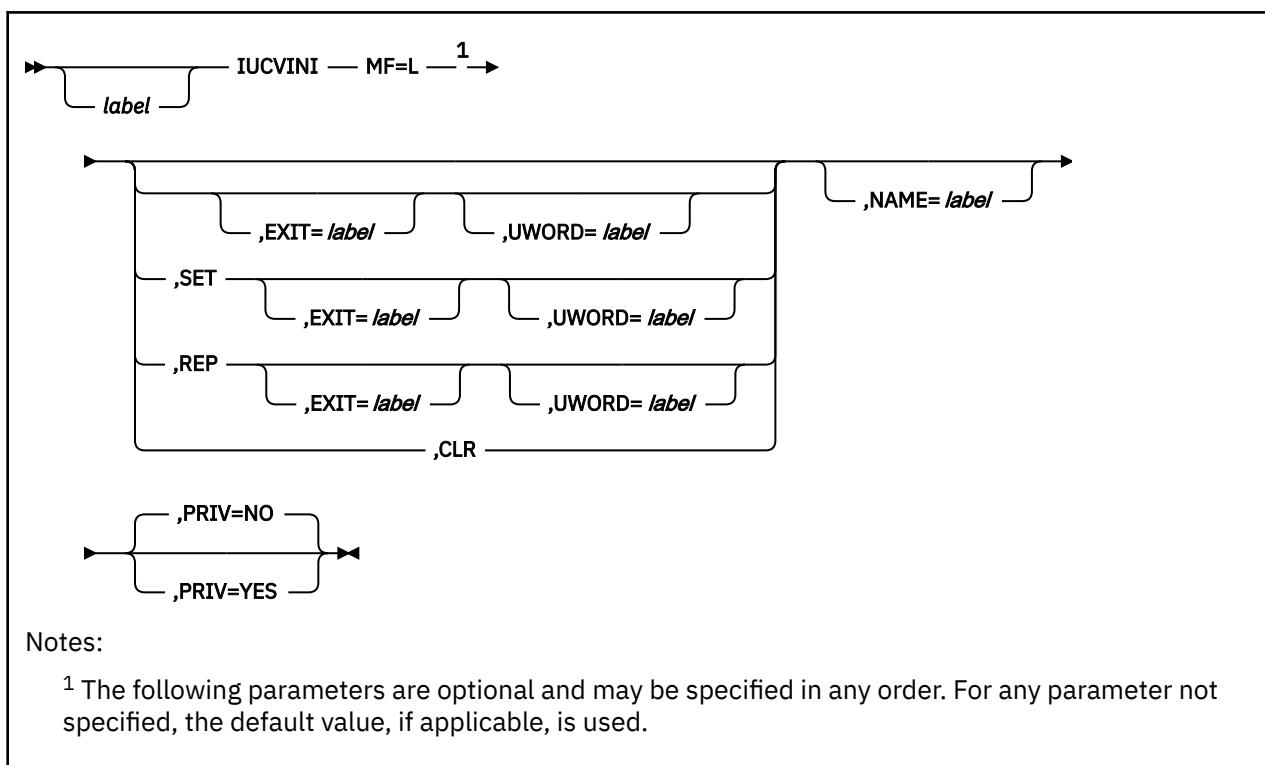
When this macro completes processing, it passes to the caller a return code in register 15.

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'00'	0	Function completed successfully.
X'04'	4	This name is already being used by another IUCV or APPC/VM user.
X'08'	8	The IUCV or APPC/VM facility cannot be used unless the IUCVINI macro, with the SET parameter specified, is issued first.
X'10'	16	Either the NAME parameter was not specified or its address was specified as zero.
X'14'	20	Either the EXIT parameter was not specified or its address was specified as zero.
X'28'	40	The function requested was unrecognizable by GCS. Specify SET, REP, or CLR.
X'2C'	44	Invalid parameter list.
X'34'	52	Either the user did not enter the IUCVINI REP instruction from the same task that it issued the IUCVINI SET instruction, or the IUCVINI REP instruction was issued by a privileged user.

Hex Code	Decimal Code	Meaning
X'7C'	204	An error occurred in obtaining storage to satisfy the IUCV or APPC/VM request. The return code from the GETMAIN macro is 204.
X'xxx'	1xxx	An error occurred while trying to sever all the user's communication paths. 'xxx' is the value in the IPRCODE field in the SEVER parameter list. The section of <i>z/VM: CP Programming Services</i> that defines the fields in the IUCV and APPC/VM parameter lists.

ABEND Code	Reason Code	Meaning
FCA	1101	A GETMAIN macro failed when GCS tried to obtain storage for the task that ended abnormally.

## List Format



## Purpose (List Format)

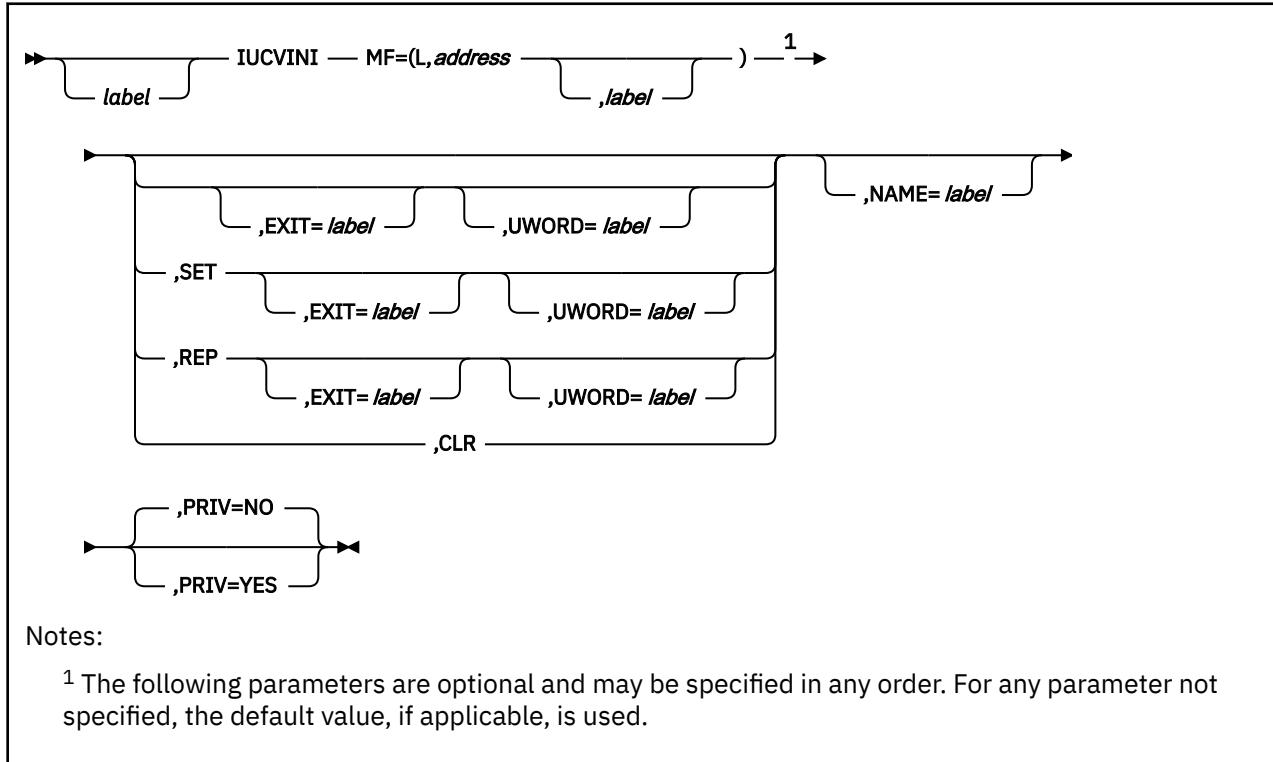
This format of the macro generates an in-line parameter list based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation. Only the preceding parameters listed are valid in the list format of this macro.

## Added Parameter

### MF=L

Specifies the list format of this macro.

## List Address Format



### Purpose (List Address Format)

This format of the macro does not produce any executable code that starts the function. However, it does produce executable code that moves the parameter values that you specify into a certain parameter list. If you enter the macro using this format, then you must do so before any related invocation of the macro using the execute format.

Only the preceding parameters listed are valid in the list address format of this macro.

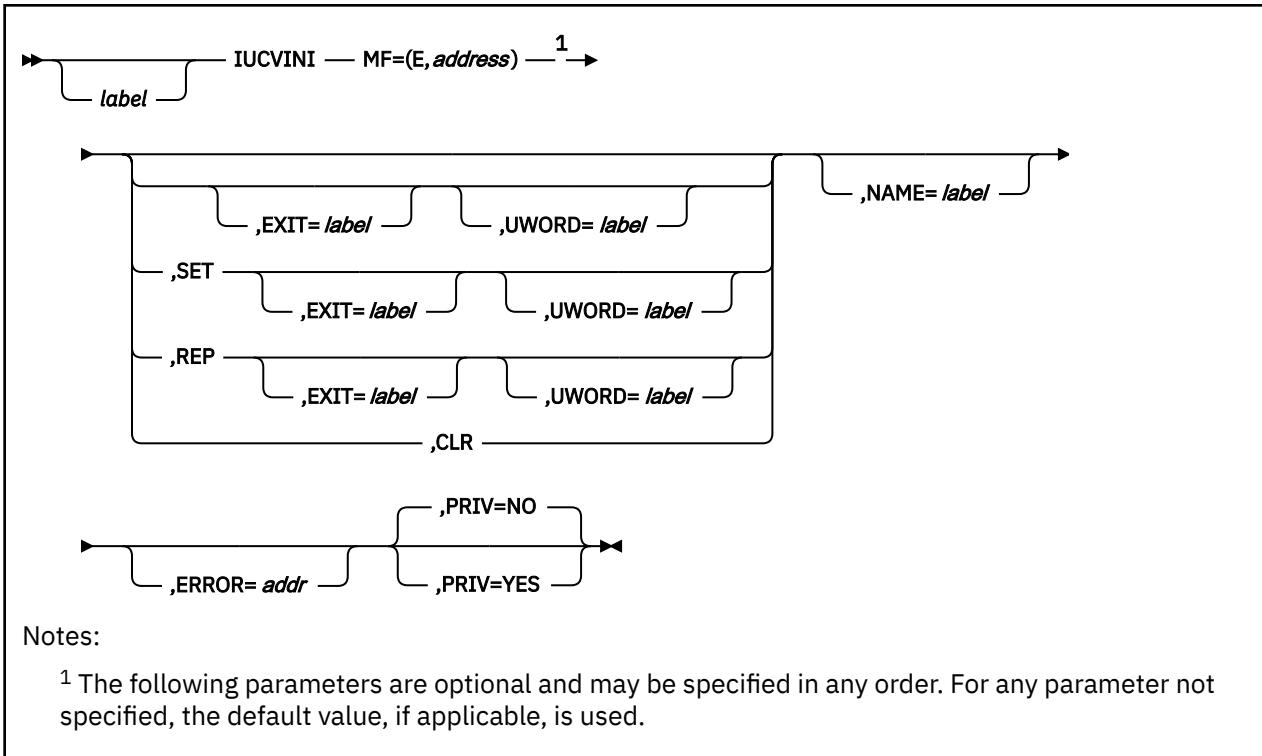
### Added Parameter (List Address Format)

#### **MF=(L,*address*,*label*)**

*address* specifies the address of the parameter list into which you want the parameter values you mention placed. This address can be within the user's program or somewhere in free storage.

*label* is optional and is a user specified label, indicating that you want to determine the length of the parameter list. The macro expansion equates the label you specify with the length of the parameter list.

### Execute Format



## Purpose (Execute Format)

This format of the macro generates code that executes the function using a parameter list whose address you specify.

Note that only the preceding parameters listed are valid in the execute format of this macro.

## Added Parameter (Execute Format)

### **MF=(E, *address*)**

*address* specifies the address of the parameter list to be used by the macro.

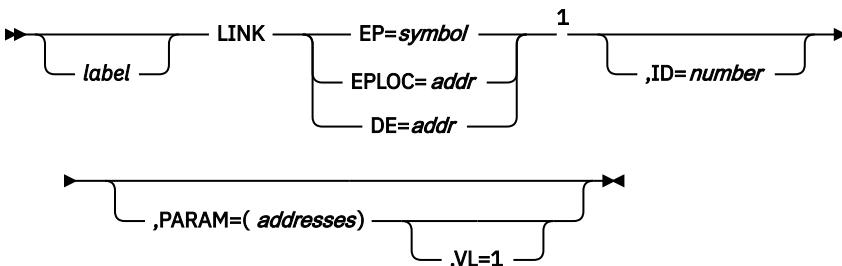
You can add or modify values in this parameter list by specifying them in this macro.

## LINK

The LINK macro is available in standard, list and execute formats.

### Standard Format

See also “List Format” on page 296 and “Execute Format” on page 296.



Notes:

<sup>1</sup> The following parameters are optional and may be specified in any order. For any parameter not specified, the default value, if applicable, is used.

### Purpose

Use the LINK macro to pass control to a certain entry point in another load module with the intent that control will eventually return to the program issuing the instruction.

GCS provides several techniques for passing control from one program to another.

### Parameters

#### EP

Specifies the name of the entry point within the program that is to receive control.

The entry point name can be any one of the following:

- The name of the entry point as previously defined through the IDENTIFY macro. See “[IDENTIFY](#) on [page 270](#).
- The name of the entry point declared in a shared segment directory through the CONTENTS macro. See “[CONTENTS](#)” on [page 197](#).
- A member name (or alias) in the directory of a load library.

When looking for the entry point name that you specify, GCS searches the following items in the following order:

1. Your private storage, because the module associated with the entry point name may already be loaded.
2. Any shared segment directories that may have been created through the CONTENTS macro.
3. The directories of any load libraries that may have been defined for your virtual machine through the GLOBAL LOADLIB command. For more information on the GLOBAL command, see “[GLOBAL](#)” on [page 101](#).

You must write this parameter as a symbol.

#### EPLOC

Specifies the address containing the name of the entry point of the program that is to receive control.

The name, as stored, can be up to 8 bytes long. If it is fewer than 8 bytes long, the name must be padded on the right with blanks.

You can write this parameter as an assembler program label or as register (2) through (12).

**DE**

Specifies the address of the name field within the list entry for the entry point in question.

You must previously have created this list entry for the entry point using the BLDL macro. See "[BLDL](#)" on page 181.

You can write this parameter as an assembler program label or as register (2) through (12).

**ID**

Specifies a number that GCS is to put in bytes 3 and 4 of the last instruction in the LINK macro expansion.

The last instruction in the LINK macro is a NOP instruction. GCS will place the number that you specify in this parameter into this NOP instruction. You can then use it as a debugging tool. Choose a number from 0 to 4095 or a symbol.

You can write this parameter as decimal digits or as an assembler program label.

**PARAM**

Specifies one or more parameter addresses that GCS will pass to the called program.

GCS builds a parameter list containing these addresses in the order which you specify them. Then, the system passes the address of this parameter list to the program called in register 1. If you omit this parameter, then register 1 remains unchanged.

You can write these parameters as assembler program labels or as registers (2) through (12).

**VL=1**

Indicates that the program called expects a variable number of parameters to be passed to it.

You must write this parameter exactly as shown, and you can use it only with the PARAM parameter. To omit the VL=1 parameter is to say that the program called expects a set number of parameters.

## Usage

1. If you enter the LINK macro and the load module in question is not resident in virtual storage, then GCS will load the module for you. Then, after the module is run, GCS removes it from storage. This is satisfactory if you intend to pass control to the module only once.

However, loading a module into virtual storage involves a good deal of overhead processing. If you intend to pass control to the module more than once, it is far more efficient to enter the LOAD macro yourself just one time. This avoids all the overhead processing involved in having GCS repeatedly load the module for you.

2. The relationship between the program issuing the LINK macro and the program receiving control is the same as that established by a BAL assembler language instruction. After the program being called has completed execution, control is returned to the program that issued the LINK instruction.
3. The LINK macro handles the setting of the addressing mode (24-bit or 31-bit addressing) when passing control. The called program is given control in the addressing mode indicated in its loadlib entry or by the CONTENTS macro. On entry to the called program, the high order bit, bit 0 of register 14, is set to indicate the addressing mode of the issuer of the LINK macro. If bit 0 is 0, the issuer is executing in 24-bit addressing mode; if bit 0 is 1, the issuer is executing in 31-bit addressing mode. This makes it possible to return control to the issuer in the addressing mode in which it was executing.
4. It is the responsibility of the program issuing the LINK instruction to provide the program receiving control with the address of an area where the former's registers will be saved. This address must be placed in register 13 by the program issuing the LINK instruction.
5. It is the responsibility of the program called to place the value of the other program's registers in this save area after it gets control. And, just before the called program returns control, the values must be

restored to registers 0 through 14. A return code can be placed in register 15; if not, then register 15 must be restored.

6. You can use the LINK macro to link to a serially reusable program. If the program is being used by someone else, then you will be placed in the WAIT state until the other user is finished.
7. If the program being called is reentrant, then it is loaded into key 0 storage. This ensures that it is not accidentally modified or tampered with.

### Examples

```
LINK EP=PROGRAMB,PARAM=(ADDRA,ADDRB,ADDRC)
```

Pass control to an entry point named PROGRAMB. PROGRAMB expects exactly three parameters be passed to it. These parameters may be found at addresses ADDRA, ADDR B, and ADDR C, respectively.

```
LINKIT LINK EPLOC=PROGADDR,PARAM=((2),(3)),VL=1
```

Pass control to an entry point whose name can be found at the address corresponding to the label PROGADDR. This program expects a variable number of parameters be passed to it, in this case two. The address of the first parameter can be found in register 2, and that of the second in register 3. LINKIT is the label on this instruction.

```
LINK DE=BLDLNAM,ID=6
```

Pass control to a certain entry point. The system looks for the name of the entry point in the BLDL list entry for that entry point. The name field of the list entry corresponds to the address of the label BLDLNAM. As an aid to debugging, the LINK macro places the value six in bytes 3 and 4 of the final instruction that it generates.

### Input to the Program Receiving Control

Register 0	Unpredictable. May be used by the GCS supervisor.
Register 1-13	Unchanged. Register 1 will contain the address of the parameter list, if it was specified.
Register 14	The address to which control is to return after the called program completes execution.
Register 15	The address of the entry point in the program called.

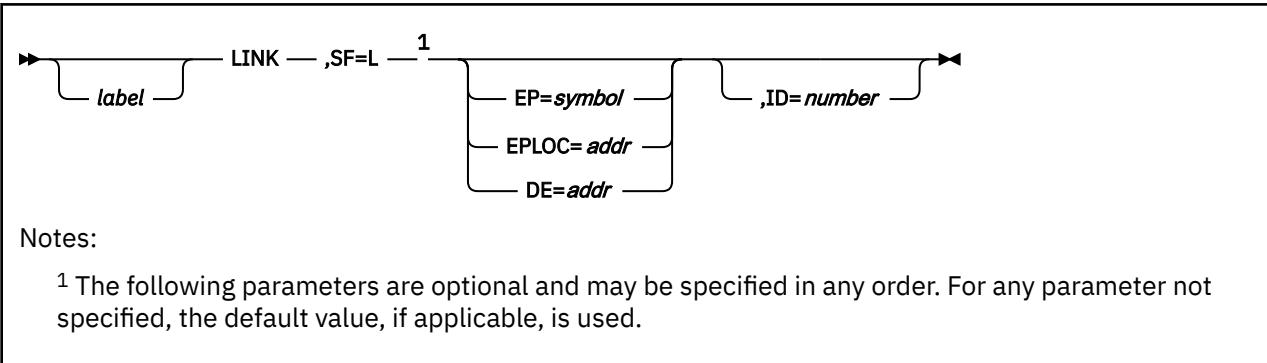
### Messages

The LINK macro generates no return codes.

ABEND Code	Reason Code	Meaning
106	0B	An error was found when the supervisor attempted to load the requested module into virtual storage.
106	0C	Insufficient virtual storage was available to load the requested module.
206		Invalid parameter list.
406		The module is marked ONLY LOADABLE.
706		The linkage editor marked the module NOT EXECUTABLE.
806	04	Either the program could not be found or no load libraries were defined by the GLOBAL command.
806	08	An irrecoverable I/O error occurred when the BLDL control program attempted to search the directory.

ABEND Code	Reason Code	Meaning
806	10	When GCS attempted to close the load library used by the BLDL macro, it found that the load library had never been opened.
906		The maximum use count or the maximum load count of the module has been reached.
A06		Your task is already waiting for this serially reusable module.

## List Format



## Purpose (List Format)

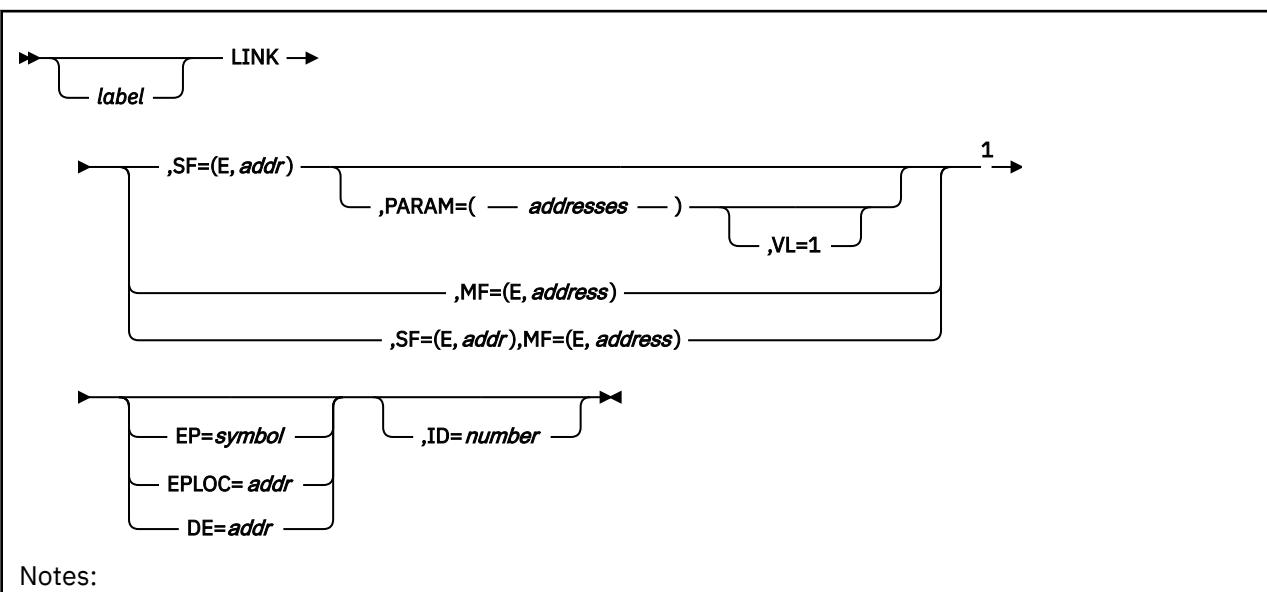
This format of the macro generates an in-line parameter list, based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation. Only the preceding parameters listed are valid in the list format of this macro.

## Added Parameter

### SF=L

Specifies the list format of this macro.

## Execute Format



<sup>1</sup> The following parameters are optional and may be specified in any order. For any parameter not specified, the default value, if applicable, is used.

## Purpose (Execute Format)

This format of the macro generates code that executes the function, using a parameter list whose address you specify. Only the preceding parameters listed are valid in the execute format of this macro.

## Added Parameter (Execute Format)

### **SF=(E, *address*)**

*address* specifies the address of the parameter list to be used by the macro. This is the parameter list that was generated through the list format of this macro.

You can add or modify values in this parameter list by specifying them in this macro.

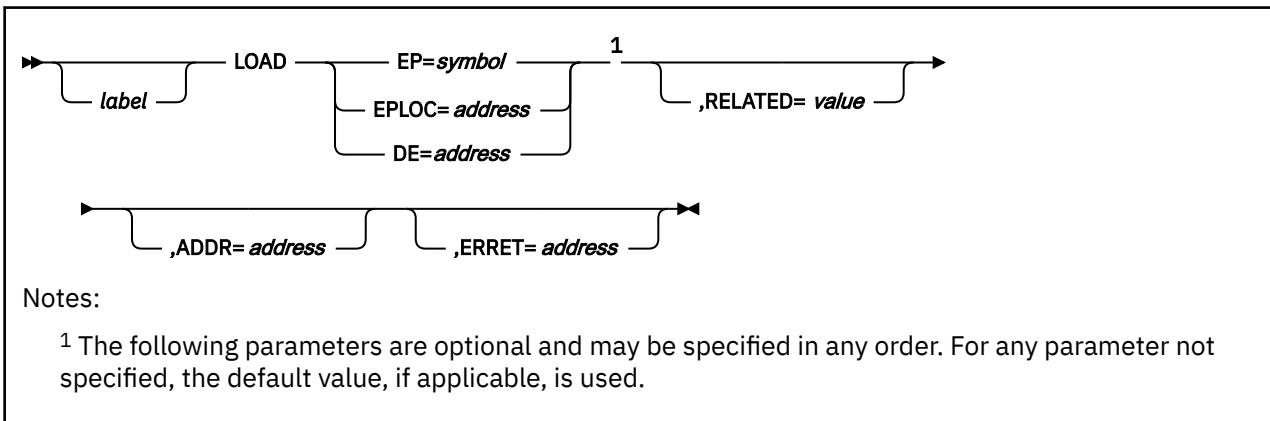
### **MF=(E, *address*)**

*address* specifies the address of the remote parameter list to be used by the called program.

You can add or modify values in this parameter list by specifying them in this macro.

# LOAD

## Format



## Purpose

Use the LOAD macro to bring a load module, containing a specified entry point, into virtual storage. The load module will be placed above or below the 16MB line depending on the RMODE of the module; RMODE is specified in the directory entry for the module.

This makes the code at that entry point available for your use.

## Parameters

### EP

Specifies the name of the entry point contained in the load module to be brought into storage.

The entry point name can be any one of the following:

- The name of the entry point as previously defined through the IDENTIFY macro. See “[IDENTIFY](#) on page 270.”
- The name of the entry point declared in a shared segment CONTENTS macro. See “[CONTENTS](#)” on page 197.”
- A member name (or alias) in the directory of a load library.

When looking for the entry point name that you specify, GCS searches the following items in the following order:

1. Your private storage, because the module associated with the entry point name may already be loaded.
2. Any shared segment directories that may have been created through the CONTENTS macro.
3. The directories of any load libraries that may have been defined for your virtual machine through the GLOBAL LOADLIB command. For more information on the GLOBAL command, see “[GLOBAL](#)” on page 101.”

You must write this parameter as a symbol.

### EPLOC

Specifies the address in your program where you have stored the name of the entry point.

This name may be up to 8 bytes long. If it is fewer than 8 bytes long, it must be padded on the right with blanks. Again, the entry point name can refer to one of the three things listed under the EP parameter.

You can write this parameter as an RX-type address, as register (0), or as register (2) through (12).

#### **DE**

Specifies the address of the NAME field within the directory list entry associated with the entry point in question.

**Note:** RMODE of the load module must agree with the address of DE. That is, if the user specifies an address above 16MB, the load module must have an RMODE of ANY.

GCS assumes that you have created this list entry within the directory using the BLDL macro. See “[BLDL](#)” on page 181. When using the BLDL macro for this particular purpose, specify at least 62 bytes as the length of the list entry for your entry point.

You can write this parameter as an RX-type address, as register (0), or as register (2) through (12).

#### **RELATED**

Specifies documentation data that you are using to relate this macro to a DELETE macro.

The value you assign to this parameter has nothing to do with the execution of the macro itself. It merely relates one macro (LOAD) to a macro that provides an opposite, though related, service (DELETE).

The format and contents of this parameter are at your discretion and can be any valid coding value.

#### **ADDR**

Specifies the address where the module is loaded, this must also be the entry point address. You are allowed to specify an address in private or common storage.

Only applications in supervisor state may specify an address in common storage. To prevent the alteration of the loaded area of common storage while it is being referenced, the common lock may be obtained. Storage for the module must be previously allocated in the requester's key on a doubleword boundary and within the virtual machine size or within common storage boundaries. If the application is running in an authorized machine in supervisor state, the storage is not checked to see if it is in the correct key. The key of the storage is checked for all other calls.

The DCB parameter that is used in MVS with the ADDR parameter is not used. The usual search order for disks finds the module.

In searching for the module to be loaded, this function does not search either private or common storage. The search for the module consists of searching the globaled loadlibs using the normal search order of accessed disks.

Modules loaded with the ADDR parameter cannot be deleted and are not listed by QUERY LOADALL.

You can write this parameter as an RX-type address, as register (0), or as register (2) through (12).

#### **ERRET**

Specifies the address of a routine to receive control if an error in the load process causes an ABEND.

The error routine receives the ABEND code that describes the problem in register 1. In register 15, it receives a reason code that explains why the ABEND occurred.

**Note:** If an invalid macro parameter is given, the error routine does not receive control.

### **Usage**

1. If you specify the DE parameter, then GCS assumes that a list entry has been created for the entry point in the directory entry list using the BLDL macro.
2. The LOAD macro does not pass control to the entry point in question. Rather, the address of the entry point is returned to your program in register 0. The LOAD macro sets the high order bit of the entry point address in register 0 to indicate the module's AMODE, which is obtained from the directory entry

for the module. If the module's AMODE is 31-bit, it sets the indicator to 1. If the module's AMODE is 24-bit, it sets the indicator to 0. If the module's AMODE is ANY, it sets the indicator to correspond to the caller's AMODE.

3. The entire load module containing the specified entry point is brought into virtual storage. This happens, however, only if there is no other usable copy of the module available. It remains in your private storage until no outstanding requests for the module remain.
4. For each LOAD macro that you enter, except if ADDR is specified, you must also enter a corresponding DELETE macro. See “[DELETE](#)” on page 202.
5. If the program called is reentrant, then it is loaded into key 0 storage. This ensures that it is not accidentally modified or tampered with.

### Examples

```
LOADIT LOAD EP=XYZ,RELATED=DLEETIT
:
:
DLEETIT DELETE EP=XYZ,RELATED=LOADIT
```

Bring the load module containing the entry point XYZ into virtual storage. This LOAD macro is cross-referenced with a related DELETE macro by use of the RELATED parameters in each.

### Return Codes and ABEND Codes

The program issuing the LOAD macro receives the following information in its registers.

Register	Contents
0	The address of the entry point specified in the LOAD macro.
1	If the load module is in private storage, then this is the length of the load module in doublewords. If the load module is in a shared segment, then this length is set to zero.
15	A return code of zero indicating a successful load.

The LOAD macro generates the following ABEND codes. If applicable, a reason code is returned in register 15.

ABEND codes 206, 706, and 906 are associated with a reason code of 04 only when the ERRET parameter is specified.

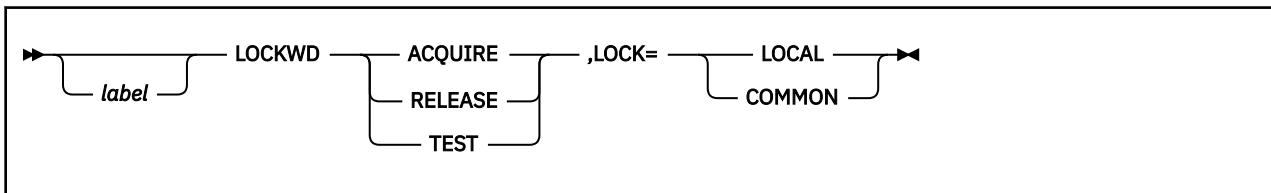
ABEND Code	Reason Code	Meaning
106	0B	An error was found when the supervisor attempted to load the requested module.
106	0C	Insufficient virtual storage was available to load the requested module.
206	04	Invalid parameter list.
706	04	The linkage editor marked the requested load module as NOT EXECUTABLE.
806	04	Either the program could not be found or no load libraries were defined by the GLOBAL LOADLIB command.
806	08	An irrecoverable I/O error occurred when the BLDL control program routine attempted to search the directory.
806	10	When GCS attempted to close the load library used by the BLDL macro, it found that the load library had never been opened.

<b>ABEND Code</b>	<b>Reason Code</b>	<b>Meaning</b>
906	04	The LOAD COUNT or USE COUNT for the load module have reached the maximum of 32767.

## LOCKWD

---

### Format



### Purpose

Use the LOCKWD macro to acquire or release a lock on common or private storage.

GCS allows several virtual machines in a virtual machine group to share common storage. This creates competition among the machines for access to the shared storage. Multitasking within a single virtual machine creates competition among several tasks for access to local resources. The word *resources* includes the virtual machine's private storage, I/O devices, tapes, disks, and so forth.

The LOCKWD macro helps you to manage this competition. It allows a virtual machine to acquire exclusive use of common storage while it accesses, and possibly modifies, the data therein. It allows one of several tasks within a virtual machine to acquire exclusive use of a private resource. Once the virtual machine or task is finished, it must then reissue the LOCKWD macro to release its lock so others can use the resource.

The LOCKWD macro is an authorized GCS function.

### Parameters

#### ACQUIRE

Indicates that the virtual machine or task wants to establish the lock specified in the macro.

#### RELEASE

Indicates that the virtual machine or task wants to give up the lock it acquired previously. That lock is specified in the macro.

#### TEST

Indicates that the virtual machine or task wants to know if it holds a lock on common storage.

This option is valid only with the LOCK=COMMON parameter.

#### LOCK

Indicates that the description of the lock to be acquired or released follows.

#### LOCAL

Indicates that a task within a single virtual machine either wants to acquire or release a lock on the machine's local resources.

#### COMMON

Indicates that a virtual machine within a virtual machine group wants to acquire or release a lock on the common storage shared by the entire group.

### Usage

1. Before you acquire a lock on common storage, you must first acquire a lock on your own local resources. This ensures that your task cannot be interrupted by any other task also seeking a lock on common storage.
2. The supervisor acquires and releases locks for the virtual machine or task.

3. If a certain virtual machine holds a lock on common storage, then no other virtual machine in the group may acquire that lock until it is released. A virtual machine that requests a lock on common storage already held by another machine is placed in the WAIT state.
4. If a task within a virtual machine has obtained a lock on the machine's private storage, then that task is disabled from interrupts. This means that no other task within the virtual machine can interrupt until the task holding the lock releases it. In effect, no other task in the machine may run or obtain access to private storage until this time.
5. There are two ways to release a lock:
  - A virtual machine or task explicitly reissues the LOCKWD macro with the RELEASE parameter and lock properly specified.
  - A virtual machine or task that is holding a lock ends.
6. The LOCKWD macro can help manage the natural competition for storage access among virtual machines and tasks.
7. Often an authorized program will be called to perform work for an unauthorized program. Usually the authorized program runs in a different key from the unauthorized program. In such cases, the LOCKWD macro is required before the authorized program issues the VALIDATE macro. See "["VALIDATE" on page 362](#)".

**NOT-PI**

8. Some virtual machines and tasks run in supervisor state. Those that do are able to inspect and modify the fullword in storage that contains the lock. Under no circumstances should this fullword be modified! This privilege is strictly reserved to the GCS supervisor.

**NOT-PI end**

9. If you have requested a lock on common storage, you must be careful to release that lock when you are through with your task. Failure to release any lock can cause unnecessary and prolonged delays for other virtual machines in the group that are waiting for access to common storage.

**Examples**

```
LOCKWD ACQUIRE,LOCK=COMMON
```

The task requests a lock on common storage. Presumably, the task has already acquired a lock on its own local resources.

```
LOCKWD TEST,LOCK=COMMON
```

The task wants to know if it holds the lock on common storage.

**Return Codes and ABEND Codes**

When this macro completes processing, it passes to the caller a return code in register 15.

**General Return Codes:**

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'00'	0	The lock was successfully acquired or released.
X'04'	4	For an ACQUIRE request, this return code means that the virtual machine or task making the request already holds the lock specified. For a RELEASE request, the virtual machine or task making the request does not hold the lock specified.

**For the TEST function:**

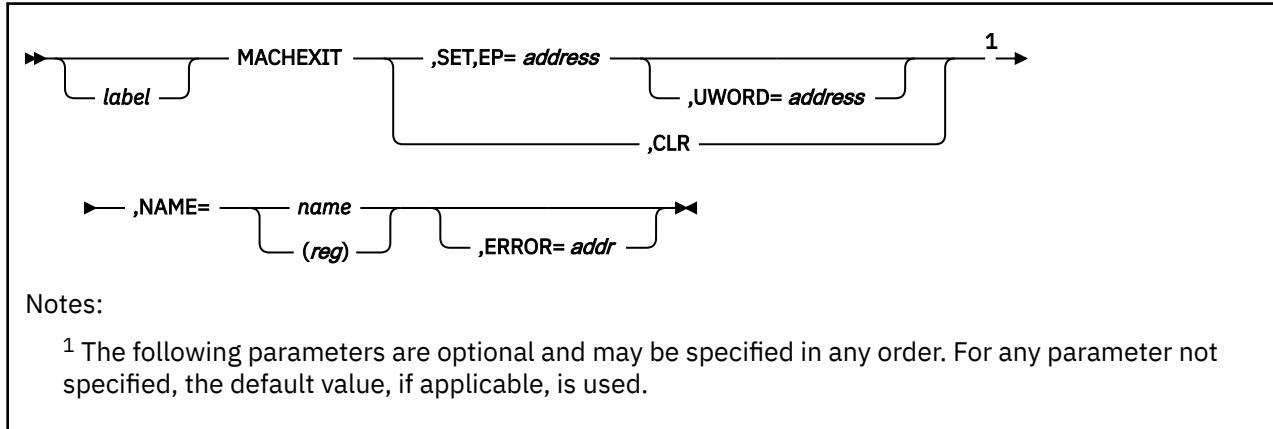
Hex Code	Decimal Code	Meaning
X'00'	0	The lock is free.
X'04'	4	Your machine and task hold the lock on common storage.
X'08'	8	Another machine and task hold the lock on common storage.
ABEND Code	Meaning	
0600		Your task does not hold a lock on its local resources. Your task must acquire a lock on its local resources before it tries to acquire a lock on common storage.

## MACHEXIT

The MACHEXIT macro is available in standard, list, list address and execute formats.

### Standard Format

See also [“List Format” on page 308](#), [“List Address Format” on page 308](#) and [“Execute Format” on page 309](#).



### Purpose

Use the MACHEXIT macro to declare or cancel a machine termination exit routine for a virtual machine group.

Often it is useful to declare a machine termination exit routine for your entire virtual machine group. This routine will receive control when one of the virtual machines in the group is reset.

**Note:** A virtual machine is reset under one of the following conditions: LOGOFF, IPL, when certain machine checks occur, and when certain authorized commands are issued, namely SYSTEM RESET, SYSTEM CLEAR, DEFINE STORAGE, and SET MACHINE. A virtual machine is also reset when its GCS supervisor ends abnormally or when it issues the IUCV SEVER or IUCV RETRIEVE BUFFER instruction. It may also be forced to reset by the CP operator.

To illustrate, let us say that a virtual machine group is processing a certain file. The authorized machine that is managing the effort needs to know if another member of the group resets so it can make certain adjustments in the processing. A machine termination exit routine may be provided to analyze the situation that caused a machine to reset. The exit routine may then make the necessary adjustments or it may communicate with the managing authorized machine so that the latter can make the adjustments.

A machine termination exit routine can help your virtual machine group manage its common storage. A machine termination exit routine can also perform CP SENDS to a machine, if it is running disconnected and if the user performing the SENDS is defined as the secondary user of the target machine.

The AMODE of the machine exit will be taken from the correspondent entry in the CONTENTS macro. See [“CONTENTS” on page 197](#).

Use the MACHEXIT macro to declare or cancel a machine termination exit routine for an entire virtual machine group.

The MACHEXIT macro is an authorized GCS function.

## Parameters

### SET

Indicates that you are declaring a machine termination exit routine for your virtual machine group.

### CLR

Indicates that you are canceling a machine termination exit routine that was previously declared for your virtual machine group.

Any authorized virtual machine in the group can cancel such a routine. It is not necessary that the routine be canceled by the same machine that declared it.

### EP

Specifies the address of the machine termination exit routine that you are declaring.

The routine in question must be resident in a shared segment. That is, a routine whose entry point is defined in a shared segment directory that was created through the CONTENTS macro. See “[CONTENTS](#)” on page 197.

You can write this parameter as an assembler program label or as register (2) through (12).

### UWORD

Specifies a fullword of data that you want passed to the machine termination exit routine, if it ever gains control.

You can use this parameter to pass any information you please.

If you write this parameter as an assembler program label, then the address associated with that label is passed to the exit routine. If you write it as register (2) through (12), then the contents of the register are passed to the routine.

### NAME

Specifies a one to eight-character name that identifies the machine termination exit routine to the MACHEXIT macro.

This name must not be confused with the routine's module name, program name, or entry point name. The name referred to by this parameter is simply a character string used to identify the routine to the MACHEXIT macro. Outside the MACHEXIT macro environment, this name is meaningless.

Not every authorized machine in the group knows the routine's address. This option provides a way for any authorized machine to refer to the exit.

Note that the name for the routine is declared by the authorized machine that declares the exit routine. That machine must supply both the name and the address of the routine declared, thereby associating the name with the address.

You can write this parameter as the name itself or as register (2) through (12). If you store it as a name less than eight characters long, and specify it using a register, then it must be padded on the right with blanks. A name consisting of more than eight characters would be truncated. GCS does not allow a name consisting of all blanks. If you write it as a register, then the register must contain the address of the name.

### ERROR

Specifies the address of an error routine that will receive control if an error occurs in the MACHEXIT macro.

If you omit this parameter and an error occurs, then control will return to the instruction following the MACHEXIT macro, just as it would were there no error.

You can write this parameter as an assembler program label or as register (2) through (12).

## Usage

1. Only an authorized virtual machine can issue the MACHEXIT macro.

2. A machine termination exit routine always runs in the recovery machine designated for the virtual machine group. Moreover, it runs in the same key as the virtual machine that declared it, and it always runs in supervisor state.
3. An authorized member of a virtual machine group can declare more than one machine termination exit routine for the group. Each will run in the event one of the machines in the group resets. However, the routines will not necessarily run in the order which they were declared.
4. A machine termination exit routine will be executed normally in the AMODE specified in the correspondent CONTENTS entry. However, if the AMODE parameter in the CONTENTS macro is DEFINED, then the address of the routine in the MACHEXIT macro will be considered a 32 bit address with the AMODE being the first bit.
5. A machine termination exit routine is always associated with the task that declared it. When a task terminates, any machine termination exit routine it may have declared is canceled.
6. In a typical scenario, a machine termination exit routine may be scheduled for execution when one virtual machine resets and later be canceled by another virtual machine. However, the routine would still run because it has already been scheduled. You should take this into account when designing your over-all processing procedure.
7. No machine termination exit routine can receive control through the AUTHCALL macro. Such a routine receives control only if it is properly declared through the MACHEXIT macro and if some virtual machine within the group resets.
8. When the machine termination exit routine receives control, its registers contain the following.

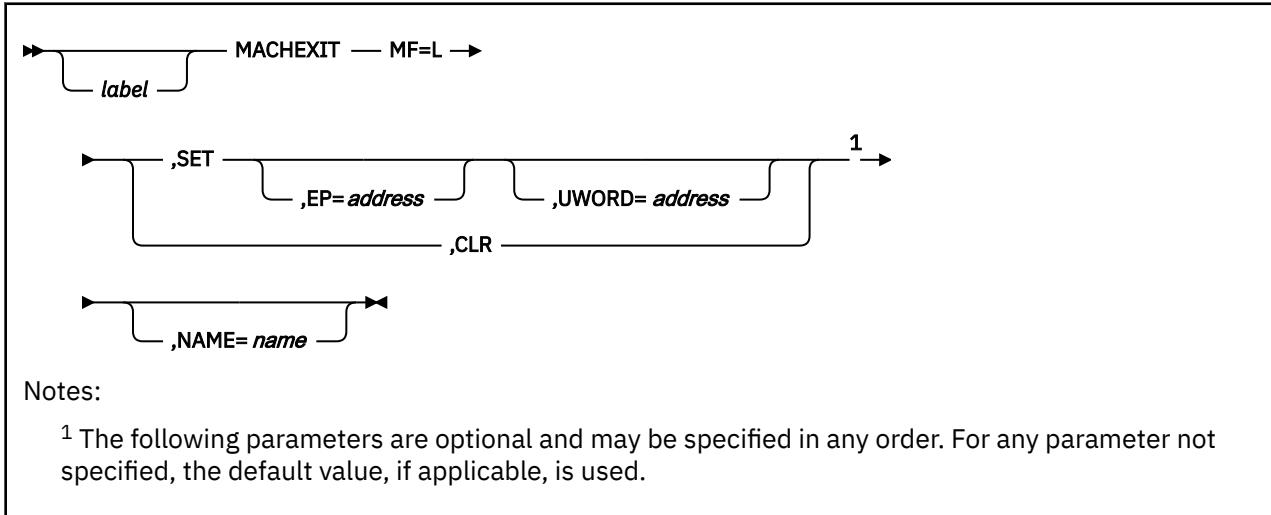
<b>Register</b>	<b>Contents</b>
0	Bits 0 - 15: The machine ID of the virtual machine that was reset. Bits 16 - 31: Reserved.
1	The UWORD parameter specified in the MACHEXIT macro that declared the routine.
13	The address of a 72-byte save area.
14	The return address.
15	The address of the entry point in the exit routine.

## Return Codes and ABEND Codes

When this macro completes processing, it passes to the caller a return code in register 15.

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'00'	0	Function completed successfully.
X'04'	4	The specified machine termination exit routine has already been declared.
X'08'	8	The specified machine termination exit routine is not in common storage.
X'18'	24	Invalid parameter list.
X'2C'	44	The name of the machine termination exit routine that you want to cancel could not be found.
X'30'	48	The CONTENTS entry has AMODE=DEFINED or AMODE=CALLER, the caller is in AMODE 24 and the exit routine address is above the 16MB line.

## List Format



## Purpose (List Format)

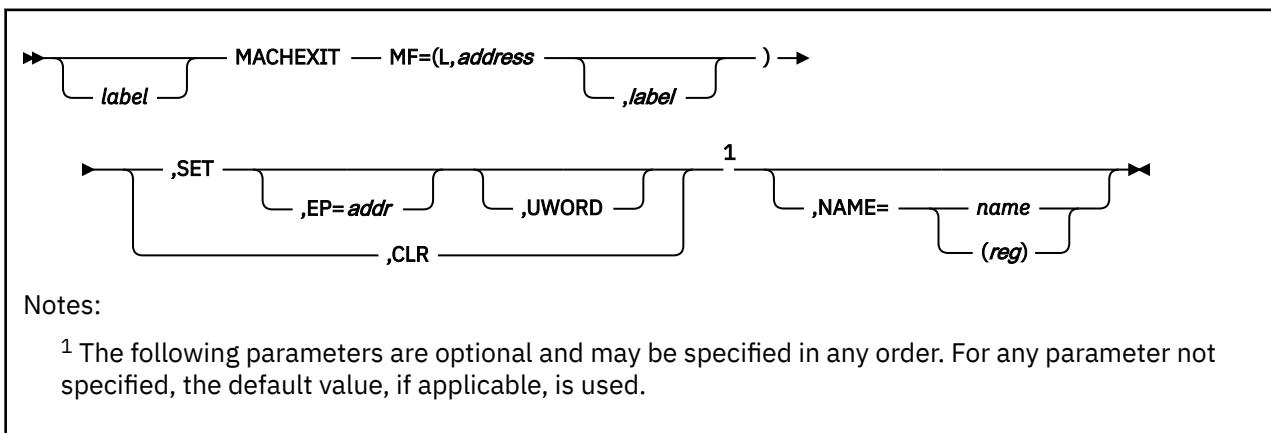
This format of the macro generates an in-line parameter list based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation. Only the preceding parameters listed are valid in the list format of this macro.

## Added Parameter

### MF=L

Specifies the list format of this macro.

## List Address Format



## Purpose (List Address Format)

This format of the macro does not produce any executable code that invokes the function. However, it does produce executable code that moves the parameter values that you specify into a certain parameter list. If you enter the macro using this format, then you must do so before any related invocation of the macro using the execute format. Only the preceding parameters listed are valid in the list address format of this macro.

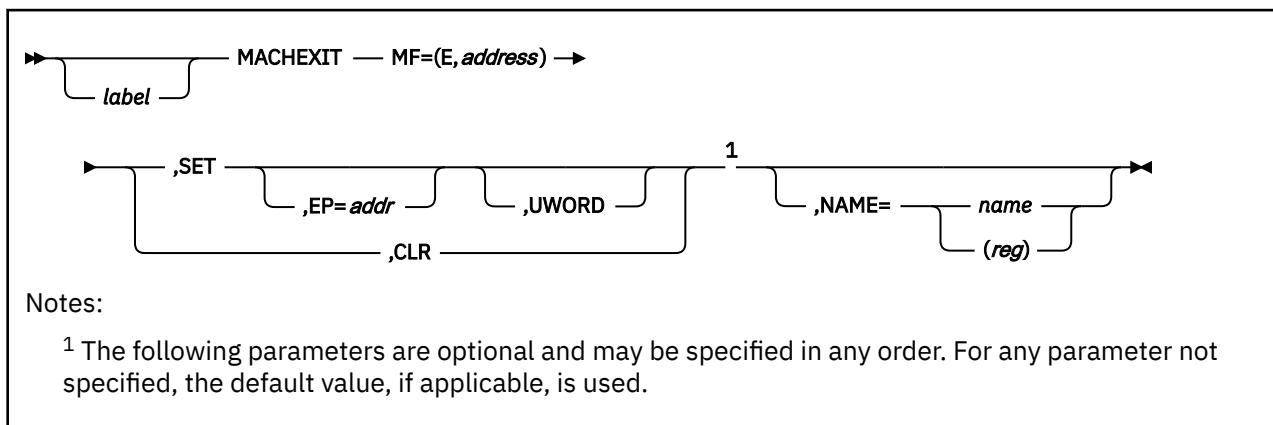
## Added Parameter (List Address Format)

**MF=(L, address,label)**

*address* specifies the address of the parameter list into which you want the parameter values you mention placed. This address can be within your program or somewhere in free storage.

*label* is optional and is a user-specified label, indicating that you want to determine the length of the parameter list. The macro expansion equates the label you specify with the length of the parameter list.

## Execute Format



## Purpose (Execute Format)

This format of the macro generates code that executes the function using a parameter list whose address you specify.

## Added Parameter (Execute Format)

**MF=(E, address)**

*address* specifies the address of the parameter list to be used by the macro.

You can add or modify values in this parameter list by specifying them in this macro.

# PGLOCK

---

## Format



## Purpose

Use the PGLOCK macro to lock a certain page of virtual storage into real storage.

If your program performs real I/O operations, then the pages of storage used for these operations must be locked into real storage.

The PGLOCK macro locks a specified page of your virtual storage into real storage. This makes the page ineligible for page-out.

The PGLOCK macro is an authorized GCS function.

## Parameters

### **reg**

Specifies the register that contains the address of the virtual page to be locked into real storage.

You can write this parameter as register (0) or as register (2) through (12).

## Usage

1. The task that issues the PGLOCK macro must be running in supervisor state. Also, the DIAG98 parameter must be specified in the OPTION control statement in the virtual machine's directory entry.
2. Use of the PGLOCK macro can enhance your program's efficiency by making the CP virtual-to-real translation step unnecessary. Also, it rids the system of the need to repeatedly lock and unlock pages of your storage every time you perform an input or output operation.
3. The AMODE of the program issuing the PGLOCK must be the same as the AMODE of the program issuing the PGULOCK. See “[PGULOCK](#)” on page 312.
4. In AMODE 24 special consideration needs to be given to the number of pages locked. In this mode pages are obtained from a special area situated below the 16MB line, thus the amount available is limited. (See Usage Notes for the Diagnose Code X'98' in [z/VM: CP Programming Services](#).)
5. The PGLOCK macro returns the real address of the locked page in register 1.
6. If the address you specify for the page is not on a page boundary, then the page that contains that address will be locked into real storage.
7. There are two ways for a page locked by the PGLOCK macro to be unlocked:
  - The task that issued the PGLOCK macro ends
  - A task explicitly issues the PGULOCK macro, correctly specifying the virtual address of the page to be unlocked.
8. A supervisor state program often must build a channel control program in real storage. When it does, it should use the PGLOCK macro to lock into real storage the page in where it is building the channel control program. See “[GENIO](#)” on page 247.
9. If you engage in real input/output activities, you must observe certain restrictions.

First, the storage size declared for your virtual machine must be large enough to accept the page you wish to lock.

Second, the storage size declared for your virtual machine group's recovery machine must be at least as large as that declared for your machine. This is to allow for the possibility that the recovery machine may be called upon to process exit routines you specified through the GENIO macro. See “[GENIO](#)” on [page 247](#).

## **Return Codes and ABEND Codes**

The PGLOCK macro generates no ABEND codes.

When this macro completes processing, it passes to the caller a return code in register 15.

---

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'00'	0	Function completed successfully.
X'04'	4	Reserved.
X'08'	8	The virtual address of the page in question is invalid.
X'0C'	12	GCS is unable to lock the specified page. No real page frames are available.
X'10'	16	The specified page is already locked.
X'14'	20	The virtual machine issuing this macro is not authorized to perform any real I/O operations.

---

# PGULOCK

---

## Format



## Purpose

Use the PGULOCK macro to unlock a certain page of virtual storage that was locked in real storage.

If you need to lock a certain page of virtual storage into real storage, you should take care to release it when it is no longer needed. Otherwise you tie up an important resource.

The PGULOCK macro unlocks a certain page of virtual storage that was previously locked in real storage using the PGLOCK macro. Unlocking such a page makes it eligible for page-out once again.

The AMODE of the program issuing the PGULOCK macro needs to be the same as the AMODE of the program where the corresponding PGLOCK was issued.

The PGULOCK macro is an authorized GCS function.

## Parameters

### **reg**

Specifies the register that contains the address of the virtual page to be unlocked from real storage.

You can write this parameter as register (1) through (12).

## Usage

1. The task that issues the PGULOCK macro must be running in supervisor state. Also, the DIAG98 parameter must be in the OPTION control statement in the virtual machine's directory entry.
2. If a PGULOCK macro is not issued for a page that is locked, then the page is automatically unlocked when the task that locked it ends.
3. A locked page does not necessarily have to be unlocked by the same task that locked it.

## Return Codes and ABEND Codes

The PGULOCK macro generates no ABEND codes.

When this macro completes processing, it passes to the caller a return code in register 15.

---

Hex Code	Decimal Code	Meaning
X'00'	0	Function completed successfully.
X'04'	4	Reserved.
X'08'	8	The virtual address of the page in question is invalid.
X'0C'	12	The specified page is not locked.

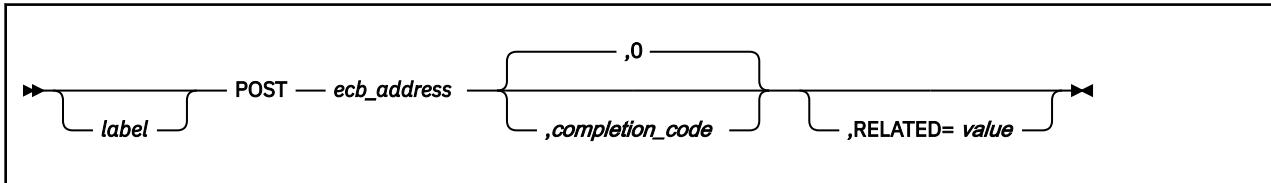
---

Hex Code	Decimal Code	Meaning
X'10'	16	The virtual machine issuing this macro is not authorized to perform any real I/O operations.

## POST

---

### Format



### Purpose

Use the POST macro to signal a task that the event it is waiting for has taken place.

A task that has issued the WAIT macro cannot continue until a certain event has taken place. See “[WAIT](#)” on page 365. It is the responsibility of the program effecting this event to inform the waiting task that the event has occurred.

Each such event is associated with an event control block (ECB). This ECB defines the event that is to occur and indicates to the waiting task whether it has occurred.

Use the POST macro to inform a task that the event it is waiting for has taken place.

### Parameters

#### ***ecb\_address***

Specifies the address of the event control block associated with the event that has occurred.

You can write this parameter as an RX-type address or as register (1) through (12).

#### ***completion\_code***

Specifies the code describing the manner which the event in question took place.

These codes have significance only to the programmers at your installation (and to the programs they write). Each installation must define the meaning of some or all of these completion codes and document them.

A completion code may be any number from 0 to  $2^{30}-1$ . If you omit this parameter, a completion code of 0 is assumed, by default. The completion code may be specified as a constant or as register (0) through (15).

#### **RELATED**

Specifies documentation data that you are using to relate this macro to a WAIT macro. The value you assign to this parameter has nothing to do with the execution of the macro itself. It merely relates one macro (POST) to another instruction that provides an opposite, though related, service (WAIT).

The format and content of this parameter are at your discretion, and can be any valid coding values.

### Usage

1. The task issuing the WAIT macro and the task issuing the POST instruction provide storage for each event control block. Each ECB is a fullword on a fullword boundary.
2. Bit 0 of the ECB is called the WAIT bit. If this bit is set to 1, then it means that some task is waiting for the event associated with that ECB to occur.
3. Bit 1 of the ECB is called the POST bit. The POST macro sets the POST bit of the appropriate ECB to 1 and then resets the WAIT bit to 0. These actions signal the waiting task that the event in question has taken place.

4. The remaining 30 bits of the ECB hold the completion code, after the ECB is posted.
5. Tasks are not always placed in the WAIT state after having issued the WAIT macro. Because the task is immediately satisfied, there is no reason for it to go into the WAIT state.
6. It is possible for a program to perform a branch entry into the POST macro code.

Those programmers who find it necessary to perform such a branch entry must be disabled for interrupts and be running in supervisor state and in key 0. They must do the following before taking this branch:

- a. Provide a save area in virtual storage that is 224 bytes long. In the first word of this save area you must store the number 152. In the third word of this save area you must store the sum of the address of the save area plus 72.
- b. You must be certain that the registers contain the following information:

Register	Contents
0	The COMPLETION CODE in the low-order 30 bits.
1	The address of the ECB in question.
13	The address of the 224-byte save area.
14	The return address within your program.
15	The address of the entry point in the POST macro to which you are branching.

- c. Because the point to which you will branch will be in low storage, use the FLS macro to generate the FLS DSECT. See “[FLS](#)” on page 231. Include the

```
USING FLS,0
```

instruction in your program, and branch to the address stored at the address associated with the label FLSPOST.

7. Be certain that none of your tasks change any of the bits in an ECB for which a WAIT instruction has been issued. Only after the POST bit has been set to 1 and its contents analyzed is it safe to alter an ECB.

## Examples

```
DONE POST (3),657
```

A certain event has taken place. The ECB associated with this event can be found at the address in register 3. The POST bit at this address is to be set to 1, and the WAIT bit reset to 0. A completion code of 657 is also placed in the ECB. DONE is the label on this instruction.

```
POST (8)
```

This means the same as in the last example, with two exceptions. The address of the ECB is in register 8, and the completion code is 0, by default.

## Return Codes and ABEND Codes

Note that these return codes are possible only when a branch entry to the POST macro is involved.

Hex Code	Decimal Code	Meaning
X'04'	4	The address of an ECB was invalid.
X'08'	8	The state block that is waiting for the ECB to be posted is not in the virtual machine's task block/state block structure.

## **POST**

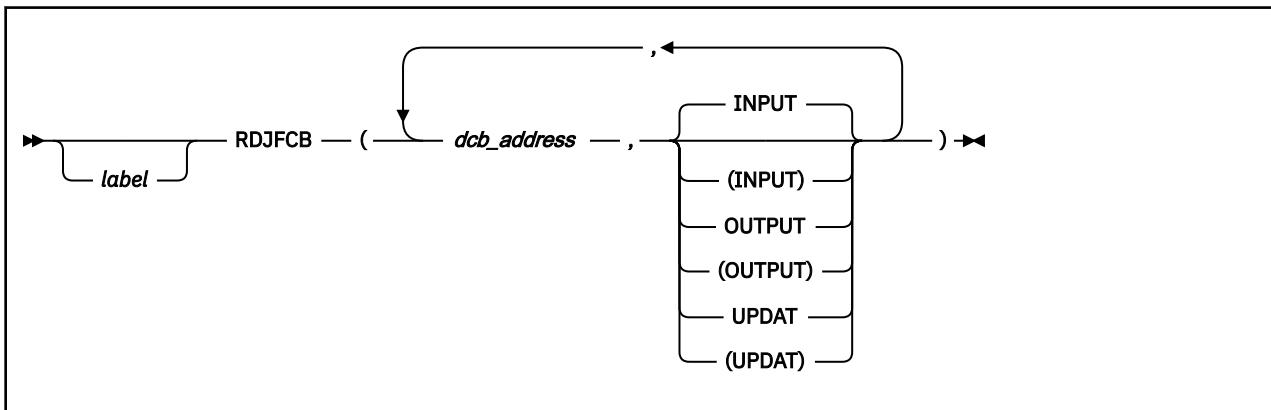
Note that these ABEND codes are possible only during a usual SVC call from the POST macro.

<b>ABEND Code</b>	<b>Meaning</b>
0F8	The GCS supervisor was called in access register mode.
102	The ECB in question is not addressable by the program issuing the POST macro.
202	The state block associated with the ECB to be posted is not in the task block/state block structure of the task waiting for the event.

# RDJFCB

---

## Format



## Purpose

The RDJFCB macro causes information about a file defined by the FILEDEF command to be moved into the user's area as identified through the EXLST parameter of the DCB macro for each data control block specified.

## Parameters

### **dcb\_address**

is the address of the data control block associated with the file for which the JFCB is to be read. The parameters INPUT, OUTPUT, and UPDAT do not affect RDJFCB processing.

### **INPUT**

Indicates that your file is to be treated as an input file. Unless otherwise specified, this parameter applies by default.

### **OUTPUT**

Indicates that your file is to be treated as an output file.

### **UPDAT**

Indicates that you intend to update an already existing file.

## Usage

1. An exit list address must be provided in each DCB specified by an RDJFCB macro.
2. Each exit list must contain an active entry that specifies the virtual storage address of the area into which the returned information is to be placed.
3. The exit list entry of 4 bytes must contain a X'07' in byte 0 indicating the 3 byte address, which follows is the address of the user buffer where the information will be returned.
4. The low-order 3 bytes contains the address of a 176 byte area to receive the returned information.
5. The area must be located within the user's storage, and must be located below the 16 MB line.
6. The virtual storage area into which the returned information is read must be at least 176 bytes long.
7. Each exit list entry must be 4 bytes long.
8. The system recognizes only the first occurrence of an X'07' exit list entry code.
9. The end of the exit list is indicated by setting the high order bit in the entry code to 1.

## Return Codes and ABEND Codes

Register 15 contains a return code from the RDJFCB macro, which is always 0.

Several conditions will cause the RDJFCB function to terminate abnormally with a X'240' ABEND. The reason codes and their meanings are as follows:

Hex Code	Decimal Code	Meaning
X'04'	4	Parameter list or DCB address is invalid.
X'08'	8	No EXLST address was found in the DCB.
X'0C'	12	No address exit was specified in the DCB exit list.
X'10'	16	The return information buffer is not within the user's storage.

Only fields provided by the RDJFCB function are programming interfaces.

- Some or all of the following fields may be filled in depending on characteristics of the data set at the time the RDJFCB is issued.

*Table 16. Information returned by RDJFCB for BSAM/QSAM*

Dec (Hex)	Description	Length
0 (0)	Data set name, left justified, padded with blanks.  File name 8 bytes File type 8 bytes File mode 2 bytes	44X
87 (57)	Indicator byte  X'C0' - Data set does not exist X'80' - DISP MOD specified on FILEDEF X'40' - Data set exists	1X
99 (63)	Data organization byte  X'00' - Not a VSAM data set.	1X
104 (68)	Logical record length	2X

*Table 17. Information returned by RDJFCB for VSAM*

Dec (Hex)	Description	Length
87 (57)	Indicator byte  X'C0' - Always set to new for VSAM.	1X

---

*Table 17. Information returned by RDJFCB for VSAM (continued)*

Dec (Hex)	Description	Length
99 (63)	Data organization byte.  X'08' - Indicates VSAM data set.	1X

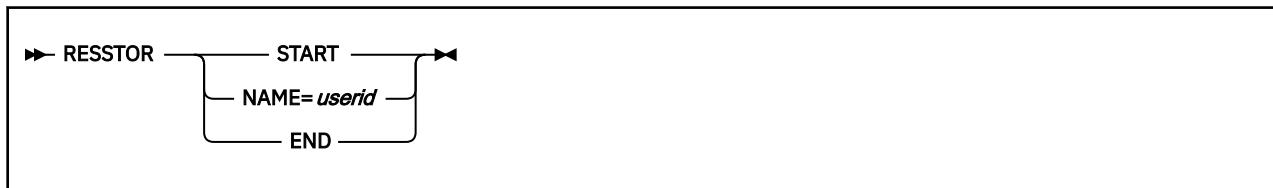
---

- Usage Note: A DCB must always be used with RDJFCB. If the DDNAME in the DCB is a VSAM data set and a DLBL has been issued, then the information returned will be as described under VSAM.

# RESSTOR

---

## Format



## Purpose

Use the RESSTOR macro to generate a list of the virtual machines in the GCS group that are to have storage reserved at IPL time for both the BAM and VSAM segments specified on the CONFIG macro.

## Parameters

### START

Indicates that this RESSTOR macro marks the beginning of the VSAM user block.

The RESSTOR user block must begin with a RESSTOR macro with this parameter specified.

### NAME

Specifies the user ID of the virtual machine that is to have storage reserved for the VSAM and BAM segments at IPL time.

### END

Indicates that this RESSTOR instruction marks the end of the RESSTOR user block.

The RESSTOR user block must end with a RESSTOR macro with this parameter specified.

## Usage

1. Most installations will not explicitly use the RESSTOR macro to build the GROUP CONFIGURATION FILE. Those equipped with at least one full-screen display terminal can take advantage of GCS build panels. These data entry panels, called by the GROUP command, eliminate the need to build the file by explicitly coding these macros. When you start the GROUP command without a full-screen terminal, your file will have to be built using the editor and coding the macros manually.
2. The GROUP CONFIGURATION FILE adopts the system name as its file name. This name corresponds exactly with that specified in the SYSNAME parameter of the CONFIG instruction. The file type of the GROUP CONFIGURATION FILE is always GROUP.
3. Remember that in using the RESSTOR macros you are creating blocks of information. Thus, all occurrences of the RESSTOR instruction must be physically grouped together in the GROUP CONFIGURATION FILE.

## Examples

This example illustrates the storage reserved user block of a GROUP CONFIGURATION FILE.

```

.
.
.
RESSTOR START
RESSTOR NAME=USERID1
RESSTOR NAME=USERID2
RESSTOR NAME=USERID3
RESSTOR NAME=USERID4
RESSTOR END
  
```

```
.
```

The block begins with the RESSTOR instruction with the START parameter specified. Four user IDs are then specified, indicating that these virtual machines are to have storage automatically reserved for the VSAM and BAM segments when they IPL. The storage reserved block is then concluded with an RESSTOR instruction with the END parameter specified.

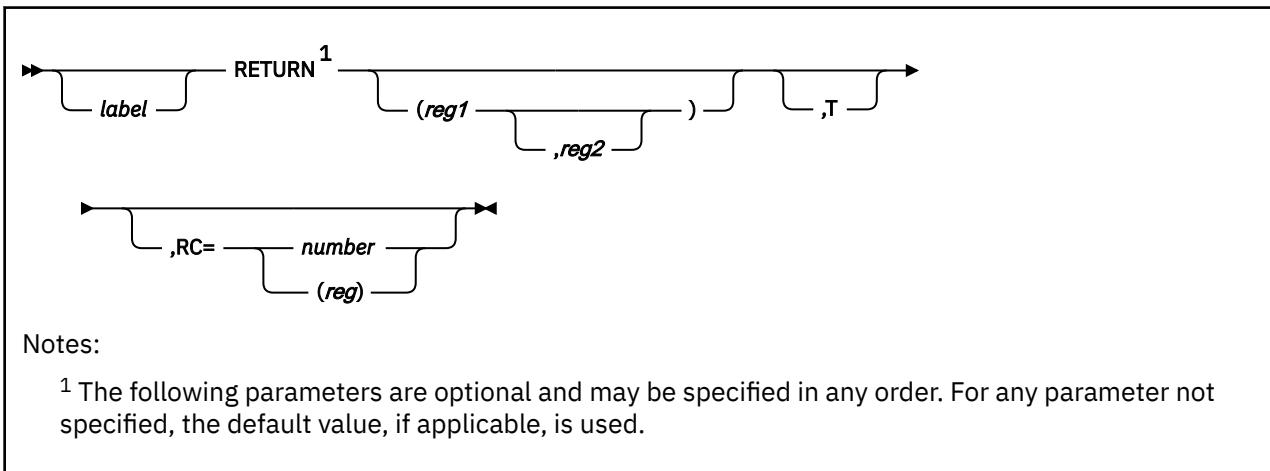
## Return Codes and ABEND Codes

The RESSTOR macro generates no return codes and no ABEND codes.

## RETURN

---

### Format



### Purpose

Use the RETURN macro to return control from one program to the program that called it.

The RETURN macro can also restore the contents of certain registers belonging to the program to which control is returning. It can also supply the program with a return code and flag the save area where the values of its registers were saved.

### Parameters

#### (reg1)

#### (reg1,reg2)

Specifies the single register, (reg1), or the range of registers, (reg1,reg2), whose values are to be restored from the save area.

The RETURN macro uses the same conventions for restoring registers that the SAVE macro uses. They are restored in the following general order: 14, 15, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12.

To specify a range of registers to be restored, substitute the first register in the range for the reg1 parameter. Then, substitute the last register in the range for the reg2 parameter. Obviously, a subset of the preceding general order is permissible, but remember the general order when specifying a range of registers.

Never specify register 13 as a register whose value is to be restored.

If you omit this parameter, no registers are restored.

#### T

Indicates that you want the save area, from which the register values are restored, to be flagged.

The flag indicates that the program issuing the RETURN instruction (which save the values) has returned control to the program that called it.

The flag itself is a byte of all ones placed in the high-order byte of word 4 in the save area.

#### RC

Specifies the return code to be passed to the program to which control is being returned.

The value of this return code has meaning only to the applications involved.

You can write this parameter as decimal digits, as an EQU symbol, or as register (15). If you write it as one or more digits or as a symbol, then the return code is right-justified in register 15 just before control is returned. If you write it as register (15), then the macro assumes that the program returning control has placed the return code in register 15. Register 15 will be left alone during the restoration of the other registers.

If you omit this parameter, the contents of register 15 will be determined by the reg1 or reg1,reg2 parameter.

## Usage

If registers are to be restored or if the save area is to be flagged, then register 13 must contain the address of the save area.

## Examples

```
GOBACK RETURN (14,7),T,RC=40
```

The program requests that control be returned to the program that called it. Registers 14, 15, and registers 0 through 7 are to be restored. A flag byte is to be placed in the save area, and a return code of 40 is to be placed in register 15. Note that the return code replaces the value that was just restored to register 15. GOBACK is the label on this instruction.

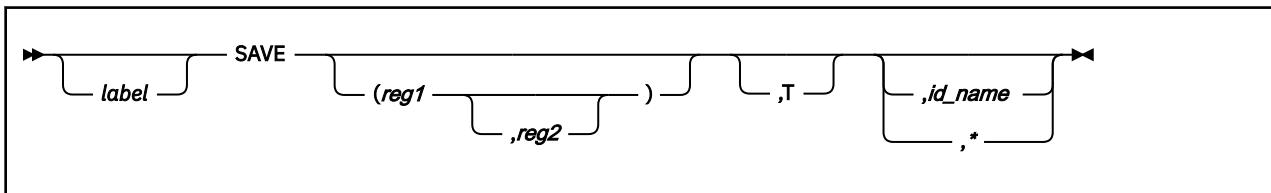
## Return Codes and ABEND Codes

The RETURN macro generates no return codes and no ABEND codes.

# SAVE

---

## Format



## Purpose

Use the SAVE macro in a called program to save the values of certain registers belonging to the program that called it.

By convention, it is the responsibility of any program called by another to save the values in the registers when it receives control. It is the responsibility of the calling program to provide storage where the values in its registers can be saved. The calling program must also place the address of this save area in register 13 before it calls the other program.

Note that the SAVE macro uses the standard conventions for saving registers. That is, they are saved in an area composed of 18 contiguous fullwords, starting at the fourth fullword. And, they are saved in the following general order: 14, 15, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12. Register 13 is never saved.

## Parameters

### **(reg1)**

### **(reg1,reg2)**

Specifies the single register (reg1) or range of registers (reg1,reg2) whose values are stored in the save area.

To specify a range of registers, consider the general order which registers are saved. Substitute the first register number in the range for the reg1 parameter. And, substitute the last register in the range for the reg2 parameter. Obviously, a subset of this order is permissible but remember the general order when specifying a range of registers.

Never specify register 13 as a register whose value is to be saved.

You must write the register or range of registers as decimal digits.

### **T**

Indicates that regardless of what other registers are saved or not saved, registers 14 and 15 are saved.

Use this parameter if you want to save the values in registers 14 and 15 and those in another subset of registers, of which 14 and 15 are not a part of. The other subset can be specified by the (reg1) or (reg1,reg2) parameter, while registers 14 and 15 are specified by the T parameter.

The T parameter can also be specified alone, indicating that only registers 14 and 15 are to be saved.

### ***id\_name***

Specifies an identifier or label that is to be associated with the SAVE macro.

You can use this identifier as a debugging aid when you enter several SAVE macros. You can assign this parameter a unique, mnemonic value that will be inserted in any dump you might request. This lets you associate a section within the dump with a specific SAVE macro, and thereby with a specific save area.

A byte containing the length of the ID NAME appears in the dump 4 bytes after the address in register 15. The ID NAME itself begins 5 bytes after this address.

\*

If you write the ID NAME parameter as an asterisk (\*), then the label on the SAVE instruction itself will be assigned to it. If you omit this parameter entirely, then the label on the appropriate CSECT instruction will be assigned to the ID NAME parameter. If no label appears on the CSECT instruction, then this parameter is ignored.

## Usage

The SAVE macro must be the first instruction at the entry point of any called program. This is because register 15 must contain the address of the macro, which it might not if the SAVE instruction was issued later.

## Examples

```
SAVE (14,12),,*
```

The program requests that the values in registers 14, 15, and registers 0 through 12 be saved. Because an asterisk is specified and no label appears on this instruction, the label on the appropriate CSECT instruction is assigned to the ID NAME parameter.

```
SAVE (5,7),T
```

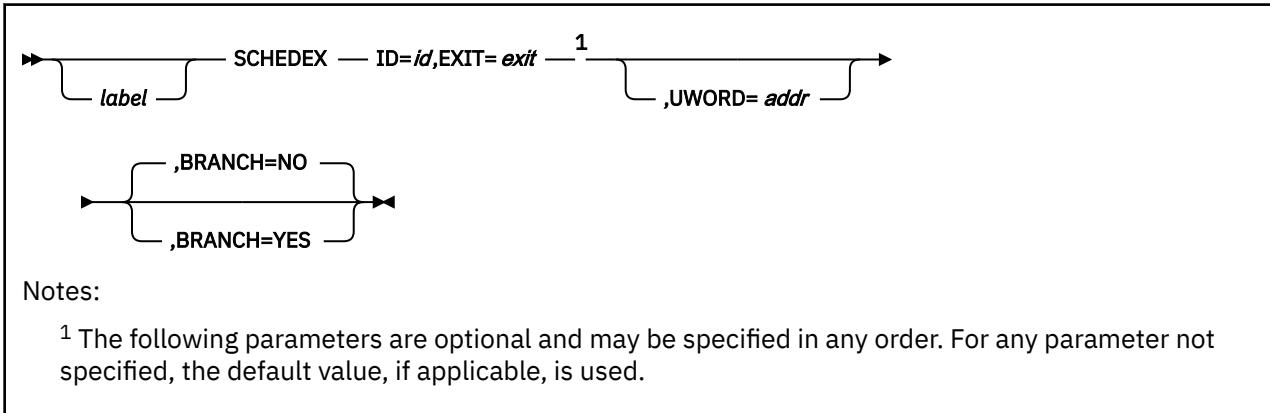
The program requests that the values in registers 5, 6, and 7 be saved. Because registers 14 and 15 are not within this range and because the program wants them saved, the T parameter is also specified.

## Return Codes and ABEND Codes

The SAVE macro generates no return codes and no ABEND codes.

# SCHEDEX

## Format



## Purpose

Use the SCHEDEX macro to schedule an exit to a specific task.

One feature of GCS is that it permits virtual machines to work together in virtual machine groups. A virtual machine group consists of several virtual machines sharing common storage and, usually, a common purpose. Within each virtual machine more than one task can be running simultaneously.

For a variety of reasons, one task may decide that it needs another task to perform work for it. The SCHEDEX macro will schedule an exit to that task. This means that the next time the second task is dispatched the exit receives control so it can perform the needed work.

The SCHEDEX macro is an authorized GCS function.

## Parameters

### ID

Specifies the identifier of the virtual machine that contains the task requesting the exit, and the identifier of the task to which an exit is to be scheduled.

This is a fullword parameter containing the virtual machine identification in the high-order halfword and the task identification in the low-order halfword.

If the task ID is zero, then the task identification will be the SYSTEM TASK, by default.

You can write this parameter as an assembler program label, as register (0), or as register (2) through (12). If you write it as a label, then the machine and task identifiers must be at the address associated with that label. If you write it as a register, then the machine and task identifiers must be in that register. In either case, GCS expects that they be in the proper format.

### EXIT

Specifies the address of the exit routine to be scheduled.

This routine must be in a shared segment that was linked to the virtual machine at GCS initialization time. After the task is dispatched, it receives control.

You can write this address as an assembler program label, as register (2) through (12), or as register (15).

### UWORD

Specifies an optional fullword parameter that can be passed to the exit routine in question.

You can use this parameter to pass any information you wish.

You can write this parameter as an assembler program label or as register (1) through (12). If you write it as an assembler program label, then the address of the label is passed to the exit routine. If you write it as a register number, then the contents of that register will be passed to the exit routine. If this parameter is not specified, then it is passed with a value of zero.

#### **BRANCH**

Specifies whether your task should branch directly to the SCHEDEX service routine.

##### **YES**

Specifies that your task should branch directly to the SCHEDEX service routine.

##### **NO**

Specifies that you want to use the customary SVC interface. This option is the default.

#### **Usage**

1. It is important to realize that the SCHEDEX macro does not turn control over to any task. It merely schedules an exit to the appropriate task, which receives control only when it has been dispatched.
2. The SCHEDEX macro is not limited to one virtual machine. Note that the purpose of the ID parameter is not only to identify the task in question but also the virtual machine which it resides. For example, TASK X, residing in VIRTUAL MACHINE A, can schedule an exit to TASK Y, which resides in VIRTUAL MACHINE B.
3. The task issuing the SCHEDEX macro resumes usual execution when it receives the return code from the macro. It does not wait for the scheduled exit routine to run but proceeds to its own next executable statement.
4. Any exit routine scheduled through the SCHEDEX macro runs in key 0.
5. A zero return code from the SCHEDEX macro does not necessarily mean that the exit has been scheduled. The request has been sent to CP. If the virtual machine where the exit resides is part of the group, then the exit will be scheduled.
6. If you specify BRANCH=YES, your task must be in supervisor state, key 0, and disabled for interrupts.

An interrupt handler cannot use the branch interface to the SCHEDEX service routine.

Register 2 contains the address of the exit routine entry point. When BRANCH=YES, the macro destroys the previous contents of register 2. You may want to save and, later, restore the contents of register 2.

Before the branch, register 13 must contain the address of a 72-byte register save area.

Register 15 contains the address of the SCHEDEX entry point.

Because branching directly to the SCHEDEX service routine avoids the supervisor call, no trace entry for the macro is generated.

7. The AMODE of the exit will be taken from the correspondent entry of the CONTENTS macro. See [“CONTENTS” on page 197](#).

#### **Examples**

```

SCHEDEX ID=IDENT,EXIT=(3)
.
.
.
IDENT    DC  H'2'
DC  H'4'

```

Schedule an exit on the virtual machine, whose machine ID is 2, and on a task therein, whose task ID is 4. The address of the routine to receive control is in register 3.

The program to which an exit is scheduled receives the following information in its registers.

Register	Contents
1	The user word (UWORD) specified in the SCHEDEX macro.
13	The address of the register save area.
14	The address to which control is to return after the exit program completes execution.
15	The address of the entry point in the exit program.

## Return Codes and ABEND Codes

When this macro completes processing, it passes to the caller a return code in register 15.

Hex Code	Decimal Code	Meaning
X'00'	0	Your request has been sent to CP.
X'04'	4	The virtual machine identifier that you specified was invalid.
X'08'	8	The address of the exit routine you specified is not in a shared segment.
X'0C'	12	The task identifier is invalid. This return code is significant only if the exit is scheduled to run on your virtual machine.
X'30'	48	The CONTENTS entry has AMODE=DEFINED or AMODE=CALLER, the caller is in AMODE 24 and the exit routine address is above the 16MB line.

## ABEND Code Reason Code Meaning

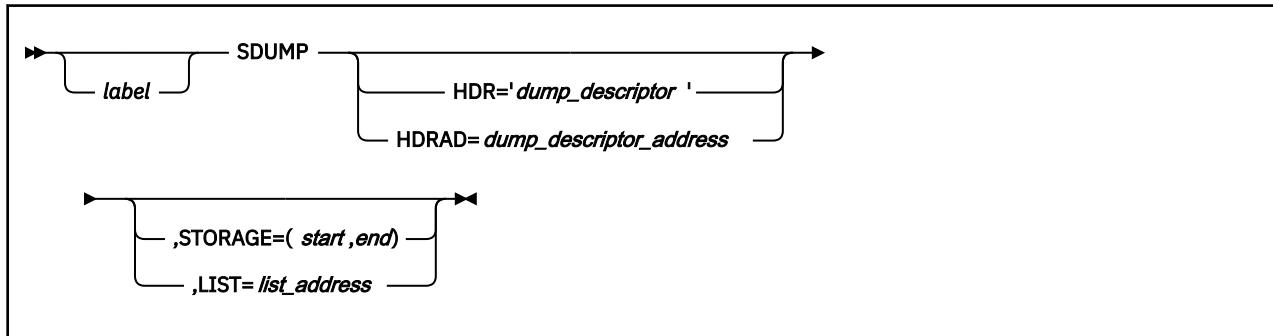
ABEND Code	Reason Code	Meaning
OF8	16	The GCS supervisor was called in access register mode.
FCB	0A01	Insufficient storage was available to satisfy a GETMAIN instruction that the system issued.

## SDUMP

The SDUMP macro is available in standard, list and execute formats.

### Standard Format

See also “List Format” on page 331 and “Execute Format” on page 331.



### Purpose

Use the SDUMP macro to request a recording of the contents of your virtual machine's storage. A dump is a recording of the contents of a virtual machine's storage at a given moment. Use the SDUMP macro to produce a dump of part or all of your virtual machine's storage.

### Parameters

#### HDR

Specifies a string of characters that you can use to describe the dump. The dump description will be displayed by the DVF DUMPSCAN 'DUMPID' subcommand.

This character string is placed in the dump to help you to identify it quickly. This string can contain up to 100 characters and must be surrounded by single quotation marks.

#### HDRAD

Specifies the address of a string of characters you stored previously that describe the dump. The dump description will be displayed by the DVF DUMPSCAN 'DUMPID' subcommand.

This character string is placed in the dump to help you to identify it quickly. This string can contain up to 100 characters. The first byte at this address must contain the hexadecimal length of the character string and no single quotation marks are required.

You can write this parameter as an assembler program label or as register (2) through (12).

#### STORAGE

Specifies the range of virtual storage addresses to be recorded in the dump.

**Note:** From the format illustration each pair of addresses must be separated by a comma and enclosed in parentheses. You can specify more than one range of addresses if you wish. Just be certain that each starting address is less than its corresponding ending address.

#### LIST

Specifies the address of a list that contains one or more pairs of addresses. Each pair of addresses in the list specifies a range of virtual storage addresses to be included in the dump.

This list can contain up to 2049 different pairs of addresses, which can overlap each other. If they do, then CP will resolve two or more overlapping pairs into one pair.

The high-order bit of the fullword containing the last ending address in the list must be set to 1 to indicate the end of the list. All other high-order bits in the list must be reset to 0.

You can write this parameter as an assembler program label or as register (2) through (12).

## Usage

1. If both the STORAGE and LIST parameters are omitted from the SDUMP macro, then GCS assumes that all virtual storage in the machine is recorded in the dump. This includes any discontiguous saved segments the virtual machine may be using.
2. It is important to understand the rules governing who receives the spool file containing the dump and what that file contains.

For security reasons, not every user is authorized to receive dumps containing fetch-protected data. Those who are authorized are listed among the authorized users at GCS build time. If a common dump receiver was specified at GCS build time, then that individual receives the dump. Otherwise the issuer of the SDUMP macro receives the dump.

Bear in mind that if the person receiving the dump is not authorized to handle fetch-protected data, that data will be omitted from the dump. However, all requested non-fetch-protected data and private key 14 storage will be included in the dump.

3. No dump will be produced if dumps are suppressed through the SET DUMP OFF command.

## Examples

```
DUMPALL SDUMP HDR='ALL MY STORAGE'
```

A dump of the entire virtual machine's storage is requested. The character string ALL MY STORAGE is placed in the dump for ready identification. The dump is sent to the member of the virtual machine group authorized to receive it. If no one is so authorized, then the dump is sent to the issuer of the macro. Fetch-protected data will be included in the dump only if the recipient is authorized to handle such data. DUMPALL is the label on this instruction.

```
DUMPALL SDUMP HDR='THREE STORAGE AREAS',STORAGE=(A,B,C,D,E,F)
```

A dump of certain portions of virtual storage is requested. Each pair of labels identifies the start and end addresses for each storage area to be dumped. This example shows 3 storage areas being requested, but you can request from 1 to 2049 different pair of addresses.

```
SDUMP HDRAD=(5),LIST=RANGES
```

Another dump of certain portions of virtual storage is requested. The address of a string of characters describing the dump can be found at the address in register 5. The first byte of register 5 must contain the length of the character string, in hexadecimal. This character string is to be placed in the dump for ready identification. A list containing at least one pair of addresses can be found at the address associated with the label RANGES. Each pair of addresses in the list specifies a range of virtual storage addresses to be included in the dump. Presumably the high-order bit of the last ending address has been set to 1 to indicate the end of the list.

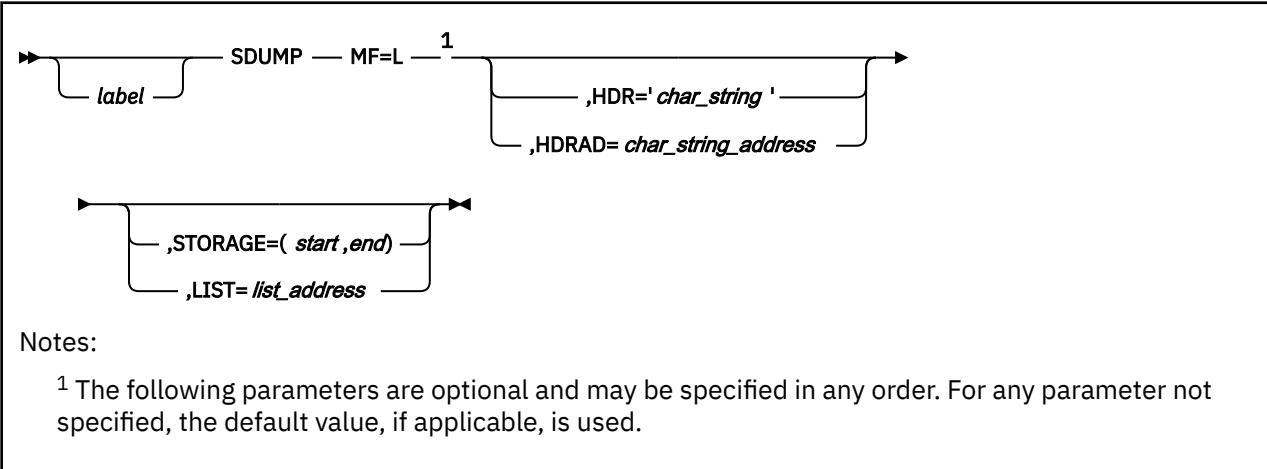
## Return Codes and ABEND Codes

When this macro completes processing, it passes a return code to the caller in register 15.

Hex Code	Decimal Code	Meaning
X'00'	0	All requested areas have been included in the dump.
X'04'	4	Only a portion of the requested areas was included in the dump.

Hex Code	Decimal Code	Meaning
X'08'	8	A range beyond the last byte of virtual storage was requested, or SET DUMP OFF has been issued. (GCS was unable to produce a dump).
<b>ABEND Code Reason Code Meaning</b>		
233	8	Invalid parameter list address.

## List Format



## Purpose (List Format)

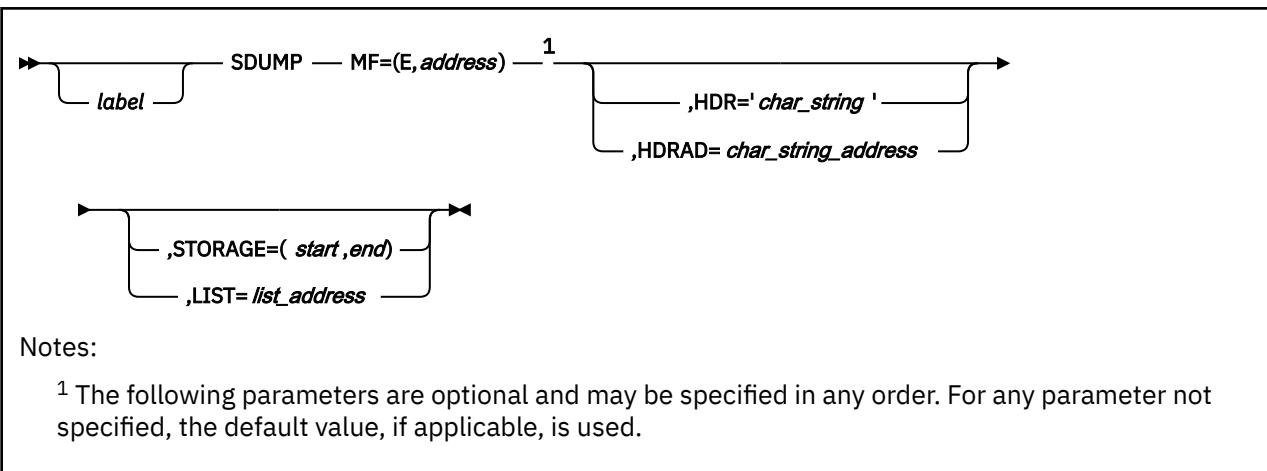
This format of the macro generates an in-line parameter list based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation.

## Added Parameter

### MF=L

Specifies the list format of this macro.

## Execute Format



**Purpose (Execute Format)**

This format of the macro generates code that executes the function, using a parameter list whose address you specify.

**Added Parameter (Execute Format)****MF=(E,*address*)**

*address* specifies the address of the parameter list to be used by the macro.

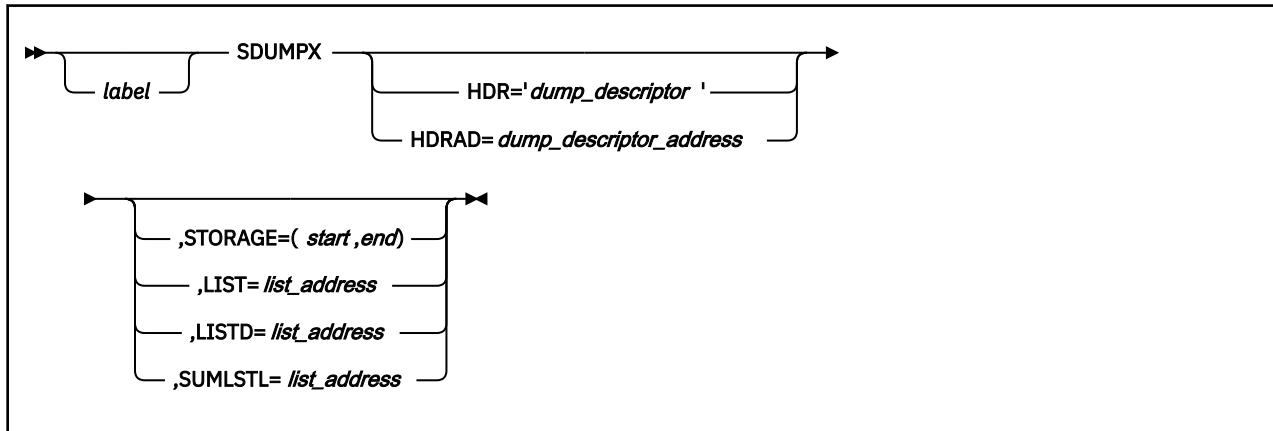
You can add or modify values in this parameter list by specifying them in this macro.

# SDUMPX

The SDUMPX macro is available in standard, list and execute formats.

## Standard Format

See also “List Format” on page 331 and “Execute Format” on page 331.



## Purpose

Use the SDUMPX macro when you are running in an XC virtual machine and want to dump part or all of a data space that you are accessing. A dump is a recording of the contents of a virtual machine's storage at a given moment.

## Parameters

### HDR

Specifies a string of characters that can be used to describe the dump. The dump description will be displayed by the DVF DUMPSCAN "DUMPID" subcommand when either the STORAGE, LIST, or LISTD, operand of SDUMPX is used to produce the dump.

This character string is placed in the dump to help you to identify it quickly. This string can contain up to 100 characters and must be surrounded by single quotation marks.

### HDRAD

Specifies the address of a string of characters you stored previously that describe the dump. The dump description will be displayed by the DVF DUMPSCAN "DUMPID" subcommand when either the STORAGE, LIST, or LISTD, operand of SDUMPX is used to produce the dump.

This character string is placed in the dump to help you to identify it quickly. This string can contain up to 100 characters. The first byte at this address must contain the hexadecimal length of the character string and no single quotation marks are required.

You can write this parameter as an assembler program label or as register (2) through (12).

### STORAGE

Specifies the range of virtual storage addresses to be recorded in the dump.

**Note:** From the format illustration each pair of addresses must be separated by a comma and enclosed in parentheses. You can specify more than one range of addresses. Just be certain that each starting address is less than its corresponding ending address.

**LIST**

Specifies the address of a list that contains one or more pairs of addresses. Each pair of addresses in the list specifies a range of virtual storage addresses to be included in the dump.

This list can contain up to 2049 different pairs of addresses, which can overlap each other. If they do, then CP will resolve two or more overlapping pairs into one pair.

The high-order bit of the fullword containing the last ending address in the list must be set to 1 to indicate the end of the list. All other high-order bits in the list must be reset to 0.

You can write this parameter as an assembler program label or as register (2) through (12).

**LISTD**

Specifies the address of a list of address ranges, qualified by an ASIT of 8 bytes that reference the data space you want to dump. The ASIT uniquely identifies the data space within the scope of the VM system. The ASIT is similar to the STOKEN of MVS/ESA™.

**Specify the ASIT and ranges as follows:**

The following is the format of the input list

<i>Table 18. SDUMPX LISTD parameter list format</i>
<b>4 byte fields</b>
Length of parameter list
First ASIT ( 8 bytes )
Number of ranges to be dumped for this ASIT
Range 1 starting address
Range 1 ending address
Range n starting address
Range n ending address
Last ASIT ( 8 bytes )
Number of ranges to be dumped for this ASIT
Range 1 starting address
Range 1 ending address
Range n starting address
Range n ending address

The first fullword of the list contains the number of bytes (including the first word) in the list. ASIT refers to any address/data space. SDUMPX does not dump data space storage that has not been referenced.

**SUMLSTL**

Specifies the address of a list of address ranges, qualified by an ALET of 4 bytes. The ALET is obtained by use of the ALSERV macro which is used to add a data space to a list of data spaces that can be accessed by a given virtual machine. The ALET that is returned from that request can be used to establish addressability to the data space.

The LISTD and SUMLSTL parameters are RX-type addresses or registers (2) - (12).

**Specify the ALET and ranges as follows:**

The following is the format of the input list

*Table 19. SDUMPX SUMLSTL parameter list format.*

#### 4 byte fields

Length of list

First ALET ( 4 bytes )

Number of ranges (n) to be dumped for this ALET

Range 1 starting address

Range 1 ending address

Range n starting address

Range n ending address

Last ALET ( 4 bytes )

Number of ranges (n) to be dumped for this ALET

Range 1 starting address

Range 1 ending address

Range n starting address

Range n ending address

## Usage

1. A dump of all storage is received if no operands are used with the SDUMPX macro.
2. SDUMPX cannot be used from Access Register (AR) mode.
3. It is important to understand the rules governing who receives the spool file containing the dump and what that file contains.

For security reasons, not every user is authorized to receive dumps containing fetch-protected data. Those who are authorized are listed among the authorized users at GCS build time. If a common dump receiver was specified at GCS build time, then that individual receives the dump. Otherwise the issuer of the SDUMPX macro receives the dump.

Bear in mind that if the person receiving the dump is not authorized to handle fetch-protected data, that data will be omitted from the dump. However, all requested non-fetch-protected data and key 14 storage, will be included in the dump.

4. No dump will be produced if dumps are suppressed through the SET DUMP OFF command.

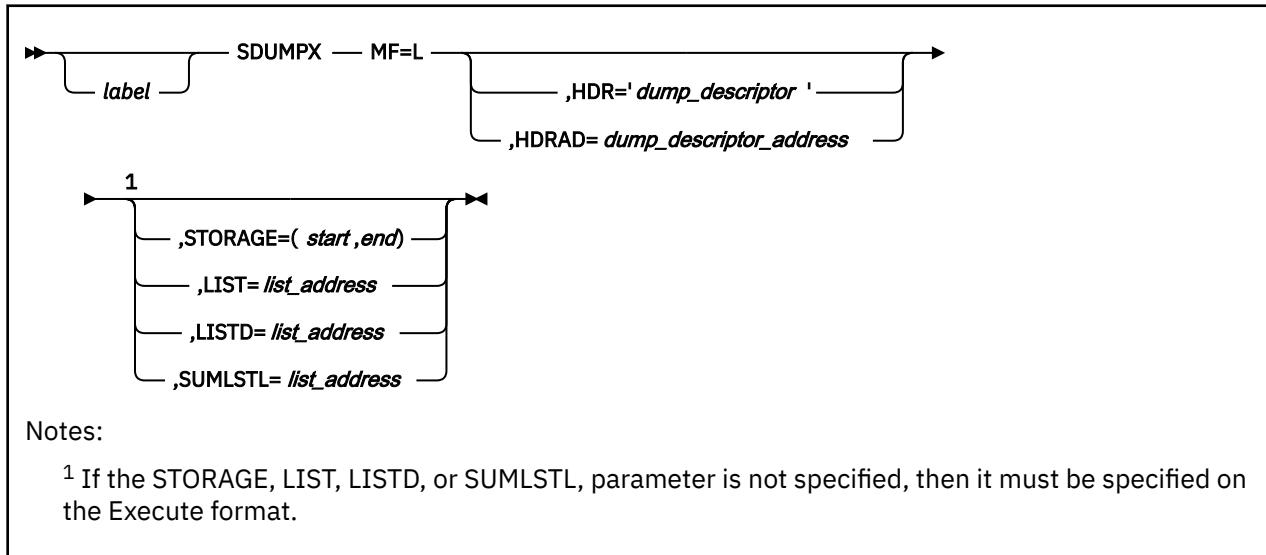
## Messages

When this macro completes processing, it passes a return code to the caller in register 15.

Hex Code	Decimal Code	Meaning
X'00'	0	All requested areas have been included in the dump.
X'04'	4	Only a portion of the requested areas was included in the dump.
X'08'	8	A range beyond the last byte of virtual storage was requested, a data space was not properly addressed, the SDUMPX macro was issued from a non-XC virtual machine, or SET DUMP OFF has been issued. (GCS was unable to produce a dump).

**ABEND Code Reason Code Meaning**

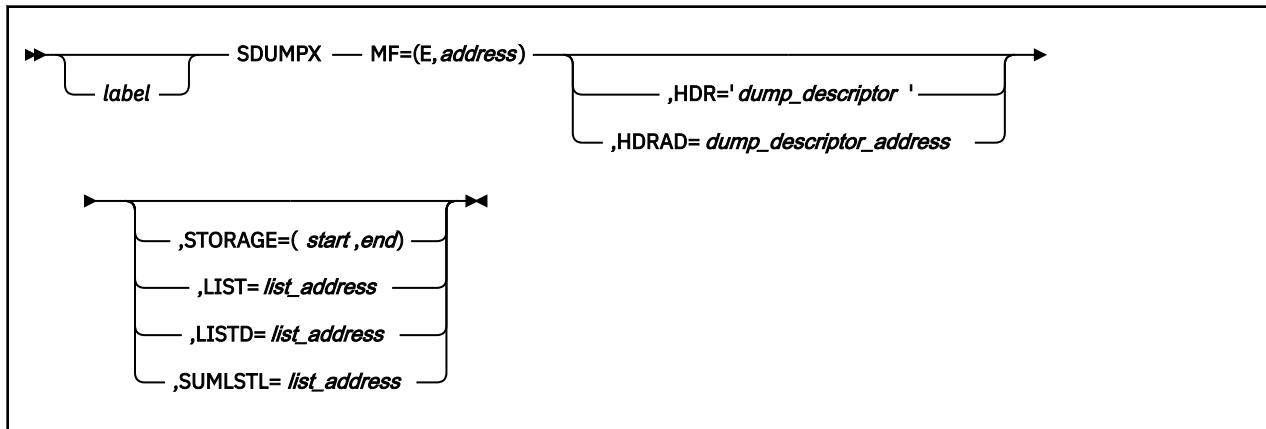
233	4	Invalid parameter list structure.
233	8	Invalid parameter list address.

**List Format****Purpose (List Format)**

This format of the macro generates an in-line parameter list based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation.

**Added Parameter****MF=L**

Specifies the list format of this macro.

**Execute Format**

## Purpose (Execute Format)

This format of the macro generates code that executes the function, using a parameter list whose address you specify.

## Added Parameter (Execute Format)

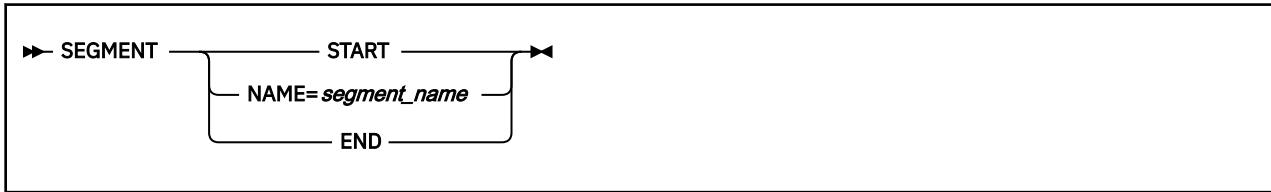
### MF=(E,*address*)

*address* specifies the address of the parameter list to be used by the macro.

You can add or modify values in this parameter list by specifying them in this macro.

# SEGMENT

## Format



## Purpose

Use the SEGMENT macro to define which saved segments will be linked to each member of a virtual machine group.

The GROUP CONFIGURATION FILE describes a GCS virtual machine group. This file is divided into data blocks:

### Configuration

Defines the virtual machine group's configuration so that it conforms to the needs of your installation. See "[CONFIG](#)" on page 193.

### Segment

Identifies which saved segments will be automatically linked to each member of the group at IPL time.

### Authorized User

Identifies which members of the group are *authorized*. That is, which members are permitted to perform authorized GCS functions. The Authorized GCS Service Macros are identified in [Chapter 5, "GCS Macros," on page 157](#).

Use the SEGMENT macro to create a segment block for the GROUP CONFIGURATION FILE.

## Parameters

### START

Indicates that this SEGMENT macro marks the beginning of the segment block.

The segment block must begin with a SEGMENT macro with this parameter.

### NAME

Specifies the name of a saved segment that is to be linked automatically to each member of the virtual machine group at IPL time.

### END

Indicates that this SEGMENT macro marks the end of the segment block.

The segment block must end with a SEGMENT macro with this parameter.

## Usage

1. Most installations will not explicitly use the SEGMENT macro to build the GROUP CONFIGURATION FILE. Those equipped with at least one full-screen display terminal can take advantage of GCS build panels. These data entry panels, started by the GROUP command, eliminate the need to explicitly code these macros. However, without a full-screen terminal, your file will have to be built using the editor and coding the macros manually.
2. The GROUP CONFIGURATION FILE adopts the system name (virtual machine group name) as its file name. Its file type is always GROUP.

3. By using the SEGMENT macro, you are creating blocks of information. Thus, all occurrences of the SEGMENT macro must be physically grouped together in the GROUP CONFIGURATION FILE.

### Examples

This example illustrates the segment block portion of a GROUP CONFIGURATION FILE.

```
.  
. .  
SEGMENT START  
SEGMENT NAME=SS5  
SEGMENT NAME=SS8  
SEGMENT NAME=SS11  
SEGMENT NAME=SS17  
SEGMENT END  
. .
```

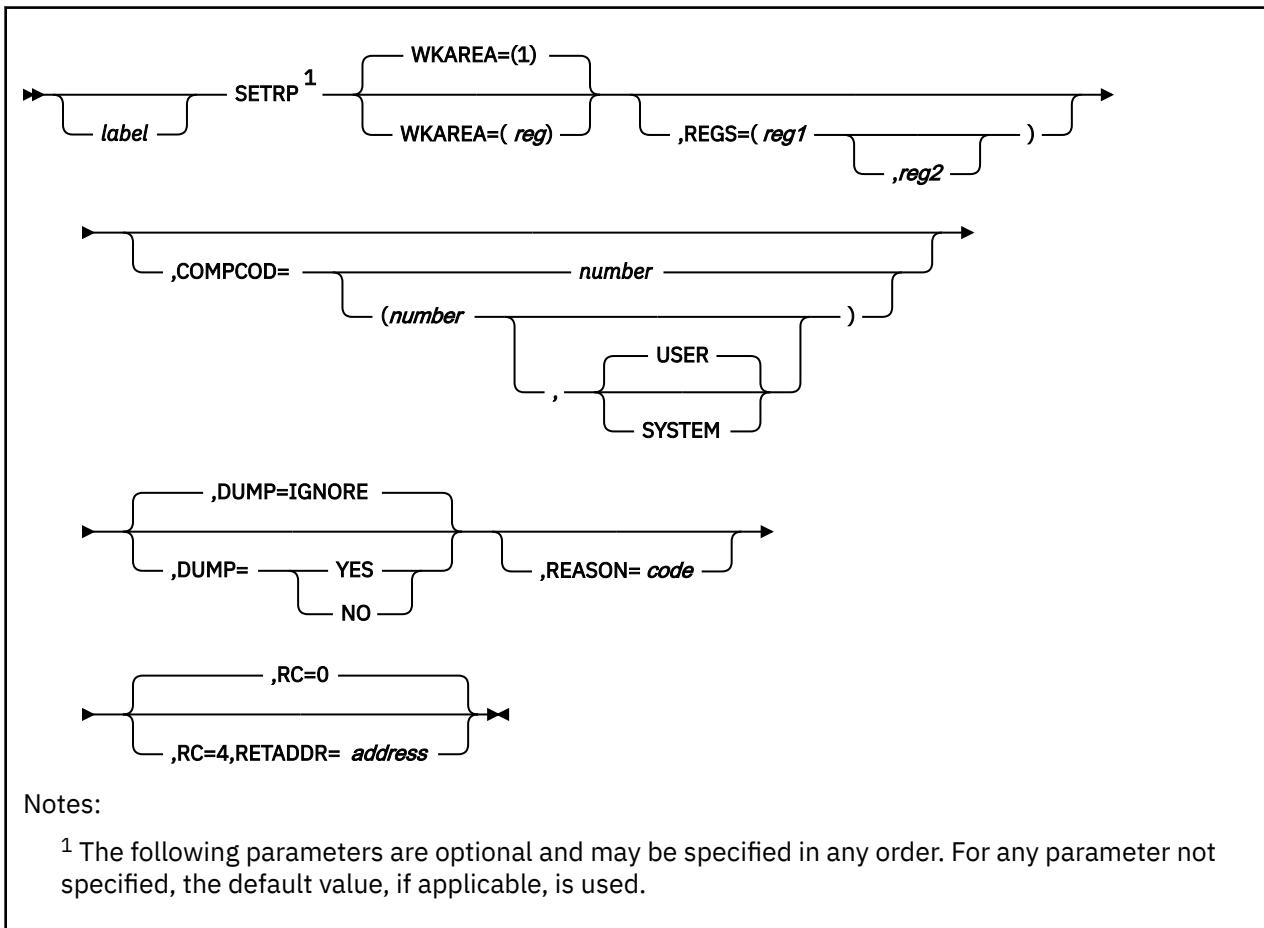
The block begins with the SEGMENT macro with the START parameter. The names of four saved segments, which are to be linked automatically to every virtual machine group member, are then specified. The segment block then concludes with a SEGMENT macro with the END parameter.

### Return Codes and ABEND Codes

The SEGMENT macro generates no return codes and no ABEND codes.

## SETRP

### Format



### Purpose

Use the SETRP macro to set certain parameters in the system diagnostic work area.

Often an application identifies an exit routine for each task that will receive control if the task ends abnormally. See “[ESTAE](#)” on page 223.

When the ABEND macro is issued for a specific task, a system diagnostic work area (SDWA) is created. See “[ABEND](#)” on page 162.

The SDWA is an area of storage that contains important information about the task that has just ended abnormally. The exit routine uses this information to analyze the problem. To appreciate the SETRP macro fully, you should also have a sound understanding of the IHASDWA macro. Review the entry titled “[IHASDWA](#)” on page 273.

Use the SETRP macro in an exit routine that you defined through the ESTAE macro. The SETRP macro will set (or reset) certain parameters in the SDWA. Prominent among these is the RC parameter. This will let GCS know whether your recovery routine should get control and try to revive your task.

### Parameters

**WKAREA**

Specifies the address of the system diagnostic work area that will be passed to your recovery routine. If you omit this parameter, then the address of the SDWA must be in register 1. Otherwise, you can write this parameter as register (1) through (12).

**REGS**

Specifies the single register (reg1) or range of registers (reg1,reg2) belonging to the failed task, whose values are to be restored from the save area pointed to by register 13.

To specify a range of registers, consider the order in which registers are saved: 14, 15, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12. Substitute the first register number in the range for the reg1 parameter. And, substitute the last register in the range for the reg2 parameter. A subset of this order is permissible.

Never specify register 13 as a register whose value is to be restored.

If you specify this parameter, then, when it is finished, your exit routine will branch to the address in register 14, which you designated through the ESTAE macro. This will return control to the GCS supervisor. If you omit this parameter, then no such branch will be taken, making it your responsibility to code the return from your exit routine.

You can write the register or range of registers as decimal digits.

**COMPCOD**

Specifies the completion code that will overlay the current completion code in the SDWA.

This completion code must be a number from 0 to 4095. The meaning of each completion code is governed by your application.

You can write this parameter as a symbol, as decimal digits, or as register (2) through (12).

**USER**

Indicates that the completion code specified is defined by the user or the application. Unless otherwise stated, this is the case, by default.

**SYSTEM**

Indicates that the completion code specified is defined by the GCS supervisor.

**DUMP**

Indicates whether you want a dump produced containing the contents of the virtual machine which the ABENDED task was running.

GCS will send the dump to the virtual reader belonging to the member of your virtual machine group designated to receive dumps. If this member is not authorized, then only nonfetch-protected and key 14 data will be included in the dump.

**IGNORE**

Indicates that you do not want this SETRP macro to change any dump specification made by a previous SETRP or ABEND macro. That is, whatever any previous SETRP or ABEND macro said about producing or not producing a dump will remain in force.

This is the case, by default.

**YES**

Indicates that a dump of the virtual machine which the ABENDED task was running will be produced.

**NO**

Indicates that no such dump will be produced.

Both the YES and NO parameters override any dump specification made by a previous SETRP or ABEND macro.

**REASON**

Specifies the code associated with the invocation of the ABEND exit. The code can be any 4-byte number specified in decimal (31-bit) or hexadecimal (32-bit).

**RC**

Specifies the return code that the exit routine you specified through the ESTAE macro will pass to your recovery routine. This return code describes what your recovery routine should do.

**0**

Indicates that GCS should continue to terminate the ABENDED task. This is the case, by default.

**4**

Indicates that GCS should give control to your retry routine, which will attempt to process the ABENDED task again.

**RETADDR**

Specifies the address in the ABENDED task that will receive control when the attempt to retry it is made.

This parameter is valid only if RC=4 is also specified.

You can write this parameter as an RX-type address or as register (2) through (12).

**Examples**

```
RETRY SETRP WKAREA=(3),REGS=(14,12),COMPCOD=635,RC=4,RETADDR=(12)
```

The task requests that certain fields in the SDWA be set and that the failed routine be tried again. The address of the SDWA is in register 3. Registers 14, 15, and 0 through 12, belonging to the failed routine, are to be restored. A user completion code of 635 is to overlay the completion code field in the SDWA. The RC=4 parameter indicates that the failed routine, at the address in register 12, should be tried again. RETRY is the label on this macro.

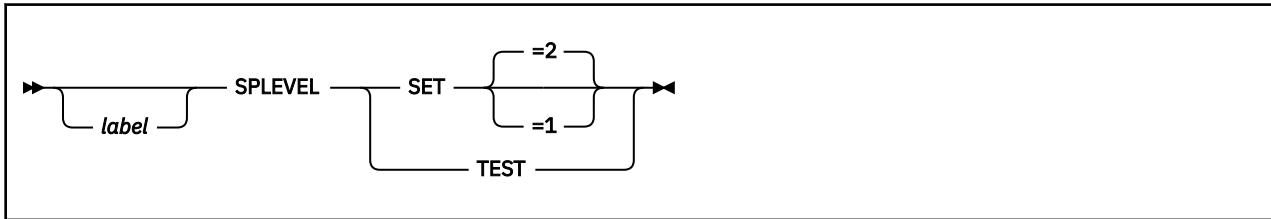
**Return Codes and ABEND Codes**

The SETRP macro generates no return codes and no abend codes.

# SPLEVEL

---

## Format



## Purpose

Use the SPLEVEL macro to set or test the macro level.

Some macros in the GCS macro library have expansions which will not function in programs running under the 370 Accommodation Facility because of incompatible parameter lists. SPLEVEL makes it possible to generate compatible macro parameter lists for these programs.

The macros interrogate a global symbol (set by SPLEVEL) during assembly to determine the type of expansion to generate. For more information on macros see [“GCS Macro Level and Parameter Lists” on page 157](#) and for information about global set symbols, see *Assembler H Version 2 Application Programming: Language Reference*.

## Parameters

### SET

Specifies the value of the global set symbol which will be used to determine the format of the macro's parameter list expansion.

#### =1

Macro expansion results in 24 bit addresses below the 16MB line.

#### =2

Macro expansion results in 31 bit addresses for use both above and below the 16MB line.

### SET

When SET is specified without a value, its default value is assumed of 2.

### TEST

Specifies that SPLEVEL should determine the macro level that is in effect. The results of the test request are returned in the global set symbol &SYSSPLV, which is defined by *GBLC* &SYSSPLV.

If SPLEVEL SET has not been previously issued during the assembly, the installation default value is inserted into the global set symbol. If SPLEVEL SET has been previously issued, the previous value as specified by n or the default value is already in the global set symbol.

## Usage

The default value obtained when a SET value is not specified is 2. The default value can be changed to 1 for a particular installation.

## Examples

```
SPLEVEL SET=1
```

Set the global set symbol to indicate GCS SP compatible macro expansion should occur.

SPLEVEL SET

Set the global set symbol to the default.

```
SPTEST    GBLC    &SYSSPLV  
          CSECT  
          ...  
          SPLEVEL TEST  
          DC      C'&SYSSPLV'
```

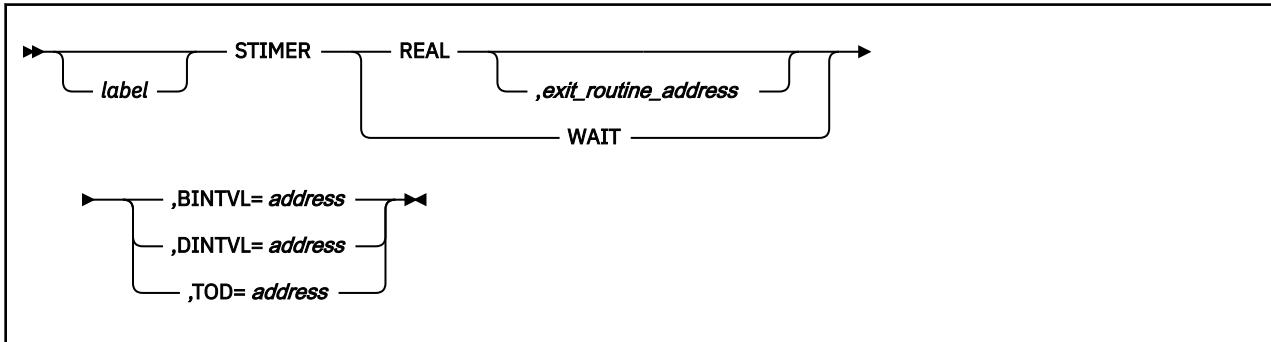
Test to determine the macro level in effect and place the results in &SYSSPLV. Set the global set symbol if no previous SPLEVEL has been issued.

## **Return Codes and ABEND Codes**

The SPLEVEL macro generates no return codes and no ABEND codes.

# STIMER

## Format



## Purpose

Use the STIMER macro to set a timer to a given time period. When time is up, your task will be notified.

At times, a task reaches a point where it needs to have something done for it. The task allocates a certain time period during which it waits for some event to occur. When told that time is up, the task resumes execution.

At other times, a task may be able to continue with other work while waiting for some event to take place. Having allocated a certain time period for this event, the task needs to be told when time is up.

To keep track of these time periods, a task sets a timer, specifying the amount of time it will allow for a certain event to take place.

## Parameters

### REAL

Indicates that the task will continue with other work while waiting for the specified time to elapse.

### *exit\_routine\_address*

Specifies the address of an exit routine that will get control at the end of the interval.

This exit routine must be resident in virtual storage and can be specified only with the REAL parameter.

The exit will always be in the AMODE of the caller.

You can write this as an RX-type address, as register (0), or as register (2) through (12).

### WAIT

Indicates that the task is to be placed in the WAIT state during the specified time period. At the end of the time period, the task will resume execution.

### BINTVL

Specifies the address containing the duration of time allocated for the event.

You must store the amount of time as an unsigned 32 bit binary number in a fullword on a fullword boundary. The low-order bit is equivalent to 0.01 seconds.

You can write this parameter as an RX-type address or as register (1) through (12).

For example, to define a BINTVL you would code:

BINTIME	DS 0F	Fullword Boundary
	DC	B'000000000000000000000000111111111'

**DINTVL**

Specifies the address containing the duration of time allocated for the event.

You must store the amount of time as EBCDIC characters in the range 0-9 on a doubleword boundary in the following format:

**HHMMSS00**

**HH** stands for the number of hours.

**MM** for the number of minutes.

**SS** for the number of seconds.

The maximum amount of time you can specify is 24 hours.

You can write this parameter as an RX-type address or as register (1) through (12).

For example, to define a DINTVL you would code:

DECTIME	DS      0D	Doubleword Boundary
	DC      CL8 '00000500'	Wait 5 Seconds

**TOD**

Specifies the address containing the time of day that marks the end of the time period.

You must store this time of day as EBCDIC characters in the range 0-9 on a doubleword boundary in following format:

**HHMMSS**

**HH** stands for the number of hours.

**MM** for the number of minutes.

**SS** for the number of seconds.

The maximum amount of time you can specify is 24 hours.

For example, to define a TOD (Time-of-day) time value you would code:

TODTIME	DS      0D	Doubleword Boundary
	DC      CL6 '084805'	Wait until 08:48:05 AM

**Usage**

1. It is the responsibility of the task issuing the STIMER macro to provide storage. The task must see to it that the appropriate time value is stored there before issuing the STIMER macro.
2. If you choose the REAL parameter and you do not specify the address of an exit routine, your task will never know the time has expired. In such a case, the supervisor does not notify your task that time is up.
3. The exit routine is responsible for saving and restoring your task's registers. It also executes in the same state and key as did your task when the latter issued the STIMER macro. After your exit routine completes execution, it returns control to the supervisor.

Input to the exit routine is:

Register	Contents
0 - 12	Unpredictable.
13	The address of a supervisor-provided save area.
14	The address to which control will transfer after the exit routine completes processing.
15	The address of the exit routine.

4. No task can have more than one timer set at the same time. If you enter an STIMER macro before the time period associated with a previous STIMER macro expires, then the second STIMER macro cancels and replaces the first.
5. All time is measured continuously in real time.
6. The SPLEVEL macro need not be issued unless you want a STIMER macro used by GCS that has an expanded parameter list, which is designed for use in the 31-bit addressing mode. A 31-bit parameter list is incompatible if you are running under the 370 Accommodation Facility. However the SPLEVEL macro lets you select either the 24-bit version or the 31-bit version
7. This macro supports both 24 and 31-bit address expansions of the parameter list. The macro expansion is controlled by the internal macro SPLEVEL. The default value is 31.

### Examples

```
CLOCKIT STIMER REAL,(6),TOD=(7)
```

The task wishes to set a timer. Because the REAL parameter is specified, the task will continue with other work while it is waiting. The specific time of day marking the end of the time period is stored at the address in register 7. When time is up, the exit routine, whose address is in register 6, receives control. CLOCKIT is the label on this instruction.

```
STIMER WAIT,DINTVL=(5)
```

The task wishes to set a timer. Because the WAIT parameter is specified, the task will be placed in the WAIT state until time is up. The amount of time, stored as characters, can be found at the address in register 5.

### Return Codes and ABEND Codes

The STIMER macro generates no return codes.

<b>ABEND Code</b>	<b>Meaning</b>
12F	Your task is in problem state and the parameter list for the macro is not in the same key as the task. You may also have incorrectly specified the DINTVL or TOD parameter. These must be in unpacked decimal format.
E2F	A parameter unsupported by GCS was specified. Unsupported parameters include TASK, GMT, TUINTVL, and MICVL.

## SYMREC

The SYMREC macro is available in standard, list and execute formats.

### Standard Format

See also “List Format” on page 348 and “Execute Format” on page 348.

```

    ➤————— SYMREC — SR=addr —>
      |————— label —————|
  
```

### Purpose

The SYMREC macro updates the symptom record with environment information. This macro is provided for users and is not used by GCS. The symptom record is a data area in the user's application that has been mapped in the ADSR macro and that is referenced by the SYMREC macro. The data in the symptom record is a description of a programming error and a description of the environment which the error occurred.

### Parameters

#### SR=addr

Specifies the address of the symptom record. The SR keyword is required. *addr* may be an A-type address or registers (2) through (12).

### List Format

```

    ➤————— SYMREC — SR=addr, MF=L —>
      |————— label —————|
  
```

### Purpose (List Format)

This format of the macro generates an in-line parameter list. It generates no executable code.

### Added Parameter

#### MF=L

Specifies the list format of this macro.

### Execute Format

```

    ➤————— SYMREC — SR=addr, MF=(E, address) —>
      |————— label —————|
  
```

## Purpose (Execute Format)

This format of the macro generates code that executes the function using a parameter list whose address you specify.

## Added Parameter (Execute Format)

### **MF=(E, address)**

Specifies the execute format of the SYMREC macro.

## Return Codes and ABEND Codes (Execute Format)

The following return codes and reason codes are generated by the SYMREC macro processing routine. The return code is returned in register 15. The reason code is returned in register 0.

Return Code	Reason Code	Meaning
12	108	The length of the ADSR section 2 is not long enough.
12	128	Portions of the symptom record were not in the key of a nonauthorized caller or a negative length was found in ADSR section 2.
12	134	The input symptom record address was not in the key of a nonauthorized caller.

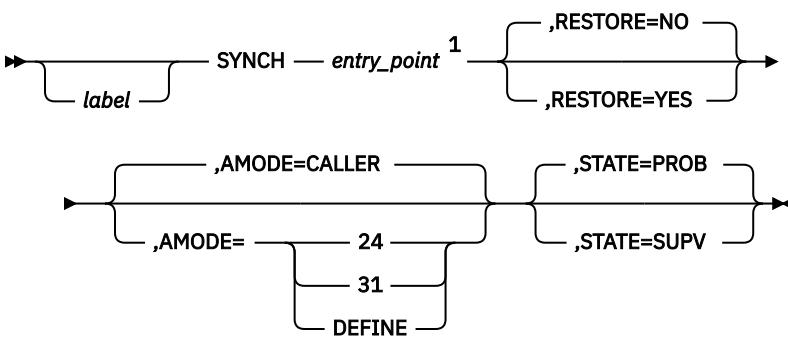
If none of the above errors occurred, the reason code and return code from the CP DIAGNOSE X'94' are placed in registers 15 and 0 respectively.

## SYNCH

The SYNCH macro is available in standard, list and execute formats.

### Standard Format

See also “List Format” on page 352 and “Execute Format” on page 353.



Notes:

<sup>1</sup> The following parameters are optional and may be specified in any order. For any parameter not specified, the default value, if applicable, is used.

### Purpose

Use the SYNCH macro to schedule a synchronous exit from one program to another, possibly with a change in state.

The SYNCH macro schedules a synchronous exit from one program to another. If desired, the SYNCH macro allows a supervisor state program to call another program and choose the state which the latter will operate. The SYNCH macro lets you control the restoration of registers belonging to the calling program.

On entry to the processing program, the high order bit, bit 0, of register 14 is set to indicate the addressing mode of the issuer of the SYNCH macro. If bit 0 is 0, the issuer is executing in 24-bit addressing mode. If bit 0 is 1, the issuer is executing in 31-bit addressing mode.

### Parameters

#### *entry\_point*

Specifies the address of the entry point that is to receive control.

The program must be resident in virtual storage.

You can write this parameter as an RX-type address or as a register. If you write it as a register, you can choose only from among registers (2) through (12) and register (15).

#### **RESTORE**

Indicates whether you want registers 2 through 13 restored when control is returned to the program that issued the SYNCH macro. If you do not specify this parameter, then, by default, no restoration takes place.

#### **YES**

Indicates that you do want this restoration to take place.

**NO**

Indicates that you do not want this restoration to take place.

**AMODE**

Specifies the addressing mode which the requested program is to receive control.

**24**

The requested program will receive control in 24-bit addressing mode.

**31**

The requested program will receive control in 31-bit addressing mode.

**DEFINED**

The requested program will receive control in the addressing mode indicated by the high order bit of the specified entry point address. If the bit is off, the requested program will receive control in 24-bit addressing mode. If the bit is set, the requested program will receive control in 31-bit addressing mode.

**Note:** The user must provide the entry point using a register and not an RX-type address.

**CALLER**

The requested program will receive control in the addressing mode of the caller.

**STATE**

Indicates the state which the program being called will function. If you do not specify this parameter, then the program being called functions in problem state, by default.

**SUPV**

Indicates that the program being called in will function the supervisor state.

**PROB**

Indicates that the program being called in will function the problem state.

**Usage**

1. The SYNCH macro makes no validity checks the entry point address. Regardless of what is at that address, control is transferred to it.
2. It is not necessary for the program that issues the SYNCH macro to be in supervisor state. Nor must a program called by a supervisor state program necessarily function in that state. The rule is if the program issuing the SYNCH instruction is a:
  - Problem state program, then the called program will also function in that state.
  - Supervisor state program, then there is a choice. Use the STATE parameter to specify which state the called program is to function.
3. It is important to remember that any program called through the SYNCH macro will always run in the same key as the program that called it. This usually is not a problem. However, a supervisor state program may call another program and specify that the latter should run in problem state. The supervisor state program should change its own key to the problem state program before it issues the SYNCH macro.
4. It is risky to use the SYNCH macro to transfer control to a program that is not reentrant. While this practice is not prohibited, the results are unpredictable.

**Examples**

```
SYNCH (2), RESTORE=NO, STATE=SUPV
```

Schedule an exit to the entry point whose address is in register 2. The program called will function in supervisor state if the program issuing the SYNCH macro is also in supervisor state. When control is returned to the program that issued the SYNCH macro, no registers will be restored.

```
SYNCHIT SYNCH ENCRYPT, RESTORE=YES, STATE=PROB
```

Schedule an exit to an address named ENCRYPT. ENCRYPT is to function in problem state. SYNCHIT is the label on this instruction. When control is returned, registers 2 through 13, belonging to the program that issued the SYNCH instruction, will be restored.

The exit program receives the following information in its registers.

Register	Contents
0-13	Unchanged.
14	The address to which control is to return after the exit program completes execution.
15	The address of the entry point in the exit program being called.

```
SYNCH (8), RESTORE=YES, AMODE=24
```

Take a synchronous exit to the program located at the address in register 8 and restore registers 2-13 when control returns. Indicate that this program is to process in 24-bit addressing mode.

```
SYNCH (8), RESTORE=YES, AMODE=DEFINED
```

Take a synchronous exit to the program located at the address given in register 8 and restore registers 2-13 when control returns. Indicate that this program is to receive control in the addressing mode defined by the high-order bit of its entry point address (in register 8).

```
SYNCH (8), RESTORE=YES, AMODE=CALLER
```

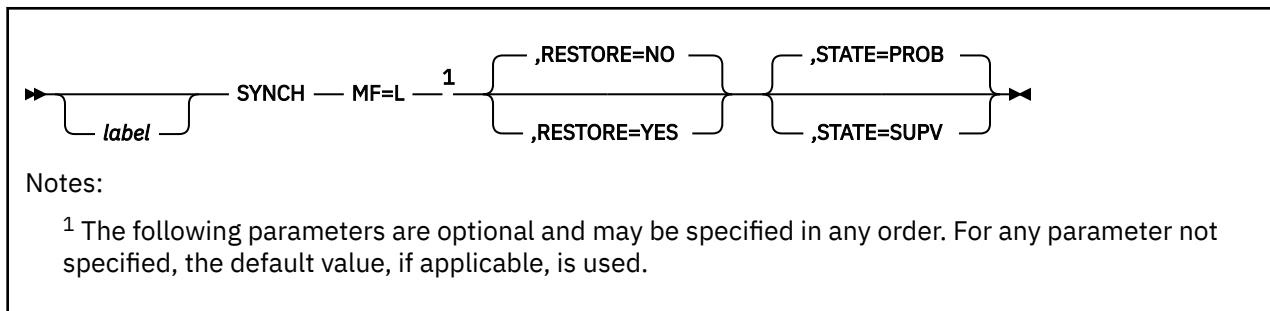
Take a synchronous exit to the program located at the address given in register 8 and restore registers 2-13 when control returns. Indicate that this program is to receive control in the addressing mode of the caller.

## Return Codes and ABEND Codes

The SYNCH macro generates no return codes.

ABEND Code	Reason Code	Meaning
106	OC	Insufficient virtual storage was available to load the requested module.
206		Either an invalid parameter list was produced or an I/O error occurred while processing.

## List Format



## Purpose (List Format)

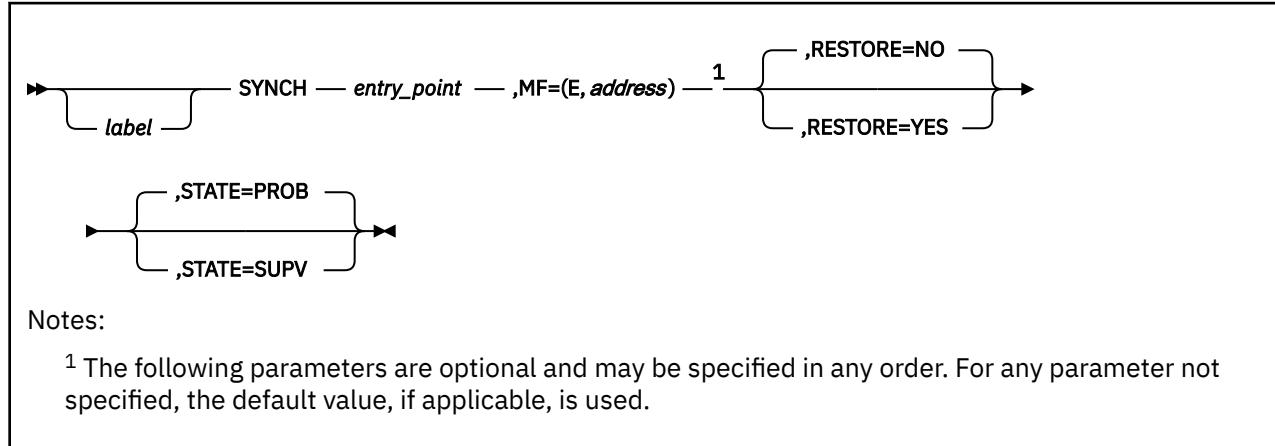
This format of the macro generates an in-line parameter list, based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation. Only the preceding parameters listed are valid in the list format of this macro.

## Added Parameter

### **MF=L**

Specifies the list format of this macro.

## Execute Format



## Purpose (Execute Format)

This format of the macro generates code that executes the function, using a parameter list whose address you specify.

## Added Parameter (Execute Format)

### **MF=(E, *address*)**

*address* specifies the address of the parameter list to be used by the macro.

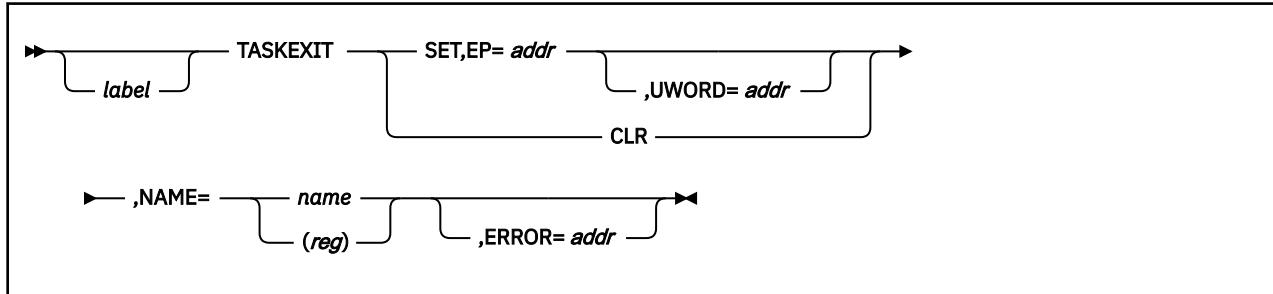
You can add or modify values in this parameter list by specifying them in this macro.

## TASKEXIT

The TASKEXIT macro is available in standard, list, list address and execute formats.

### Standard Format

See also [“List Format” on page 356](#), [“List Address Format” on page 357](#) and [“Execute Format” on page 357](#).



### Purpose

Use the TASKEXIT macro to declare a task termination exit routine for an entire virtual machine group.

A task termination exit routine, declared for an entire virtual machine group, gains control whenever a task running within the group terminates—either normally or abnormally.

There are several good reasons for declaring such an exit routine for a virtual machine group. For example, several subsystem applications may be running in various virtual machines within the group. Having a task termination exit routine declared might help the subsystem clean up after itself, monitor the various applications, and react to their progress.

The TASKEXIT macro is an authorized GCS function.

### Parameters

#### SET

Indicates that you are declaring a task termination exit routine for your entire virtual machine group.

#### CLR

Indicates that the task termination exit routine you specify is to be canceled.

#### EP

Specifies the address of the task termination exit routine in question.

This exit routine must reside in a shared segment. That is, a routine whose entry point is defined in a shared segment directory that was created through the CONTENTS macro. See [“CONTENTS” on page 197](#).

You can write this parameter as an assembler program label or as register (2) through (12).

#### UWORD

Specifies a fullword of data you want passed to the task termination exit routine if it ever gains control.

You can use this parameter to pass any information you please.

If you write this parameter as an assembler program label, then the address associated with that label is passed to the exit routine. If you write it as register (2) through (12), then the contents of the register are passed to the routine.

**NAME**

Specifies a name used in any TASKEXIT macro to refer to a certain task termination exit routine.

Do not confuse this name with the name of any entry point within the exit routine or with the name of the routine itself. This name is merely an identifier used by the TASKEXIT macro to distinguish one task termination exit routine from another. The name is meaningless outside the TASKEXIT macro environment.

This name can contain up to eight characters.

There are two ways of coding this name in the TASKEXIT macro:

- Write the actual name itself.
- Write a register number from (2) through (12). The register you specify must contain the address where the name can be found. If the name is less than eight characters long, then it must be padded on the right with blanks.

**ERROR**

Specifies the address of an error routine that is to receive control if an error is found in the TASKEXIT macro.

If you omit this parameter and an error occurs, then control will return to the instruction following the TASKEXIT macro, just as it would were there no error.

**Usage**

1. Only an authorized user can enter the TASKEXIT macro.
2. The exit routine that you define through the TASKEXIT macro must reside in a shared segment.
3. Remember that the identifier you specify in the NAME parameter is strictly for your benefit and the TASKEXIT macro. To specify the SET and NAME parameters together states the name that is associated with the exit routine in question.
4. The AMODE, which the exit will be run, is the AMODE from the correspondent entry in the CONTENTS macro. However if the AMODE parameter in the CONTENTS entry is DEFINED, then the address of the routine in the TASKEXIT macro will be considered a 32 bit address with the AMODE being the first bit. See “[CONTENTS](#)” on page 197.
5. You can declare more than one task termination exit routine for your virtual machine group. However, because the TASKEXIT macro can declare only one exit routine at a time, you will have to enter it more than once. Each exit routine that you declare will run when a task in your virtual machine group terminates. However, the order which they will run is unpredictable.
6. GCS associates the PSW key and the enable flags of the task that issues the TASKEXIT macro with those of the task termination exit routine.
7. A task termination exit routine always runs in supervisor state. Moreover, it is eligible for the same types of interrupts as the task that declared it.
8. Remember that besides the task termination exit routine declared for the entire group, an individual task may have its own exit routines declared. For example, you may have defined an exit routine through the ESTAE macro that will run if the task ends abnormally.  
Should this be the case, and the task ends, GCS sees to it that the task's exit routines are run first. Afterward the task termination exit routine is executed.
9. When the task that declared the task termination exit routine terminates, then the latter executes one last time. After that, it disappears.
10. When the task termination exit routine gains control, its registers contain the following:

<b>Register</b>	<b>Contents</b>
0	The high-order 2 bytes contain the virtual machine ID in which the terminated task was running. The low-order bytes contain the task ID.
1	The UWORLD.

Register	Contents
13	Address of the register save area.
14	Return address within the GCS supervisor.
15	The address of the task termination exit routine.

## Examples

```
DCLTE TASKEXIT SET,EP=(4),NAME=TE6,ERROR=(7)
```

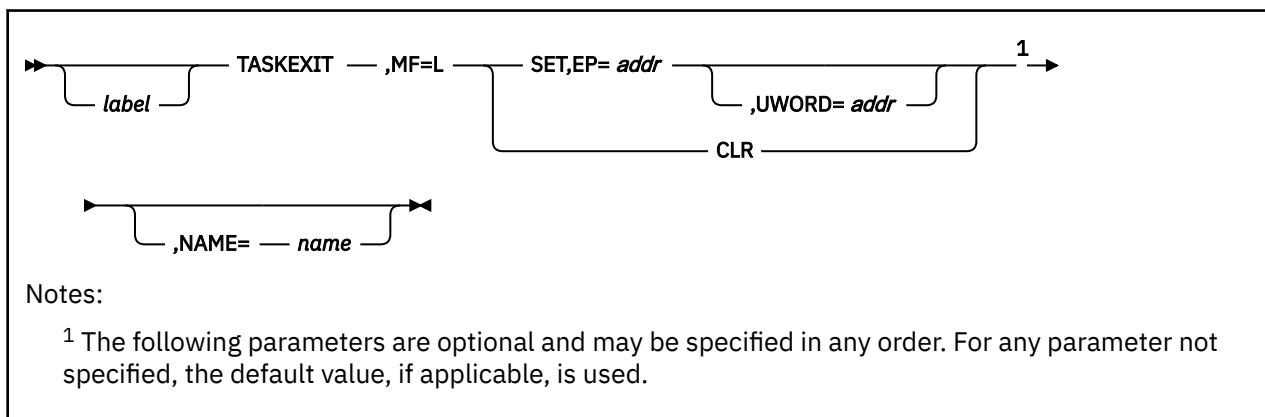
An authorized member of a virtual machine group wants to define a task termination exit routine for its entire group. The entry point of this routine is at the address in register 4. Because this routine is being newly defined, the authorized member declares the name TE6 for the routine. The address of the error routine is in register 7. DCLTE is the label on this instruction.

## Return Codes and ABEND Codes

When this macro completes processing, it passes to the caller a return code in register 15.

Hex Code	Decimal Code	Meaning
X'00'	0	Function completed successfully.
X'04'	4	This task termination exit routine has already been declared for this virtual machine group.
X'08'	8	The address you specified for the task termination exit routine is not in a shared segment.
X'18'	24	Invalid parameter list.
X'2C'	44	You specified the CLR parameter with the name of a task termination exit routine. However, no such name could be found for a task termination exit routine.
X'30'	48	The CONTENTS entry has AMODE=DEFINED or AMODE=CALLER, the caller is in AMODE 24 and the exit address is above the 16MB line.

## List Format



## Purpose (List Format)

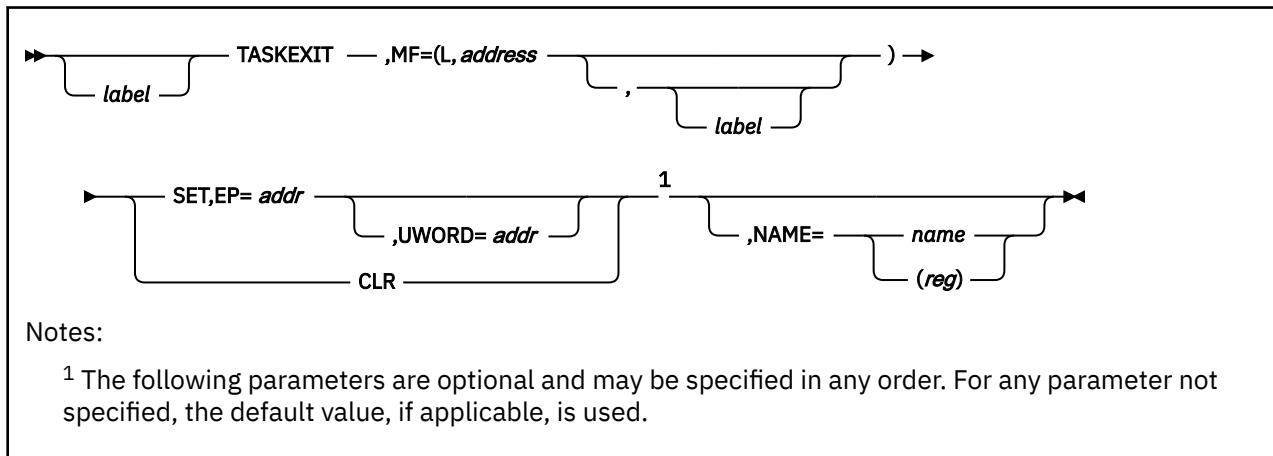
This format of the macro generates an in-line parameter list based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation. Only the preceding parameters listed are valid in the list format of this macro.

### Added Parameter

#### **MF=L**

Specifies the list format of this macro.

### List Address Format



## Purpose (List Address Format)

This format of the macro does not produce any executable code that starts the function. However, it does produce executable code that moves the parameter values that you specify into a certain parameter list. If you enter the macro using this format, then you must do so before any related invocation of the macro using the execute format. Only the preceding parameters listed are valid in the list address format of this macro.

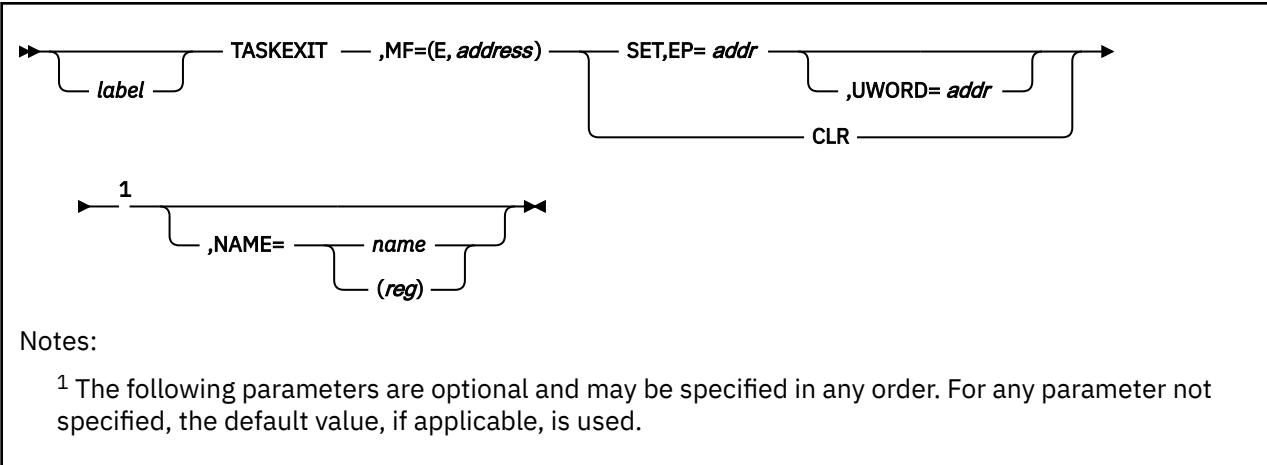
### Added Parameter (List Address Format)

#### **MF=(L, address, label)**

*address* specifies the address of the parameter list into which you want the parameter values you mention placed. This address can be within your program or somewhere in free storage.

*label* is optional and is a user-specified label, indicating that you want to determine the length of the parameter list. The macro expansion equates the label you specify with the length of the parameter list.

### Execute Format



### Purpose (Execute Format)

This format of the macro generates code that executes the function using a parameter list whose address you specify. Only the preceding parameters listed are valid in the execute format of this macro.

### Added Parameter (Execute Format)

#### **MF=(E, address)**

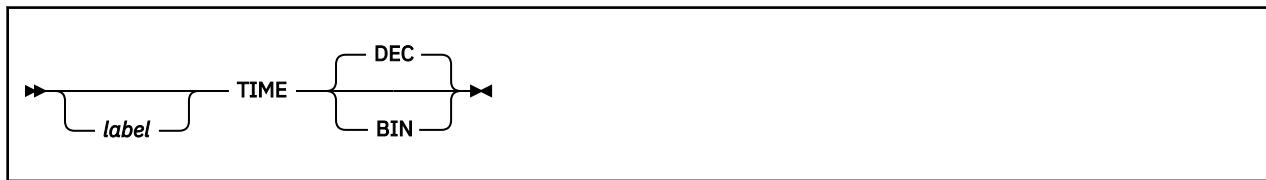
*address* specifies the address of the parameter list to be used by the macro.

You can add or modify values in this parameter list by specifying them in this macro.

# TIME

---

## Format



## Purpose

Use the TIME macro to ask the supervisor to send today's date and the correct time of day to your program.

## Parameters

### DEC

Indicates that the time of day is to be returned to your program in unsigned packed decimal format. It is stored in the following format:

#### **HHMMSS00**

**HH** stands for the number of hours.

**MM** for the number of minutes.

**SS00** for the number of seconds with 00 representing tenth and hundredths of a second.

The tenths and hundredths of seconds are always returned as 00.

Today's date is also returned to your program in packed decimal form.

If you omit all parameters from the TIME macro, then DEC is assumed, by default.

### BIN

Indicates that the time of day is to be returned to your program as an unsigned 32-bit binary number. The low-order bit is equivalent to 0.01 seconds. The tenths and hundredths of seconds are always returned as 00.

Today's date, however, will be returned to your program in packed decimal form.

## Usage

1. The time of day is returned to your program in register 0.
2. Today's date is returned to your program in register 1.
3. The date is stored in the following format:

#### **0CYYDDDF**

**0** is a half-byte of zeros. **C** is a century indicator, where a value of 0 (zero) indicates the 1900's, and a value of 1 indicates the 2000's. **YY** are the last two digits of the year. **DDD** is the Julian day of the year. **F** is a 4 bit sign character that helps you unpack and print the date, if you request it.

4. The accuracy of the time and date depends on the accuracy of the corresponding data entered by your system operator. Your system's response time is also a factor.

## Return Codes and ABEND Codes

The TIME macro generates no return codes.

ABEND Code	Meaning
E0B	A parameter not supported by GCS was specified. Unsupported parameters include TU, MIC, STCK, and ZONE=GMT.

# TTIMER

---

## Format



## Purpose

Use the TTIMER macro to cancel a timer that you set through a STIMER macro. See “[STIMER](#)” on page [345](#).

## Parameters

### CANCEL

Indicates that you wish to cancel the effect of the last STIMER macro. That is, the timer stops keeping track of elapsed time. Also, the specified branch to an exit routine, if any, is canceled.

This is the only parameter on the TTIMER macro and is required.

## Usage

The TTIMER macro has no effect if the STIMER macro you are trying to cancel included the WAIT parameter.

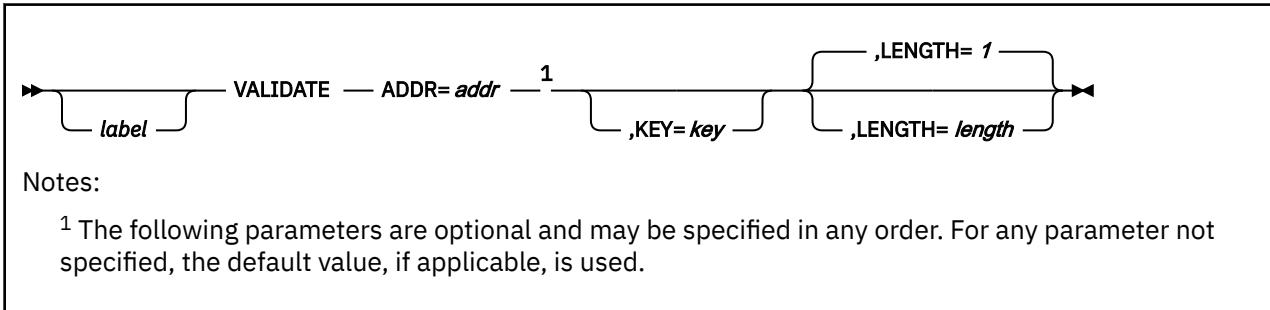
## Return Codes and ABEND Codes

The TTIMER macro generates no return codes.

ABEND Code	Meaning
E2E	Either the CANCEL parameter was not specified or a parameter not supported by GCS was specified. Unsupported parameters include TU and MIC.

# VALIDATE

## Format



## Purpose

Use the VALIDATE macro to compare keys, confirm that a virtual machine, program, and so forth, has access to a particular area of storage.

Virtual machines, tasks, and programs constantly request access to areas of storage. This does not necessarily mean that they are entitled to have each request granted. Each 4KB block of storage has a key associated with it. This key governs access to the storage block and protects the data there against unauthorized use.

There are two kinds of access available: fetch and store. If, for example, a program has fetch access, it means that it can only obtain data from the block. Fetch access prevents the program from actually changing any of the data in the block. Store access, on the other hand, allows a program to both obtain data from the storage block and alter the data therein. Also there are programs that are denied either type of access.

The VALIDATE macro confirms or denies that a program has access to a certain block of storage. If access is allowed, it indicates whether the program can have fetch-type access or store-type access.

The VALIDATE macro is an authorized GCS instruction.

## Parameters

### ADDR

Specifies the starting address of the area of storage to which the program wants access.

You can write this parameter as an assembler program label or as register (1) through (12). If you write it as a label, then the address of the label must be the starting address of the storage area in question. If you write it as a register, then the register must contain this starting address.

### KEY

Specifies the key that will be compared with the key of the storage area in question.

You can write this parameter as an assembler program label, as register (0), or as register (2) through (12). If you write it as a label, then the key must be contained in the 4 high-order bits of the byte at the address associated with that label. If you write it as a register, then the key must be in bits 24 through 27 of that register. If you do not specify a key, then VALIDATE will use the key of the task that issued the instruction.

### LENGTH

The length of the storage area in question, in bytes.

If you omit this parameter, then the length is 1, by default.

You can write this parameter as an absolute expression, as register (2) through (12), or as register (15). If you write the length as an absolute expression, then it must be a positive integer between 1 and  $2^{24}-1$ . If you write it as a register, then the register must contain a positive fullword integer within the same range.

## Usage

1. The VALIDATE macro does not obtain access for any program. It only tells whether a program is entitled to access a certain area of storage and, if so, in what way it can access the storage.
2. The supervisor determines whether the area of storage in question is addressable. If it is, then the key specified in the VALIDATE macro is compared with the key of the area of storage in question. If they do not match, the supervisor checks to see if the area of storage is fetch protected. The appropriate return code is then passed to the issuer of the macro.
3. If the key of the storage area matches the key specified in the VALIDATE macro, or if the program is running in key 0, then store access to the area is possible.
4. If the keys do not match, the program is running in a key other than 0, and the storage area is without fetch protection, then fetch access to the area is possible.
5. If the keys do not match, the program is running in a key other than 0, and the storage area has fetch protection, then no access to the area is possible.
6. Authorized programs often are asked to perform work for unauthorized programs. Before an authorized program accesses an area of storage for an unauthorized program, it should confirm that the latter is *sufficiently authorized* to have its work affect that storage. This is one of the major applications of the VALIDATE macro. In addition, system routines frequently use the VALIDATE macro to accomplish much the same thing.
7. Before an authorized program issues the VALIDATE macro, it should place a lock on the storage in question through the LOCKWD macro. This is required to prevent the key of the storage from changing. See “[LOCKWD](#)” on page 302.

## Examples

```
VALIDATE ADDR=ADDRESS,KEY=KEY1
.
.
ADDRESS DS F'5672'
KEY1  DS X'E0'
```

Confirm that the address is accessible by a program running in key 14.

```
VALIDATE ADDR=(6),KEY=(7),LENGTH=(3)
```

Confirm that the program running in the key stored in register 7 has access to the storage area beginning at the address in register 6. The length of the storage area in question is in register 3.

## Return Codes and ABEND Codes

The VALIDATE macro generates no ABEND codes.

When this macro completes processing, it passes to the caller a return code in register 15.

---

Hex Code	Decimal Code	Meaning
X'00'	0	The key of the storage area matches the key specified in the macro or, if none was specified, the key of the program that issued the macro.

---

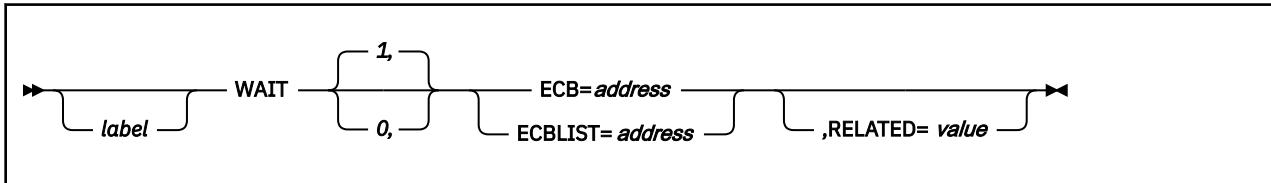
## VALIDATE

Hex Code	Decimal Code	Meaning
X'04'	4	The keys do not match but the storage area has no fetch protection. Therefore, fetch access is possible.
X'08'	8	The keys do not match and the storage area has fetch protection. Therefore, no type of access is possible.
X'0C'	12	The storage area in question is not addressable.
X'10'	16	The specified length of the storage area is less than 0 or greater than $2^{24}-1$ bytes for 370 accommodation.

# WAIT

---

## Format



## Purpose

Use the WAIT macro to wait for an event to take place before continuing processing.

Each such event is associated with an event control block (ECB). This ECB defines the event that is to occur and indicates to your task whether it has occurred. For example, your task may be unable to continue until it receives input from a certain file.

Use the WAIT macro to cause your task to wait for a certain event to take place before your task resumes processing.

## Parameters

### **1, 0,**

These are number of events that must take place before your task can resume.

You are limited to specifying either zero events or one event, written as the numerals 0 or 1.

If you omit this parameter, one event is assumed, by default. If you write 0, then the macro is treated as a NOP (NO OPERATION) assembler instruction.

### **ECB**

Specifies the address of a single ECB associated with the event for which your task must wait.

You can write this parameter as an RX-type address or as register (1) through (12).

### **ECBLIST**

Specifies the address of an area in your virtual storage that contains a string of addresses. Each address in the string points to one ECB, and there may be one or more addresses in this string.

This list of ECB addresses signifies a list of events. If one of these events occurs, then your waiting task will be able to continue. This string must begin on a fullword boundary, as must each address in the string. The high-order bit of the last address in the list must be set to 1, indicating the end of the list.

You can write the address of this string as an RX-type address or as register (1) through (12).

### **RELATED**

Specifies documentation data that you are using to relate this macro to a POST macro.

The value you assign to this parameter has nothing to do with the execution of the macro itself. It merely relates one macro (WAIT) to a macro that provides an opposite, though related, service (POST).

The format and contents of this parameter are at your discretion and can be any valid coding values.

## Usage

1. The task issuing the WAIT macro provides storage for each event control block. Each ECB is a fullword on a fullword boundary.  
Bit 0 of the ECB is called the WAIT bit. If this bit is set to 1, then it means that some task is waiting for the event associated with that ECB to occur.  
Bit 1 of the ECB is called the POST bit. If this bit is set to 1, then it means the event associated with the ECB has occurred. See “POST” on page 314.
2. If the program issuing the related POST macro has chosen to pass it, then the remaining 30 bits of the ECB will contain a completion code. This code will describe the manner that the event your program has waited for took place.  
This completion code only has meaning to the applications involved.
3. You know that the event in question has occurred when your task regains control.
4. The occurrence of any one of the events associated with the ECBs in the list will allow your task to continue.
5. Tasks are not always placed in the WAIT state after having issued the WAIT macro. The task is immediately satisfied.
6. Reset to zero each bit of the ECBs in question before you enter the WAIT macro. After your program regains control, reset these bits after the ECB is analyzed. If you do not, and the event occurs again, your program will not know it.
7. No task should change any of the bits in any ECB for which a WAIT macro has been issued. Only after the POST bit has been set to 1 and its contents analyzed is it safe to alter an ECB.
8. If you choose to branch directly to the WAIT service routine, then your registers must contain the following:

Register	Contents
0	The number of events (zero or one) that must occur before your task resumes.
1	The address of the event control block (ECB). If this is the address of a list of ECBs, then the list address must be in two's complement form, and the high-order bit of the last address in the list must be 1.
13	The address of a register save area.
14	The caller's return address.
15	The address of the entry point of the WAIT macro is obtained from the CVT (CVTVWAIT).

9. Your task must be in supervisor state, key 0, and disabled for interrupts.
10. An interrupt handler cannot use the branch interface to the WAIT service routine.
11. Because branching directly to the service routine avoids the supervisor call, no trace entry for the function is generated.

## Examples

```
HOLDIT WAIT 1,ECB=(2)
```

The task will wait for one event to occur. That event is associated with an ECB whose address is in register 2. The task will regain control when the POST bit is set to 1. HOLDIT is the label on this macro.

```
WAIT ECBLIST=(4)
```

The task will wait for one of several events to occur. The ECBs associated with each of these events can be found in a list whose starting address is in register 4.

## Return Codes and ABEND Codes

When WAIT completes processing, it passes to the caller a ABEND code in register 15 on an SVC call.

ABEND Code	Meaning
0F8	The GCS supervisor was called in access register.
101	The problem program specified several events other than 0 or 1.
201	The macro expansion contained an invalid ECB address or the end of the ECBLIST could not be found.
301	The ECB's WAIT bit is already set to 1.

When WAIT completes processing, it passes to the caller a return code in register 15 on a branch entry.

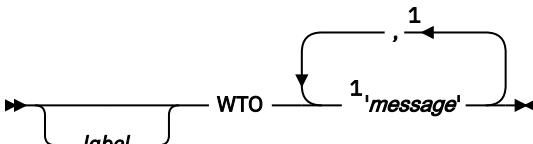
Hex Code	Decimal Code	Meaning
X'00'	0	The function completed successfully.
X'02'	2	The number of events was not zero or one.
X'04'	4	The ECB address was 0.
X'08'	8	The ECB wait flag was already on.

## WTO

The WTO macro is available in standard, list and execute formats.

### Standard Format

See also [“List Format” on page 368](#) and [“Execute Format” on page 369](#).



Notes:

<sup>1</sup> The message parameters can be repeated up to 10 times.

### Purpose

Use the WTO macro to send a message to the virtual machine console, requiring no reply.

Occasionally you will find it necessary to have a program running under GCS send a message to the virtual machine console. Use the WTO macro for this purpose. Use of this macro implies that you do not require a response to your message.

### Parameters

#### **'message'**

Specifies the text of the message to be sent to the virtual machine console.

Though they will not appear at the console, you must enclose the message in single quotation marks. Each message is displayed on a separate line and may be up to 124 characters long. If you send a message that is longer than that, it will be truncated before it is sent. You can include in your message any character that is permitted in a C-type (character) DC assembler instruction. Also, you can have up to 10 messages on a single invocation.

### Usage

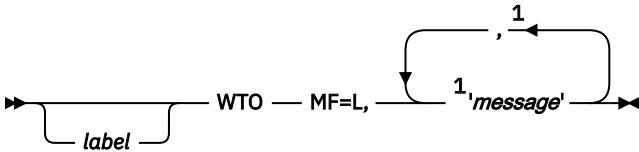
1. GCS supports multiple console message handling.
2. GCS performs no translation on your message at all. It is transmitted exactly as coded.

### Return Codes and ABEND Codes

The WTO macro generates no return codes.

ABEND Code	Meaning
D23	Either an invalid parameter list exists or insufficient space is available for processing.

### List Format



Notes:

<sup>1</sup> The message parameters can be repeated up to 10 times.

## Purpose (List Format)

This format of the macro generates an in-line parameter list, based on the parameter values that you specify. However, this format generates no executable code.

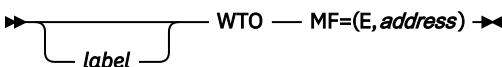
If a label is specified, it can be used to reference the start of the in-line parameter list.

## Added Parameter

### MF=L

Specifies the list format of this macro.

## Execute Format



## Purpose (Execute Format)

This format of the macro generates code that executes the function using a parameter list whose address you specify. Only the preceding parameters listed are valid in the execute format of this instruction.

## Added Parameter (Execute Format)

### MF=(E, address)

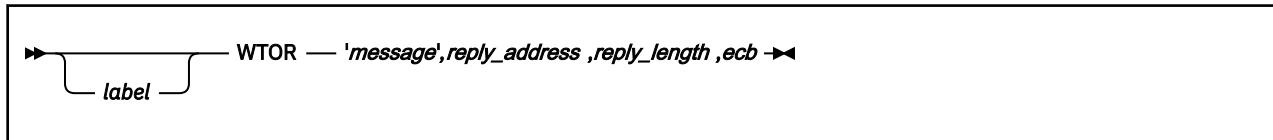
*address* specifies the address of the parameter list to be used by the macro.

## WTOR

The WTOR macro is available in standard, list and execute formats.

### Standard Format

See also [“List Format” on page 371](#) and [“Execute Format” on page 371](#).



### Purpose

Use the WTOR macro to send a message to the virtual machine console, requiring a reply.

Occasionally you will find it necessary to have a program running under GCS send a message to the virtual machine console. Moreover, your program may require a reply to its message.

Use the WTOR macro to send a message to the virtual machine console, to which you expect a reply.

The WTOR macro is available in standard, list, and execute formats.

### Parameters

#### **'message'**

Specifies the text of the message to be sent to the virtual machine console.

Though they will not appear at the console, you must enclose the message in single quotation marks. The message may be up to 121 characters long. If you send a message that is longer than that, it will be truncated before it is sent. Because the message is assembled as a variable-length record, it is not necessary to pad it with blanks. You can include in your message any character that is permitted in a C-type (character) DC assembler instruction.

#### **reply\_address**

Specifies the address in virtual storage into which you want the reply placed.

The reply will be left-aligned at this address.

You can write this parameter as an assembler program label or as register (2) through (12).

#### **reply\_length**

Specifies the maximum length of the reply that your program will accept.

This refers to the size of the reply area, the address of which you specified in the REPLY ADDRESS parameter.

This length must be from 1 to 119 bytes.

You can write this parameter as a symbol, as decimal digits, or as register (2) through (12).

#### **ecb**

Specifies the address of your event control block.

GCS uses this area of storage to indicate whether the reply to your message has been received. Event control blocks are discussed in detail under [“WAIT” on page 365](#) and [“POST” on page 314](#).

You can write this parameter as an assembler program label or as register (2) through (12).

## Usage

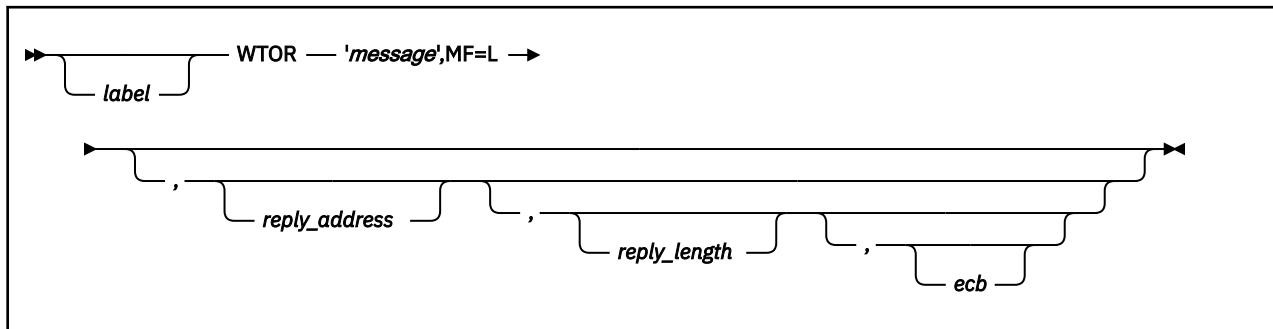
1. The WTOR macro assigns a reply identification number to the message it is transmitting for you. The operator will use this identification number when responding to your message.
2. GCS does not support multiple line messages or multiple console message handling.
3. GCS performs no translation on your message at all. It is transmitted exactly as coded.
4. The SPLEVEL macro need not be issued unless you want a WTOR macro used by GCS that has an expanded parameter list, which is designed for use in the 31-bit addressing mode. A 31-bit parameter list is incompatible if you are running under the 370 Accommodation Facility. However the SPLEVEL macro lets you select either the 24-bit version or the 31-bit version
5. This macro supports both 24 and 31 bit address expansions of the parameter list. The macro expansion is controlled by the internal macro SPLEVEL. The default value is 31.

## Return Codes and ABEND Codes

The WTOR macro generates no return codes.

ABEND Code	Meaning
D23	Either an invalid parameter list exists or insufficient space is available for processing.
E23	The address of the event control block or the address of the reply area was invalid.

## List Format



## Purpose (List Format)

This format of the macro generates an in-line parameter list, based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation.

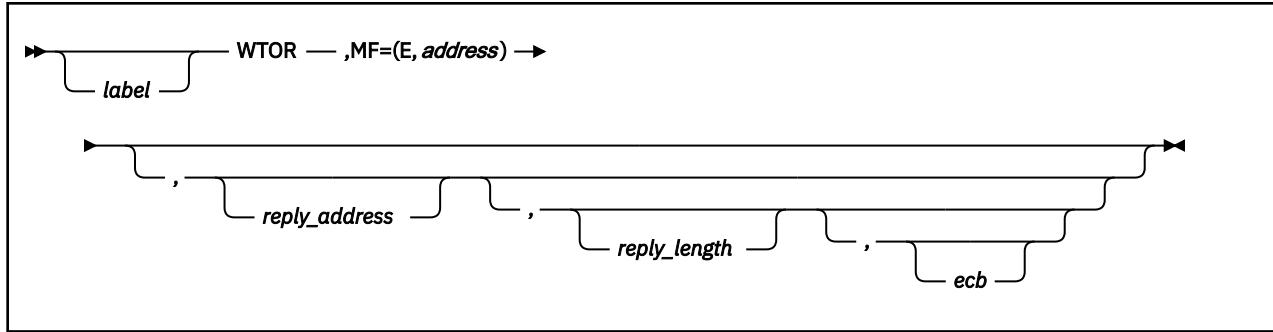
If a label is specified, it can be used to reference the start of the in-line parameter list.

## Added Parameter

### MF=L

Specifies the list format of this macro.

## Execute Format



### Purpose (Execute Format)

This format of the macro generates code that executes the function using a parameter list you specify. Only the preceding parameters listed are valid in the execute format of this instruction. The comma before the first operand is required to indicate the absence of the message operand which is not allowed in the execute format.

### Added Parameter (Execute Format)

#### MF=(E, address)

ADDRESS specifies the address of the parameter list to be used by the macro.

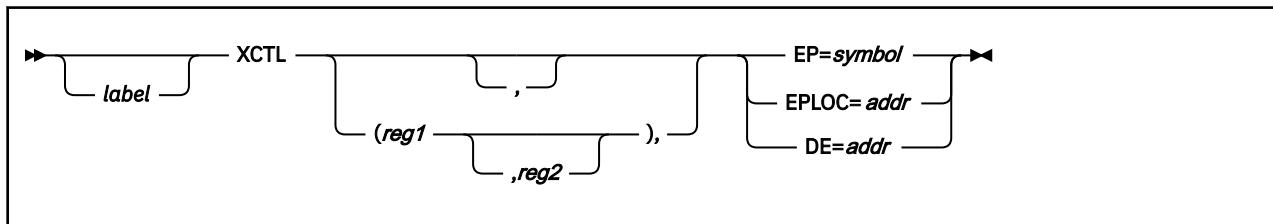
You can add or modify values in this parameter list by specifying them in this macro.

# XCTL

The XCTL macro is available in standard, list and execute formats.

## Standard Format

See also “List Format” on page 375 and “Execute Format” on page 376.



## Purpose

Use the XCTL macro to pass control to a program, expecting never to regain it.

GCS provides several techniques for passing control from one program to another. Typically, when one program passes control to another, it expects to eventually regain it. The XCTL macro lets you pass control from one program to another. If running in XA mode, XCTL handles the setting of the addressing mode when passing control to the new entry point. When the new entry point receives control, the high-order bit, bit 0, of register 14 is set to indicate the addressing mode of the issuer of the XCTL macro. If the bit is off, the issuer was executing in 24-bit addressing mode. If the bit is on, the issuer was executing in 31-bit addressing mode.

## Parameters

### (reg1),

### (reg1,reg2)

Specifies the register, or range of registers, that was saved by the issuer of the XCTL macro and that is to be restored and passed to the program called. The saving of these registers, then, becomes the responsibility of the program called by the XCTL macro.

You can write these as registers 2 through 12 or as assembler program labels. If you omit the reg2 parameter, then the only register restored is the one represented by the reg1 parameter. It is possible, however, to restore a subrange of registers within the range 2 through 12. The low register in the range must be reg1 and reg2 must be the high register in the range. If you omit these parameters entirely, no registers are restored.

Supply a set of parentheses around this parameter. And, if you specify a pair of register numbers, separate them with a comma.

### EP

Specifies the name of the entry point that receives control.

The entry point name can be:

- The name of the entry point as previously defined through the IDENTIFY macro. See “[IDENTIFY](#) on page 270.”
- The name of the entry point declared in a shared segment directory through the CONTENTS macro. See “[CONTENTS](#)” on page 197.”
- A member name (or alias) in the directory of a load library.

When looking for the entry point name that you specify, GCS searches the following items in the following order:

1. Your private storage, because the module associated with the entry point name may already be loaded.
2. Any shared segment directories that may have been created through the CONTENTS macro.
3. The directories of any load libraries that may have been defined for your virtual machine through the GLOBAL LOADLIB command. For more information on the GLOBAL command, see [“GLOBAL” on page 101](#).

You must write this parameter as an assembler program label.

**EPLOC**

The address containing the name of the entry point of the program that is to receive control.

The name, as stored, can be up to 8 bytes long. If less than 8 bytes long, then the name must be padded on the right with blanks.

You can write this parameter as an assembler program label or as register (2) through (12).

**DE**

Specifies the address of the name field within the directory list entry for the entry point. You must have previously created this list entry for the entry point using the BLDL macro. See [“BLDL” on page 181](#).

You can write this parameter as an assembler program label or as register (2) through (12).

**Usage**

1. If you enter the XCTL macro and the load module in question is not resident in virtual storage, then GCS will load the module for you. Then, after the module is run, GCS removes it from storage. This is satisfactory if you intend to pass control to the module only once.

However, loading a module into virtual storage involves a good deal of overhead processing. If you intend to pass control to the module more than once, it is far more efficient to start the LOAD macro once yourself. This avoids all the overhead processing involved in having GCS repeatedly load the module for you. See [“LOAD” on page 298](#).

2. It is the responsibility of the program issuing the XCTL macro to restore registers 2 through 14 to what they were when it first received control. Registers 13 and 14 must be restored before the XCTL macro is issued. Registers 2 through 12 (or a subset thereof) can be restored at the same time or through the (reg1,reg2) parameter.

The program issuing the XCTL macro can omit the (reg1) or (reg1,reg2) parameters. If it does, then the XCTL macro will restore no registers. It then becomes the responsibility of the program issuing the XCTL macro to restore registers 2 through 14 by itself.

3. It is the responsibility of the program receiving control through the XCTL macro to save the registers that the program that called it was saving.
4. The program called, using the standard format of the XCTL macro, may expect certain parameters passed to it. Because this program is using the standard format of the macro, it must see to it that register 1 contains the address of the parameter list, if one is expected.
5. You can use the XCTL macro to pass control to a serially reusable program. If the program is under the control of another user, then you will be placed in the WAIT state until the other user is finished.
6. If the program called is reentrant, then it is loaded into key 0 storage. This ensures that it is not accidentally modified or tampered with.
7. When control is passed to the program called, the registers contain the following information.

Register	Contents
0-13	Unchanged. Register 1 contains the address of the parameter list, if it was specified.

Register	Contents
14	The address to which control is to return after the called program completes execution.
15	The address of the entry point in the program called.

## Examples

```
XCTL (2,12),EP=PROGRAMC
```

The XCTL macro will first restore registers 2 through 12, which the program issuing it was saving. GCS assumes that this program restored registers 13 and 14 on its own. Then, control will pass to a program named PROGRAMC.

```
TRANSCTL XCTL (4),EPLOC=(6)
```

Pass control to an entry point whose name can be found at the address in register 6. The XCTL macro need only restore register 4. GCS assumes that the program issuing the XCTL macro restored registers 2, 3, and 5 through 14. TRANSCTL is the label on this macro.

```
XCTL DE=BLDLNAM
```

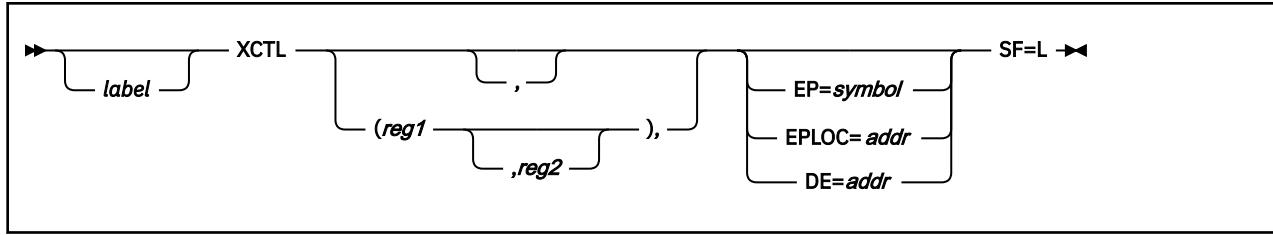
Pass control to a certain entry point. The system looks for the name of this entry point in the list entry created for that entry point. The name field of the list entry corresponds with the address of the label BLDLNAM. GCS assumes that the registers were all restored by the program issuing the XCTL macro.

## Return Codes and ABEND Codes

The XCTL macro generates no return codes.

ABEND Code	Reason Code	Meaning
106	0B	An error was found when the supervisor attempted to load the requested module into virtual storage.
106	0C	Insufficient virtual storage was available to load the requested module.
206		Invalid parameter list.
406		The module is marked ONLY LOADABLE.
706		The linkage editor marked the requested load module as NOT EXECUTABLE.
806	04	Either the program could not be found or no load libraries were defined by the GLOBAL LOADLIB command.
806	08	An irrecoverable I/O error occurred when the BLDL control program attempted to search the directory.
806	10	When GCS attempted to close the load library used by the BLDL macro, it found that the load library had never been opened.
906		The LOAD COUNT or the USE COUNT for the load module have reached the maximum of 32767.
A06		Your task is already waiting for this serially reusable module.

## List Format



## Purpose (List Format)

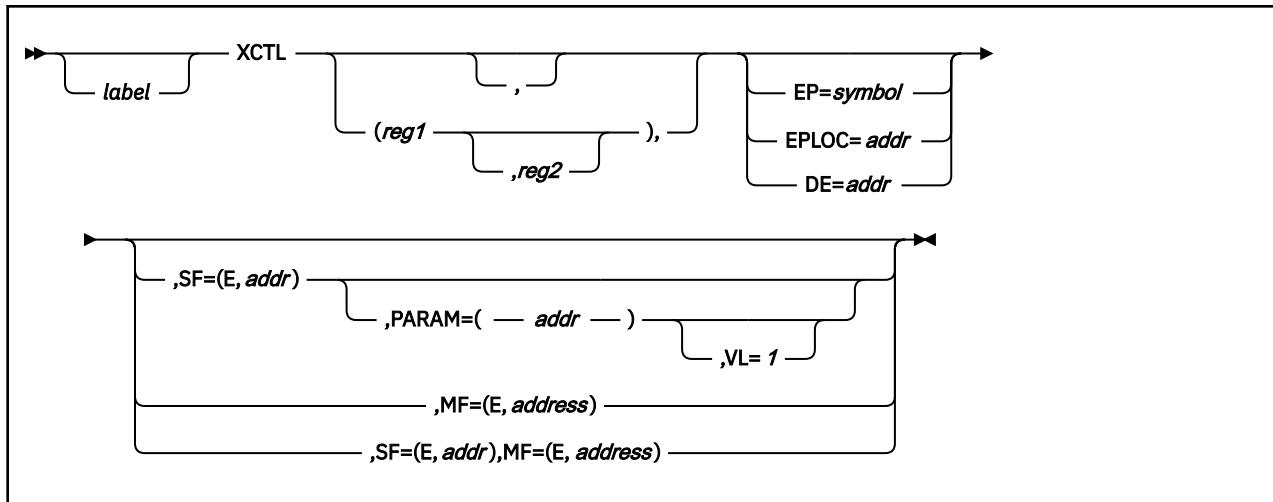
This format of the macro generates an in-line parameter list, based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation.

### Added Parameter

#### SF=L

Specifies the list format of this macro.

### Execute Format



## Purpose (Execute Format)

This format of the macro generates code that executes the function, using a parameter list whose address you specify. Only the preceding parameters listed are valid in the execute format of this macro.

### Added Parameter (Execute Format)

#### SF=(E, address)

ADDRESS specifies the address of the parameter list to be used by the macro. This is the parameter list that was generated through the list format of this macro.

You can add or modify values in this parameter list by specifying them in this macro.

#### MF=(E, address)

ADDRESS specifies the address of the remote parameter list to be used by the called program.

#### PARAM

Specifies one or more parameter addresses to be passed to the program being called. XCTL builds a parameter list containing these addresses in the order which you specify them. Then, the address of this parameter list is passed in register 1 to the program called.

You can write these parameters as assembler program labels or as registers (2) through (12).

**VL=1**

Indicates that the program called expects a variable number of parameters to be passed to it.

You must write this parameter exactly as shown and you can use it only with the PARAM parameter. To omit the VL=1 parameter is to say that the program called expects a set number of parameters.

**XCTL**

---

# Chapter 6. QSAM and BSAM Data Management Service Macros

The QSAM and BSAM data management service macros are presented in alphabetic order in the section. The GCS macros are described in [Chapter 5, “GCS Macros,” on page 157](#). The VSAM data management service macros are described in [Chapter 7, “VSAM Data Management Service Macros,” on page 417](#).

Any user applications using branch entries into QSAM and BSAM data management service macros must be in AMODE 24.

The QSAM and BSAM data management service macros are:

CHECK (BSAM)  
CLOSE (BSAM/QSAM)  
DCB (BSAM/QSAM)  
DCBD (BSAM/QSAM)  
GET (QSAM)  
NOTE (BSAM)  
OPEN (BSAM/QSAM)  
POINT (BSAM)  
PUT (QSAM)  
READ (BSAM)  
SYNADAF(BSAM/QSAM)  
SYNADRLS (BSAM/QSAM)  
WRITE (BSAM).

---

## Using QSAM and BSAM

Because QSAM and BSAM data management service is provided only below the 16MB line, all addresses provided through QSAM and BSAM data management service macros must adhere to all:

1. Branch entries to QSAM and BSAM support must be in 24-bit address mode (AMODE 24)
2. Calls are accepted in only the 24-bit address mode (AMODE 24)
3. Addresses are accepted in only the 24-bit address mode (AMODE 24)
4. Addresses must point to storage below the 16MB line
5. User exits must reside in virtual storage below the 16MB line.
6. Calls to any of these services cannot be made in AR mode.

## **CHECK (BSAM)**

### **Format**



### **Purpose**

Use the CHECK macro to test the completion of a READ or WRITE operation.

Whenever you enter a READ or WRITE macro, your task needs some way to confirm that the I/O operation completed successfully.

Use the CHECK macro immediately after each READ and WRITE macro to determine if and how the I/O operation was completed.

### **Parameters**

#### ***decb\_address***

Specifies the address of the data event control block (DECB) associated with the READ or WRITE macro you just issued.

The data event control block is created as part of the expansion of the READ or WRITE macro. It describes the input or output *event* that you have asked to take place. This control block is discussed in detail in the entries titled “[READ \(BSAM\)](#)” on page 406 and “[WRITE \(BSAM\)](#)” on page 413.

You can write this parameter as an RX-type address or as register (1) through (12).

### **Usage**

1. The CHECK macro tests for errors in the last READ or WRITE operation involving the specified DECB.

If you enter a READ macro and the END-OF-FILE condition has been raised, then the CHECK macro gives control to your end-of-file exit routine. This is the routine whose address you specified through the EODAD parameter of the DCB macro. (If necessary, review the entry “[DCB \(BSAM/QSAM\)](#)” on page 385.)

If you did not specify an end-of-file exit routine or an error occurred after you issued a WRITE macro, then GCS will give control to the error analysis routine that you specified through the SYNAD parameter in the DCB macro. If you failed to specify an error analysis routine, then your task will terminate abnormally.

2. For each READ or WRITE macro you enter you must also enter a CHECK macro. You must enter the CHECK macro immediately after the READ or WRITE macro with which it is associated. So, the sequence

```
READ...READ...WRITE...WRITE...CHECK...CHECK...CHECK...CHECK
```

is incorrect. But, the sequence

```
READ...CHECK...READ...CHECK...WRITE...CHECK...WRITE...CHECK
```

is correct.

3. GCS does not support the MVS parameter DSORG on this macro. If you include it, then an error will occur.

**Return Codes and ABEND Codes**

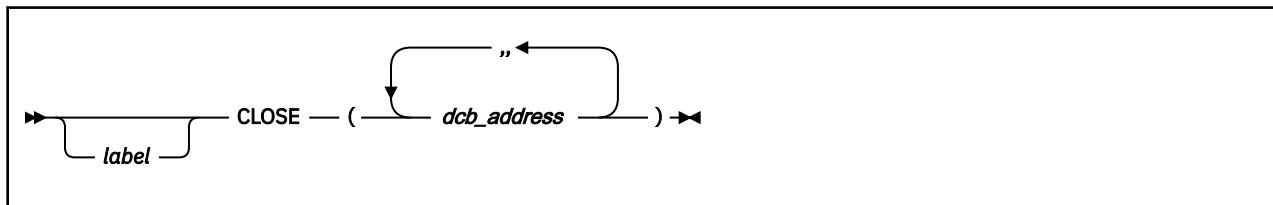
<b>ABEND Code</b>	<b>Meaning</b>
001	The data control block (DCB) of the file in question identified no SYNAD routine. Your task was terminated abnormally.
00A	An invalid address appeared in the CHECK macro, the data event control block (DECB), or the data control block (DCB).

## CLOSE (BSAM/QSAM)

The CLOSE macro is available in standard, list and execute formats.

### Standard Format

See also “List Format” on page 383 and “Execute Format” on page 383.



### Purpose

Use the CLOSE macro to close a file that your task had previously opened. After a task has finished with a particular file, the file must be closed.

### Parameters

#### **dcb\_address**

Specifies the address of the data control block associated with your file. For example, it is the address of the label on the DCB macro associated with your file. See “DCB (BSAM/QSAM)” on page 385.

More than one file can be closed by a single CLOSE macro. A double-comma is required to delimit each DCB address.

You can write this parameter as an RX-type address or as register (2) through (12).

### Usage

1. First, the CLOSE macro restores the data control block associated with your file to its original condition. That is, the original information you specified for the file in the DCB macro is restored.

The file is then *logically disconnected* from the main processor.

Finally, the I/O buffer that GCS set up for the file, when its DCB was opened, is released.

2. Only the task that opened the file can close it.

Often a file is being used by more than one task. If it is a BSAM file, then you must enter a CHECK macro for each data event control block (DECB) associated with the file before you close it. A DECB is associated with each of the file's I/O events. There may be several DECBs associated with output activity from several tasks. Therefore, you must make certain that all the tasks have completed their output to the file before you close it. The CHECK macro confirms whether there are any outstanding output events pending for the file in question, as from a WRITE macro. See “CHECK (BSAM)” on page 380 and “WRITE (BSAM)” on page 413.

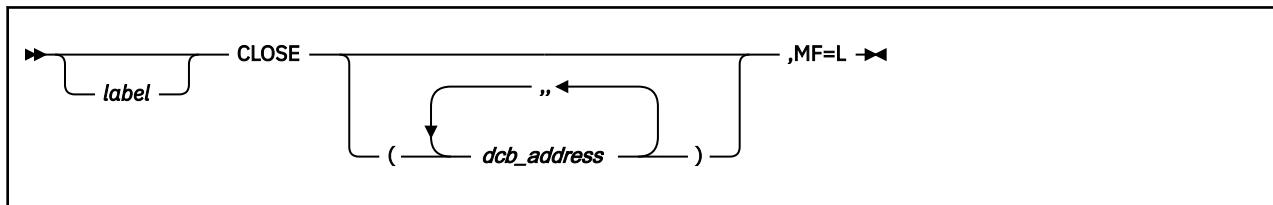
3. If you have access method control blocks (ACBs) that you wish to close, and DCBs, then you can specify a combination of both in the same CLOSE macro. GCS is able to distinguish the address of one from the address of the other, if you separate each with a double-comma.
4. The disk directory is not updated until the last file opened with the OPEN dcb address (OUTPUT) macro on the disk has been closed.

## Return Codes and ABEND Codes

The CLOSE macro generates no return codes.

ABEND Code	Meaning
014	An error occurred during the execution of the CLOSE macro. You will receive a message explaining this further.

## List Format



### Purpose (List Format)

This format of the macro generates a data management parameter list based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation.

The parameter list consists of a one-word entry for each DCB in the parameter list. The high-order byte is reserved while the 3 low-order bytes contain the address of a DCB. The end of the list is marked by setting the high-order bit of the last entry to 1.

The length of the list generated by the list format of this macro must be equal to the maximum length required by an execute format macro that refers to the same list. A maximum length list can be constructed in one of two ways.

1. Enter the macro using the list format with the maximum number of parameters required by the execute format of the macro that refers to the same list.
2. Use an appropriate number of commas in the list format of the macro to obtain a list of the required size. For example,

```
CLOSE (,,,...,,),MF=L
```

would create a list of five fullwords.

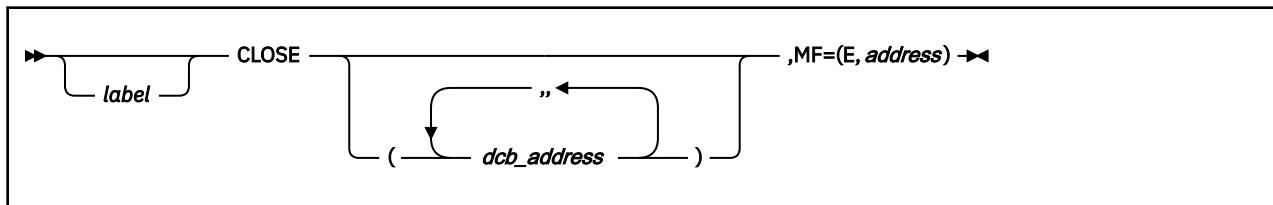
GCS assumes that any entries at the end of the list that are not referred to by the macro in the execute format were filled in by a previous macro.

## Added Parameter

### MF=L

Specifies the list format of this macro.

## Execute Format



### **Purpose (Execute Format)**

This format of the macro generates code that executes the function using a parameter list whose address you specify.

### **Added Parameter (Execute Format)**

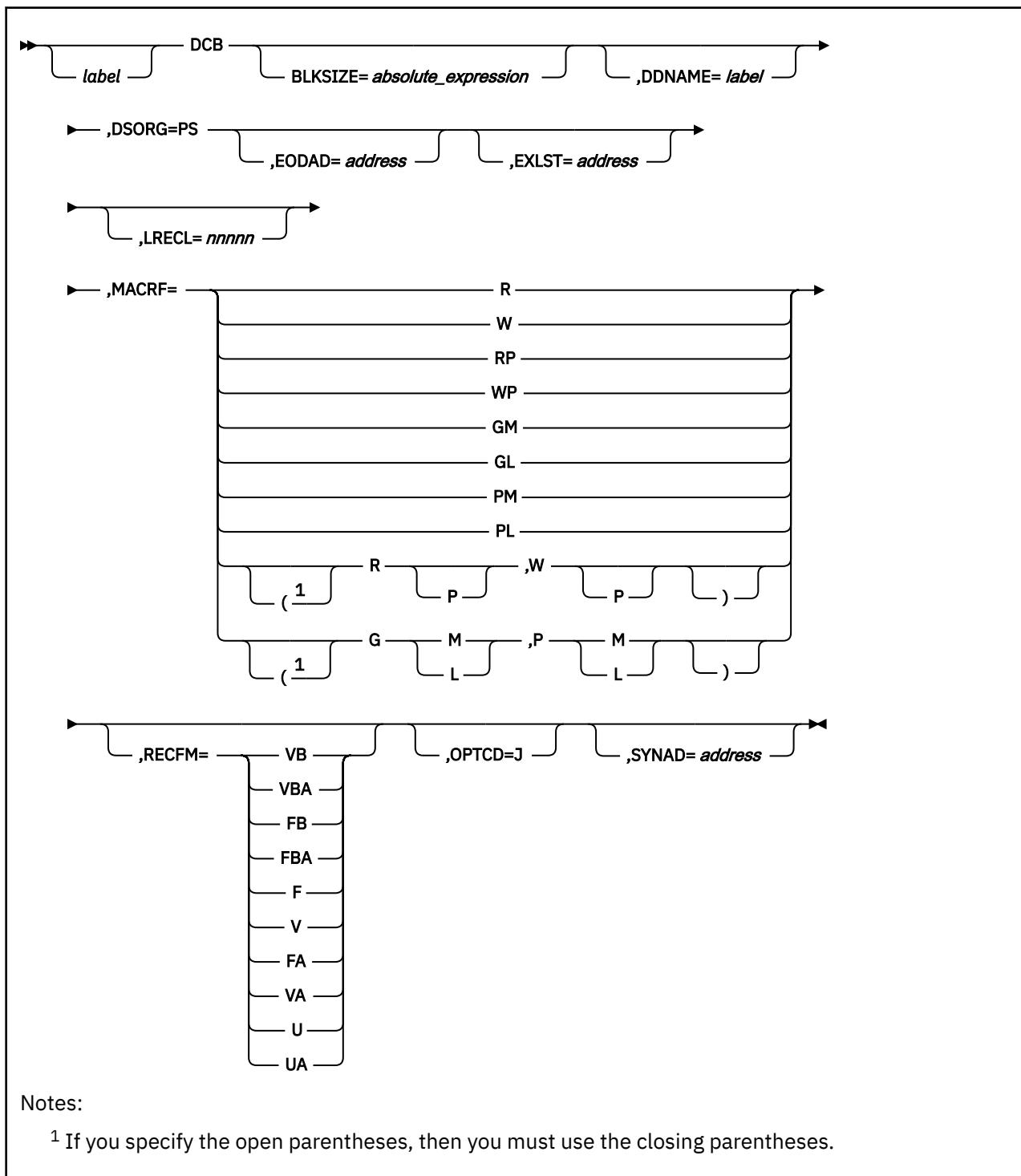
#### **MF=(E,*address*)**

ADDRESS specifies the address of the parameter list to be used by the macro.

You can add or modify values in this parameter list by specifying them in this macro.

## DCB (BSAM/QSAM)

### Format



### Purpose

Use the DCB macro to create a Data Control Block for one of your files.

For a program to process a file through BSAM or QSAM, a data control block (DCB) must be created for it. A DCB contains information that defines the characteristics of the data in the file and describes the I/O device requirements for handling the data.

Usually the DCB macro is issued sometime after the FILEDEF command is issued. The FILEDEF command provides similar information about your file. Together, the FILEDEF command and the DCB macro provide all the information necessary to the data control block. It is possible for the DCB macro to provide all data for the data control block without the help of the FILEDEF command. For more information on the FILEDEF command see “FILEDEF” on page 94.

## Parameters

It is required that the DSORG and MACRF parameters be specified in the DCB macro. The other parameters may be supplied through the:

- DCB macro
- FILEDEF command
- Physical characteristics of the file
- Direct insertion of a parameter's value or attribute into the data control block by your program. This is not too difficult, if you take advantage of the DCBD macro. See “DCBD (BSAM/QSAM)” on page 391.

However, you must be careful to insert the value in the DCB in a timely fashion. For example, it would be useless to insert the value of the DDNAME or the EXLST after issuing the OPEN macro, because that macro needs those values to process correctly.

### **BLKSIZE**

Specifies the maximum block length for the file, in bytes. For fixed-length, unblocked records, this parameter specifies the maximum individual record length.

If your file contains variable-length records, then the value specified by this parameter must include 4 extra bytes to accommodate the block descriptor word (BDW). In such a case, you can write this parameter as any number from 8 to 32756, plus 4 bytes for the BDW. (The OPEN macro simulation routine will not accept a BLKSIZE of less than eight.)

If your file contains undefined-length records, then the field in the DCB associated with this parameter (the DCBBLKSI field) can be filled in with the exact value after it is known by your program.

Alternatively, it can be specified in the LENGTH parameter of a READ or WRITE instruction. See “READ (BSAM)” on page 406 and “WRITE (BSAM)” on page 413.

### **DDNAME**

Specifies the name by which the file in question is known within your program. This parameter corresponds exactly with the DDNAME parameter in the FILEDEF command.

You can write this parameter as any label of from one to eight alphanumeric characters. The first character must be alphabetic or national.

### **DSORG=PS**

Indicates that your file consists of physical sequential records.

Because GCS supports only physical sequential file processing, this parameter is required.

### **EODAD**

Specifies the address of a routine that is to receive control when the end of an input file is reached.

It is your responsibility to provide this routine. Obviously you are only required to do so when the file, whose DCB you are creating, is an input file. You define whether it is an I/O file in the MACRF parameter as described in the following.

When GCS receives a request for input (for example, through a READ macro) and the next CHECK macro indicates that the end of the file has been reached, then this EODAD routine automatically receives control.

If this parameter is omitted and the END-OF-FILE condition is raised in an input file, then control is given to the routine whose address you specify in the SYNAD parameter, as described in the following.

If you omit both the EODAD and the SYNAD parameters, and the END-OF-FILE condition occurs, then your task terminates abnormally.

You can write this parameter as an RX-type address or as register (2) through (12).

#### **EXLST**

Specifies the address of your program's exit list.

This list contains the address(es) of one or more routines that you want executed during each OPEN macro that you request. See “[OPEN \(BSAM/QSAM\)](#)” on page 398.

If you specify this parameter, then it is the responsibility of your task to provide and maintain this exit list. Your task must provide the routines to which it refers. The list must begin on a fullword boundary, with each entry therein including a fullword. The basic format of the exit list is:

*Table 20. Exit List Format*

<b>1 Byte</b>	<b>3 Bytes</b>
Code	Routine 1's address
Code	Routine 2's address
...	...
Code	Routine n's address

The code in the first byte of each word indicates the disposition of the exit routine, whose address appears in the last 3 bytes. Note that these are the only codes that have meaning to GCS. Any others are ignored.

*Table 21. DCB Exit List Codes*

<b>Code</b>	<b>Meaning</b>
X'00'	INACTIVE routine that is not to be processed.
X'05'	ACTIVE routine that is to be processed.
X'80'	The last routine in the list. It is considered INACTIVE and is not processed.
X'85'	The last routine in the list. It is considered ACTIVE and is processed.

Just before the completion of each OPEN macro that you request, the exit list table is searched, and each active routine is processed.

You can write this parameter as an RX-type address or as register (2) through (12).

#### **LRECL**

Fixed-length record files, this parameter specifies the length, in bytes, of each record. You can write this as a number from 1 to 32760.

Variable-length record files, this parameter specifies the maximum length of any record in the file. You can write this as a number from 1 to 32752, plus 4 bytes for the record descriptor word (RDW).

It may happen that you omit this parameter in both the FILEDEF command and the DCB macro. If so, and if the file already exists, then the current LRECL value is obtained from the actual length of the file's records. However, if your file is newly created, then its logical record length must be supplied in one of the ways listed earlier. Otherwise it is considered an error.

#### **MACRF**

Specifies the type of macros that you will use to process the file in question. In effect, you use this parameter to define whether you will treat it as an input file or an output file. Also, you are stating what mode of data transmission you will use in moving data in to or out of the file.

#### **R**

(BSAM) Specifies that the READ macro will be used. See “[READ \(BSAM\)](#)” on page 406.

**W**

(BSAM) Specifies that the WRITE macro will be used. See “[WRITE \(BSAM\)](#)” on page 413.

**RP**

(BSAM) Specifies that the READ and POINT macros will be used. See “[POINT \(BSAM\)](#)” on page 402.

Specifying the RP parameter gives you the added capability of using the NOTE macro. See “[NOTE \(BSAM\)](#)” on page 396.

**WP**

(BSAM) Specifies that the WRITE and POINT macros will be used.

The WP parameter gives you the added capability of using the NOTE macro.

**GM**

(QSAM) Specifies that the GET macro in MOVE mode will be used. MOVE mode is defined in “[GET \(QSAM\)](#)” on page 394.

**GL**

(QSAM) Specifies that the GET macro in LOCATE mode will be used. LOCATE mode is defined in “[GET \(QSAM\)](#)” on page 394.

**PM**

(QSAM) Specifies that the PUT macro in MOVE mode will be used. MOVE mode is defined in “[PUT \(QSAM\)](#)” on page 404.

**PL**

(QSAM) Specifies that the PUT macro in LOCATE mode will be used. LOCATE mode is defined in “[PUT \(QSAM\)](#)” on page 404.

**RECFM**

Specifies the record format of your file.

For an existing file, the currently assigned record format is used unless another is specified. For a new file whose DCB you are creating, the record format is undefined, by default, unless one is specified.

Select from among the following record formats.

**VB**

Indicates that the records in your file are variable long, according to the LRECL parameter. It also indicates that these records are to be blocked according to the BLKSIZE parameter specified here or in the FILEDEF command.

**VBA**

Indicates the same as the VB parameter but also indicates that your file contains ANSI control characters.

**FB**

Indicates that each record in your file is of a fixed length, according to the LRECL parameter. Likewise, these records are to be blocked, according to the BLKSIZE parameter as specified here or in the FILEDEF command.

**FBA**

Indicates the same as the FB parameter but also indicates that your file contains ANSI control characters.

**F**

Indicates that each record in your file is of a fixed length, according to the LRECL parameter.

**V**

Indicates that the records in your file are variable long, according to the LRECL parameter.

**FA**

Indicates that your file is composed of fixed-length records that contain ANSI control characters.

**VA**

Indicates that your file is composed of variable-length records that contain ANSI control characters.

**U**

Indicates that the record format of your file is undefined. If the RECFM parameter is omitted, then the record format of the file is undefined, by default.

**UA**

Indicates that the record format of your file is undefined. It also indicates that your file contains ANSI control characters.

**OPTCD=J**

Indicates that the first byte in the output data stream will be a 3800 table reference character.

Such a character selects a particular character arrangement table for the printing of the output data stream on a 3800 printing subsystem. You can use this character with ANSI control characters, if you wish.

**SYNAD**

Specifies the address of your error routine that is to receive control when an irrecoverable I/O error occurs.

Under BSAM, this SYNAD routine receives control when the CHECK macro is issued. Under QSAM, it receives control automatically during the processing of the GET or PUT macro.

If you provide no error routine and an irrecoverable I/O error occurs, then your task terminates abnormally.

If you provide an error routine and an error occurs, then GCS automatically saves your program's registers and turns control over to your error routine. You must design your error routine in such a way that it does not use the register save area pointed to by register 13. This save area is for your program's registers. If your error routine needs a register save area, it must construct and maintain one of its own.

Your error routine can issue the RETURN macro, using the address in register 14, to return control to GCS. If control returns to GCS, then GCS returns control to the problem program, which can then proceed as though no error occurred. See “[RETURN](#)” on page 322.

You can write the SYNAD parameter as an RX-type address or as register (2) through (12). Remember, your program can change the address in this parameter anytime.

[Table 22 on page 389](#) shows the contents of the registers when your error routine receives control.

*Table 22. Register content when error routine receives control.*

Register	Bits	Meaning
0	0-7	Reserved.
0	8-31	For BSAM, the address of the event control block. For QSAM, these bits are all reset to 0.
1	0	The bit is set to 1 if the error was caused by an input operation.
1	1	The bit is set to 1 if the error was caused by an output operation.
1	2-7	Reserved.
1	8-31	The address of the data control block for the file in question.
2-13	0-31	The contents of the registers that existed before the macro was issued.
14	0-7	Reserved.
14	8-31	The address in the GCS supervisor to which control will return after your error routine completes processing.
15	0-7	Reserved.
15	8-31	The address of your error routine.

## Usage

1. The data control block for a BSAM or QSAM file is created during the assembly of the problem program. The data supplied by the FILEDEF command and the DCB macro are brought together at execution time to form one complete data control block. The physical characteristics on an existing disk file may also supply certain information. Among them, they can supply all necessary data for the DCB.

The FILEDEF command and the DCB macro may supply the value or attribute for the same parameter. If the value or attribute expressed by the FILEDEF command differs from that expressed by the DCB macro, then the latter will supersede the former.

2. Any READ or WRITE macro issued by your program must be tested for completion by the CHECK macro. See "[CHECK \(BSAM\)](#)" on page 380.
3. If you provide a list of exit routine addresses through the EXLST parameter, remember that your program can dynamically alter the disposition of each exit routine. Merely change the code in the first byte of the fullword containing the routine's address to indicate the desired disposition. Select from among the codes listed in [Table 21 on page 387](#).
4. Each of your exit routines must save the contents of register 14. The values in registers 2 through 13 are saved by the GCS supervisor.
5. Your SYNAD routine can end by:
  - Passing control to another routine in your program. For example, it could pass control to a program that closes the file being processed.
  - Returning control to GCS, which in turn would return control to your original program. Control would return to the instruction immediately following the one that caused the error.

If you choose the latter course, you must follow these conventions for saving and restoring registers:

- a. When it receives control from GCS, your SYNAD routine must not use the register save area pointed to by register 13. If necessary, use the SYNADAF macro to obtain the address of a register save area and message buffer that your SYNAD routine can use.

However, your SYNAD routine must release both this register save area and the message buffer, through the SYNADRLS instruction, when they are no longer needed. See "[SYNADRLS \(BSAM/QSAM\)](#)" on page 411 and "[SYNADAF \(BSAM/QSAM\)](#)" on page 409.

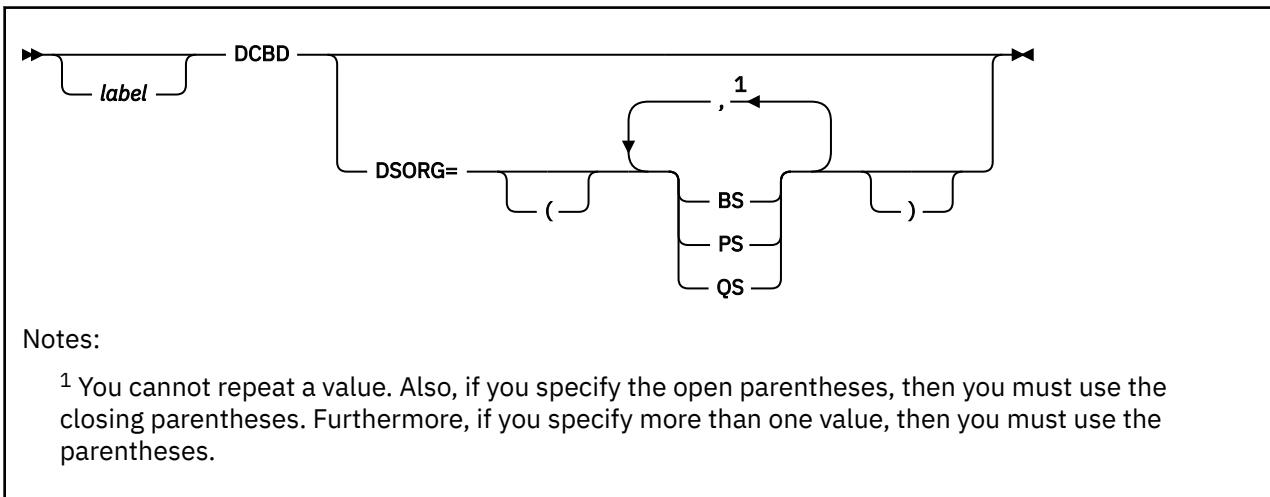
- b. Your SYNAD routine must preserve the contents of registers 13 and 14 as passed to it by GCS. Depending on your own requirements, it may also need to save the contents of registers 2 through 12. When control ultimately returns to your original program, registers 2 through 12 will contain the same values they contained when your SYNAD routine returned control to GCS. GCS does not restore your program's registers.
6. Note that GCS does not support the MVS parameter LRECL=X on this macro. If you include it, then an error will occur.
  7. When a DCB is used for output, it specifies a record format indicating variable length records. It is the your responsibility to supply the record length in the Record Descriptor Word (RDW). If this is not done, the results are unpredictable.

## Return Codes and ABEND Codes

The DCB macro generates no return codes and no ABEND codes.

## DCBD (BSAM/QSAM)

### Format



### Purpose

Use the DCBD macro to get the symbolic name for each field in a Data Control Block.

For a file to be of any use to you, a data control block (DCB) must be created for it. A DCB contains information that defines the characteristics of the data in the file and describes the I/O device requirements for handling the data.

As was explained in the entry titled “DCB (BSAM/QSAM)” on page 385, there are three ways of assigning a value to a field in a data control block through the:

- DCB macro
- FILEDEF command
- Direct insertion of a parameter's value or attribute into the data control block by your program.

The DCBD macro helps you with the third of these alternatives by producing a road map of the data control block that your program can follow while inserting certain values therein.

The DCBD macro creates a dummy control section (DSECT) modelled after a real data control block. Each field in this DSECT is assigned a symbolic name. Each symbolic name can be used as a displacement in an assembler language instruction to gain access to the corresponding field in the real data control block.

### Parameters

#### DSORG

Specifies the type of real data control block for which you want a DSECT created.

Data control blocks for BSAM files (BS parameter described in the following) and QSAM files (QS parameter described in the following) are constructed somewhat differently, though they do have fields in common. Also the PS parameter, described in the following, embraces the characteristics of both.

If you omit the DSORG parameter, then the DSECT will contain what is called a *foundation block*. A foundation block contains fields that are common to all three types of data control blocks but only those that are common.

You can specify one, two, or all three of the following parameters:

**BS**

Indicates that the data control block for which you want a DSECT created is associated with a basic sequential access file.

**PS**

Indicates that the data control block for which you want a DSECT created is associated with a physical sequential access file.

**QS**

Indicates that the data control block for which you want a DSECT created is associated with a queued sequential access file.

**Usage**

1. To use the DSECT to find your way around the data control block, simply assign the address of the DCB to a base register. Then, use the symbolic name of a field in the DSECT as the displacement to the corresponding field in the data control block.
2. You can use the same DSECT to insert data into more than one data control block. Just assign another DCB address your base register.
3. You are the one inserting data to the data control block, you must be certain that the data is inserted in a timely fashion. For example, it would be useless to insert the value of the DDNAME or the EXLST after issuing the OPEN macro, because that macro needs those values to run properly.
4. The following fields are generated by the DCBD macro:

17(11)	DCBDEVT	Device type.
26(1A)	DCBDSORG	Data set organization being used. X'40' PS -Physical sequential.
33(21)	DCBEODAD	End-of-data address.
36(24)	DCBRECFM	Record format. X'80' F - Fixed record length. X'40' V - Variable record length. X'C0' U - Undefined record length. X'10' B - Blocked records. X'04' A - ASA control characters. X'02' M - Machine control characters.
37(25)	DCBEXLST	Exit list.
40(28)	DCBDDNAM	Name of the data definition statement that defines the data set associated with the DCB.
42(2A)	DCBMACRF	Copy of the DCBMACR field used during and after OPEN.
50(32)	DCBMACR	Macro instruction reference before OPEN. Byte 1 X'20' R - READ X'04' P - POINT Byte 2 X'20' W - WRITE X'04' P - POINT
51(33)		Byte 1 X'20' G - GET X'10' M - Move mode. X'08' L - Locate mode. Byte 2 X'20' QSAM - Output P - PUT X'10' M - Move mode. X'08' L - Locate mode.
50(32)		
51(33)		
52(34)	DCBOPTCD	Option codes. X'01' J - Indicates that the first byte in the output data stream will be a 3800 table reference character.
57(39)	DCBSYNAD	Address of user's synchronous error routine.
62(3E)	DCBBLKSI	Maximum block size.

82(52)	DCBLRECL	Logical record length.
92(5C)	DCBE0B	Address of end of block module.

## Return Codes and ABEND Codes

The DCBD macro generates no return codes and no ABEND codes.

## GET (QSAM)

### Format



### Purpose

Use the GET macro to obtain the next logical record of a QSAM file for your program to process.

For a record in a QSAM data file to be processed, it must be transferred from its auxiliary storage device to main storage.

### Parameters

#### **dcb\_address**

Specifies the address of the data control block (DCB) associated with the QSAM file your program is processing.

A DCB contains information that defines the characteristics of the data stored in a file and describes the I/O device requirements for handling the data. You are responsible for having created a DCB for the file in question through the DCB macro. See “[DCB \(BSAM/QSAM\)](#)” on page 385.

You can write this parameter as an RX-type address or as register (1) through (12).

#### **area\_address**

Specifies the address of a work area into which GCS will place the next logical record.

This parameter is valid only if you are using the GET macro in MOVE mode. Moreover, it is your responsibility to provide storage for this work area in your program.

If you omit this parameter while operating in MOVE mode, then GCS assumes the address of the work area is in register 0. Otherwise you can write this parameter as an RX-type address, as register (0), or as register (2) through (12).

### Usage

1. The GET macro operates in one of two modes, namely MOVE and LOCATE. You declare which mode is to be used in obtaining records from a file when you create its data control block through the DCB macro.

#### **MOVE MODE**

GCS moves the next logical record of the file directly into the work area specified by the AREA ADDRESS parameter. The system assumes that you have provided a work area large enough to accept the largest record that may emerge from the file. If your file consists of variable-length records, then the work area must be large enough to accept the largest record plus its record descriptor word.

When the record has been successfully obtained, GCS returns the address of the work area in register 1.

#### **LOCATE MODE**

GCS moves the next logical record of the file to an input buffer. The system then places the length of the record in the DCBLRECL field of the file's data control block. It then returns the address of the input buffer in register 1.

- You may process the record in the input buffer or move it to a work area.
2. GCS assumes that the file being processed has been properly opened through the OPEN macro. See “[OPEN \(BSAM/QSAM\)](#)” on page 398.

## Return Codes and ABEND Codes

The GET macro generates no return codes.

ABEND Code	Meaning
005	Either an invalid address appears in the GET macro, or a required address parameter is missing.

## **NOTE (BSAM)**

### **Format**



### **Purpose**

Use the NOTE macro to obtain the relative position of the last block read or written in a BSAM file.

For many reasons, you may want to know the relative position of the last block you read from or wrote in a BSAM file. You may want to save the location of one or more of these blocks so that you can return to them at some later time.

The relative position of the block does not refer to its address on the disk or other such device. Rather, it refers to the block's position relative to the beginning of the file of which it is a part.

### **Parameters**

#### **dcb\_address**

Specifies the address of the data control block (DCB) associated with the BSAM file you are processing.

A DCB contains information that defines the characteristics of the data stored in a file and describes the I/O device requirements for handling its data. You are responsible for having created a DCB for the file in question through the DCB macro. See “[DCB \(BSAM/QSAM\)](#)” on page 385.

You can write this parameter as an RX-type address or as register (1) through (12).

### **Usage**

1. Before you enter the NOTE macro, you must confirm that the last I/O operation was completed successfully. Use the CHECK macro to accomplish this. See “[CHECK \(BSAM\)](#)” on page 380.
2. The NOTE macro returns the record ID (or relative position) of the last block read or written in register 1. This is the position of the record within the file relative to the beginning of the file, not to the beginning of the auxiliary storage device. The macro stores the record ID in the following format:

NNNz

NNN represents the 3-byte file system record number, and z, a byte of zeros. You must retain this value in a register or in virtual storage for future reference.

3. You can use the NOTE and POINT macros on any BSAM file. See “[POINT \(BSAM\)](#)” on page 402. However, you must inform GCS before you do through the MACRF parameter in the DCB macro.
4. GCS handles blocks as a group of records in a variable blocked file. Therefore the NOTE macro, for a variable blocked file, is passing back the file system record number of the last record read of the block. For fixed block files the NOTE macro is passing back the file system record of the first record read of the block.

### **Return Codes and ABEND Codes**

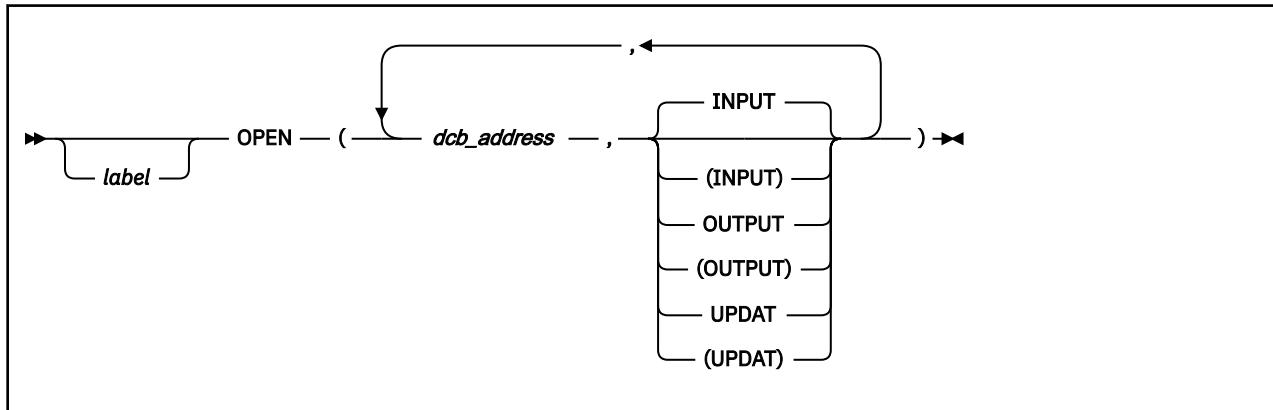
<b>ABEND Code</b>	<b>Meaning</b>
00A	Either you specified an invalid address in the NOTE macro, or an invalid address exists in the data control block associated with your file.

## **OPEN (BSAM/QSAM)**

The OPEN macro is available in standard, list and execute formats.

### **Standard Format**

See also “List Format” on page 399 and “Execute Format” on page 400.



### **Purpose**

Use the OPEN macro to open a file and prepare it for processing.

Before a program can use a file, they must be *logically connected* to each other. That is, GCS must be told where the file is and what its characteristics are. Usually this process is called *opening the file*.

### **Parameters**

#### **dcb\_address**

Specifies the address of the data control block associated with the file you want to open. For example, it is the address of the label on the DCB macro associated with your file. See “DCB (BSAM/QSAM)” on page 385.

You can write this parameter as an RX-type address or as register (2) through (12).

#### **INPUT**

Indicates that your file is to be treated as an input file. Unless otherwise specified, this parameter applies by default.

#### **OUTPUT**

Indicates that your file is to be treated as an output file.

You must specify this parameter if you are creating a new file.

#### **UPDAT**

Indicates that you intend to update an already existing file.

### **Usage**

1. Use of the OPEN macro to open a file assumes that the DCB macro has also been issued for that file.
2. The OPEN macro prepares your file for processing, then *logically connects* it to your program.

First, the information you supplied using the DCB macro and the FILEDEF command are merged into one data control block. For more information on the FILEDEF command, see “FILEDEF” on page 94.

Where an existing file is concerned, if any information necessary to the data control block is not provided by either of these sources, then it is taken from the attributes of the file itself.

Later, the exit routines specified in the DCB macro are executed and the processing method of your file (INPUT, OUTPUT, or UPDAT) is designated. After a few other details are taken care of, your file is ready for processing.

3. More than one file may be opened by a single OPEN macro. Just be certain that a comma delimits each entry in the list and that the entire list is surrounded by parentheses.
4. When choosing from among the INPUT, OUTPUT, and UPDAT parameters, be mindful of what was specified by the DCB macro in the MACRF parameter. In this respect, the OPEN and DCB macros must be compatible.

For example, if input macros were specified by the MACRF parameter, then the INPUT parameter must be applied to the corresponding OPEN macro.

5. Only the task that opened a file can close it.
6. To try to open a file that is already opened, with the same DCB, amounts to issuing a NOP (NO OPERATION) instruction.
7. It is an error to open a file specifying a DCB address that is not really the address of a data control block. The results of such an error are unpredictable.
8. If you have access method control blocks (ACBs) that you wish to open, and DCBs, then you can specify a combination of both in the same OPEN macro. GCS is able to distinguish the address of one from the address of the other, if you separate each with a comma.

## Return Codes and ABEND Codes

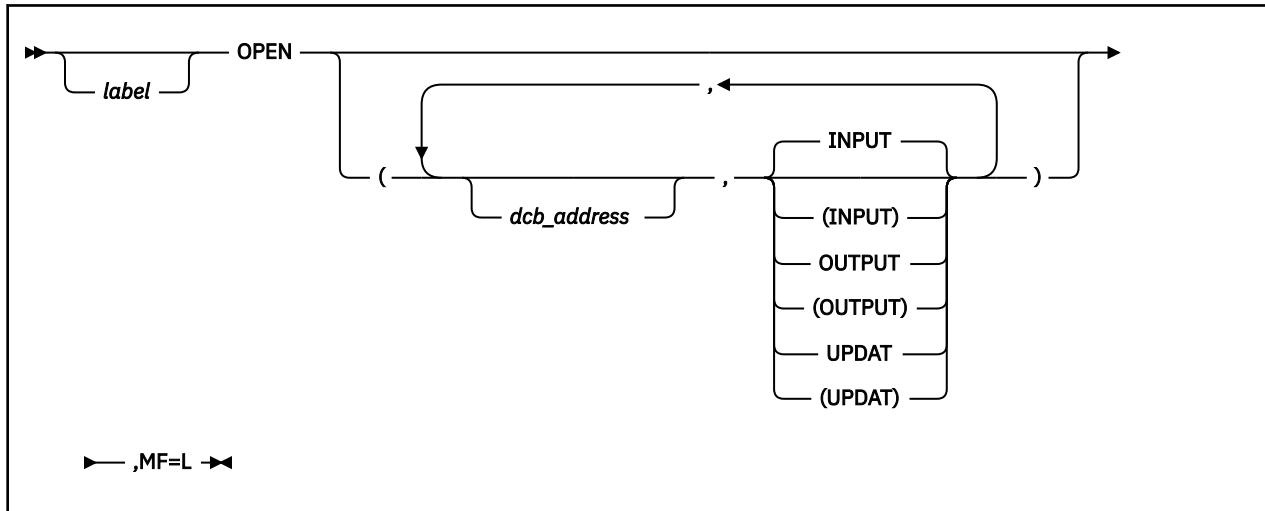
The OPEN macro generates no return codes.

---

ABEND Code	Meaning
013	An error occurred during the execution of the OPEN macro. You will receive a message explaining this further.

---

## List Format



## Purpose (List Format)

This format of the macro generates an in-line parameter list based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the

parameters using register notation. Also, note that only the preceding parameters listed are valid in the list format of this macro.

The parameter list consists of a one-word entry for each DCB in the parameter list. The high-order byte is reserved while the 3 low-order bytes contain the address of a DCB. The end of the list is marked by setting the high-order bit of the last entry to 1.

The length of the list generated by the list format of this macro must be equal to the maximum length required by an execute format macro that refers to the same list. A maximum length list can be constructed in one of two ways.

1. Enter the macro using the list format, with the maximum number of parameters required by the execute format of the macro that refers to the same list.
2. Use an appropriate number of commas in the list format of the macro to obtain a list of the required size. For example,

```
OPEN (,,,...,,),MF=L
```

would create a list of five fullwords.

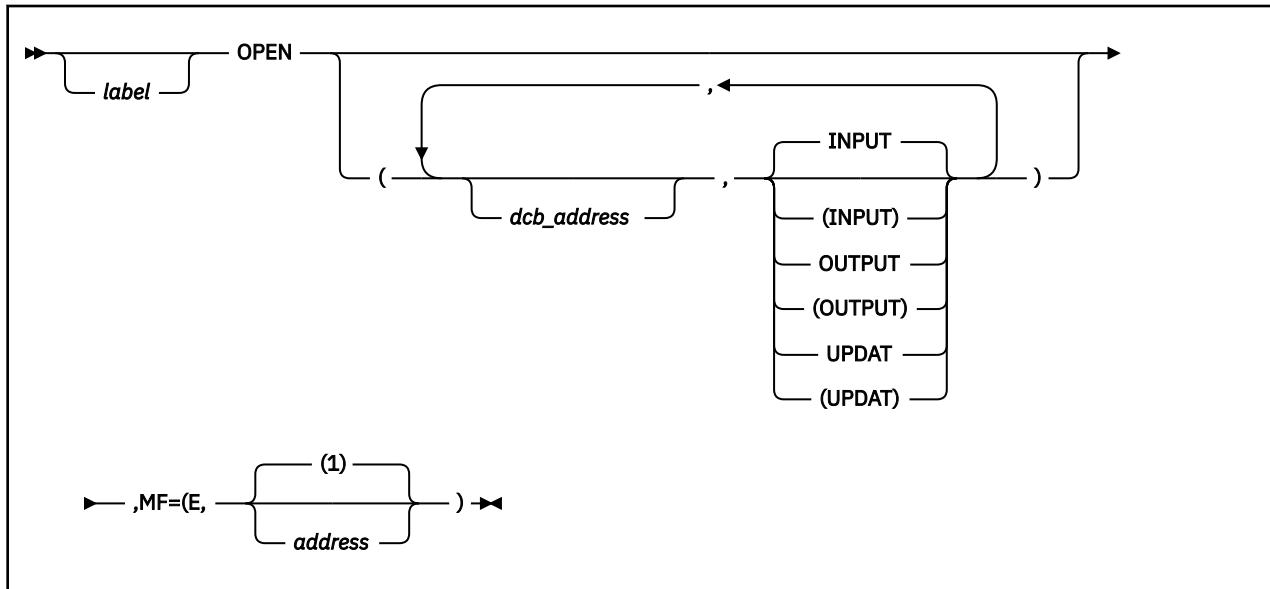
GCS assumes that any entries at the end of the list that are not referred to by the macro in the execute format were filled in by a previous instruction.

## Added Parameter

### **MF=L**

Specifies the list format of this macro.

## Execute Format



## Purpose (Execute Format)

This format of the macro generates code that executes the function using a parameter list whose address you specify. Only the preceding parameters listed are valid in the execute format of this macro.

## Added Parameter (Execute Format)

### **MF=(E, address)**

ADDRESS specifies the address of the parameter list to be used by the macro.

You can add or modify values in this parameter list by specifying them in this macro.

## **POINT (BSAM)**

### **Format**



### **Purpose**

Use the POINT macro to return to a specified block within a file.

As described in “[NOTE \(BSAM\)](#)” on page 396, the NOTE macro will give you the relative position of the last block read from or written in a file. You save one or more such locations with the intention of returning to them at some later time.

Use the POINT macro to return to one of the locations in a BSAM file that you saved through the NOTE macro. If you then enter a READ or WRITE macro, it is the block to which you have returned that will be read or written. See “[READ \(BSAM\)](#)” on page 406 and “[WRITE \(BSAM\)](#)” on page 413.

### **Parameters**

#### ***dcb\_address***

Specifies the address of the data control block (DCB) associated with the BSAM file you are processing.

A DCB contains information that defines the characteristics of the data stored in a file and describes the I/O device requirements for handling its data. You are responsible for having created a DCB for the file in question through the DCB macro. See “[DCB \(BSAM/QSAM\)](#)” on page 385.

You can write this parameter as an RX-type address or as register (1) through (12).

#### ***block\_address***

Specifies the address containing the record ID (or relative position) of the block that is to be processed next.

The record ID must be stored in a fullword on a fullword boundary.

You can write this parameter as an RX-type address, as register (0), or as register (2) through (12).

### **Usage**

1. Before you enter the POINT macro, you must confirm that the last I/O operation was completed successfully. Use the CHECK macro to accomplish this. See “[CHECK \(BSAM\)](#)” on page 380.
2. The POINT macro processes no file blocks. It merely positions a pointer to the block that is to be processed next.
3. The NOTE macro returns the record ID (or relative position) of the last block read or written in register 1. This is the position of the record within the file relative to the beginning of the file, not to the beginning of the auxiliary storage device. The macro stores the record ID in the following format:

NNNz

NNN represents the 3 byte file system record number, and z, a byte of zeros. Presumably you retained this value in a register or in virtual storage.

4. Usually, the low-order byte of the record ID is reset to 0. This indicates that the block to be affected by the next I/O instruction is the one to which the record ID points. If you set the low-order byte of the

record ID to 1, then you indicate that the block following the block to which the record ID points is to be processed.

5. If you are processing an output BSAM file, then you should enter one last WRITE macro before you close the file. This ensures that any altered block is written in the file.

### **Return Codes and ABEND Codes**

<b>ABEND Code</b>	<b>Meaning</b>
00A	You specified an invalid address in the POINT macro.

## PUT (QSAM)

### Format



### Purpose

Use the PUT macro to write the next logical record in a QSAM file.

### Parameters

#### *dcb\_address*

Specifies the address of the data control block (DCB) associated with the QSAM file your program is processing.

A DCB contains information that defines the characteristics of the data stored in a file and describes the I/O device requirements for handling its data. You are responsible for having created a DCB for the file in question through the DCB macro. See “[DCB \(BSAM/QSAM\)](#)” on page 385.

You can write this parameter as an RX-type address or as register (1) through (12).

#### *area\_address*

Specifies the address of a work area from which GCS will obtain the next logical record it will write in the file.

This parameter is valid only if you are using the PUT macro in MOVE mode. It is your responsibility to provide storage for this work area in your program.

If you omit this parameter while operating in MOVE mode, then GCS assumes the address of the work area is in register 0. Otherwise you can write this parameter as an RX-type address, as register (0), or as register (2) through (12).

### Usage

1. The PUT macro operates in one of two modes, namely MOVE and LOCATE. You declare which mode is to be used in writing records in a file when you create its data control block through the DCB macro.

#### MOVE MODE

GCS moves the next logical record to be written in the file from the work area specified by the AREA ADDRESS parameter to an output buffer. From there, the system moves the record to the auxiliary storage device containing the QSAM file in question. It then returns the address of the output buffer in register 1.

#### LOCATE MODE

The moment you enter the PUT macro, while operating in LOCATE mode, GCS writes in the QSAM file the last record you built in the output buffer. It then returns the address of the next available output buffer to you in register 1. It is at this address where your program builds the next record to be written in the file. The system does not write this record in the file until you enter the PUT instruction again.

2. GCS assumes that the file being processed has been properly opened through the OPEN macro. See “[OPEN \(BSAM/QSAM\)](#)” on page 398.

## Return Codes and ABEND Codes

The PUT macro generates no return codes.

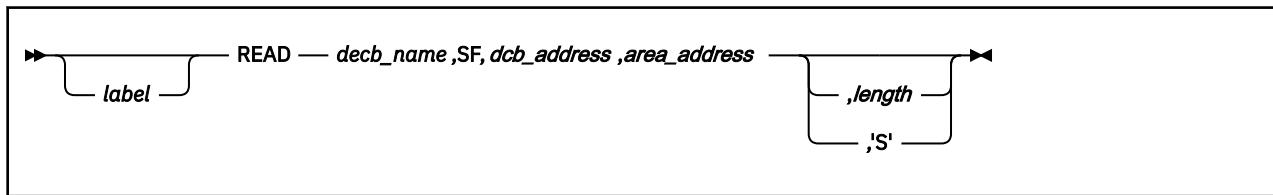
ABEND Code	Meaning
005	Either an invalid address appears in the PUT instruction, or a required address parameter is missing.

## READ (BSAM)

The READ macro is available in standard, list and execute formats.

### Standard Format

See also “List Format” on page 407 and “Execute Format” on page 408.



### Purpose

Use the READ macro to retrieve a block of data from a BSAM disk or reader file and place it into a specified area of your virtual storage.

When obtaining input from a file, your application is responsible for blocking and unblocking the data.

### Parameters

#### *decb\_name*

Specifies the label that you want applied to the data event control block.

A data event control block (DECB) is created within the expansion of the READ macro. It contains information that describes the input *event* you want to effect. The DECB will be defined in detail later. As the DECB expands within the macro, requires a label which you must supply. You will use this label to access the DECB itself.

You must write this parameter as an assembler program label.

#### **SF**

Indicates that a usual, sequential, and forward retrieval access method will be used in obtaining the block from your file.

This is the only method of extracting data from a BSAM file that GCS supports, the SF parameter is required and must be written exactly as shown.

#### *dcb\_address*

Specifies the address of the data control block (DCB) associated with the BSAM file you are processing.

A DCB contains information that defines the characteristics of the data stored in a file and describes the I/O device requirements for handling its data. You are responsible for having created a DCB for the file in question through the DCB macro. See “DCB (BSAM/QSAM)” on page 385.

You can write this parameter as an RX-type address or as register (1) through (12).

#### *area\_address*

Specifies the address in your virtual storage where you want the input block placed.

It is your program's responsibility to provide and manage this area of storage.

You can write this parameter as an assembler program label or as register (2) through (12).

#### *length*

Specifies the number of bytes you want extracted from your file.

GCS begins extracting the data starting with the next available record, as indicated by the data control block (DCB) associated with your file. This data will be placed in virtual storage starting at the address specified by the AREA ADDRESS parameter.

You can write this parameter as any number from 1 to 32760.

### 'S'

Indicates that the number of bytes to be extracted from your file will be the number found in the DCBLRECL field of the file's DCB.

This is the same number you specified previously for the LRECL parameter in the FILEDEF command or the DCB macro. See “FILEDEF” on page 94, and “DCB (BSAM/QSAM)” on page 385.

## Usage

1. Control may return to your program before the READ macro completes processing. Therefore, you must enter the CHECK macro after each READ instruction to be certain that the latter executed properly. By using the CHECK macro you confirm whether the input from your file has succeeded, failed, or is incomplete. See “CHECK (BSAM)” on page 380.
2. If you specified the UPDAT parameter in the OPEN instruction that opened your file, then both the READ and WRITE macros must use the same DECB name. See “OPEN (BSAM/QSAM)” on page 398 and “WRITE (BSAM)” on page 413.
3. The data event control block is created as part of the READ macro expansion. It defines the input event using the following format.

---

0 (0)	ECB
4 (4)	Type of I/O request, thus: 0000 1000 ---> READ
6 (6)	Length of the block being extracted
8 (8)	Address of the data control block (DCB)
12 (C)	Address in your virtual storage where the block is to be placed
16 (10)	Zeros

---

The address of the logical input block is placed in the DECB at 12 (C). It is through this address that you manage the data in the block.

## Return Codes and ABEND Codes

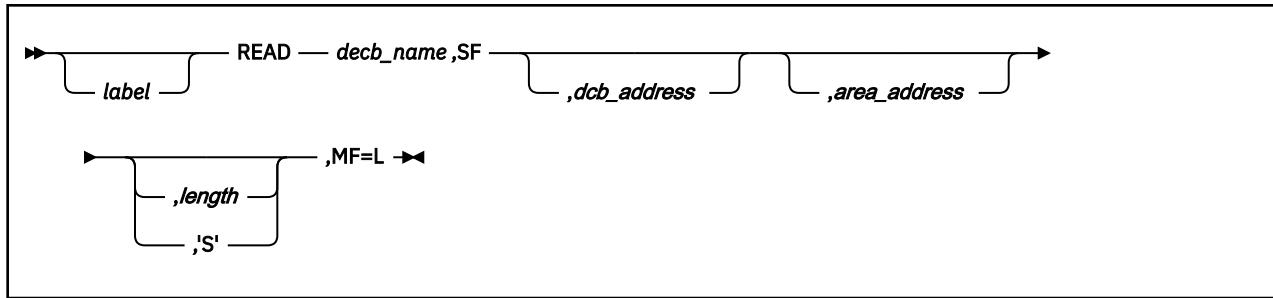
The READ macro generates no return codes.

---

ABEND Code	Meaning
001	An I/O error occurred but no SYNAD routine address was found in the file's DCB.
005	Either you specified an invalid address, or an address was missing.
010	You specified a parameter not supported by GCS.

---

## List Format



### Purpose (List Format)

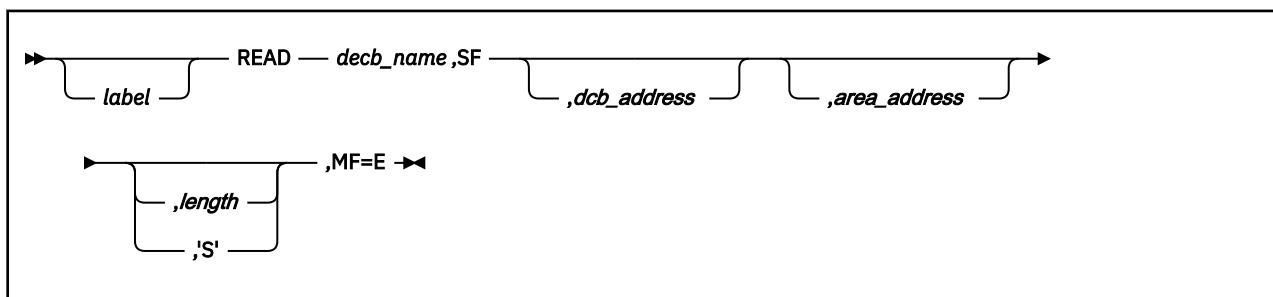
This format of the macro generates an in-line DECB based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation.

### Added Parameter

#### **MF=L**

Specifies the list format of this macro.

### Execute Format



### Purpose (Execute Format)

This format of the macro generates code that executes the function. The access method uses the DECB whose name you specify as the parameter address.

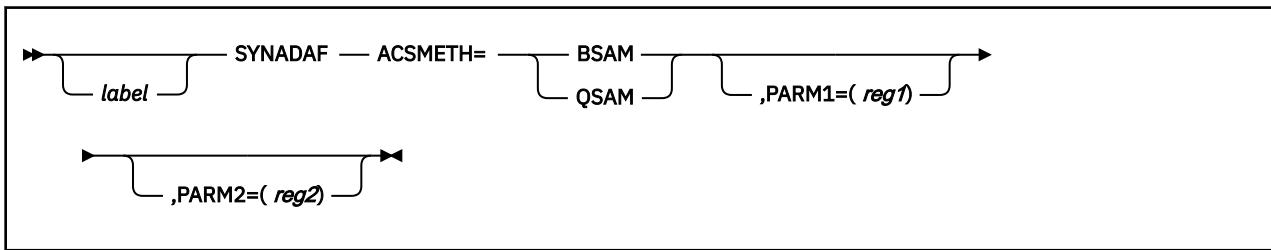
### Added Parameter (Execute Format)

#### **MF=E**

Specifies the execute format of this macro.

## SYNADAF (BSAM/QSAM)

### Format



### Purpose

Use the SYNADAF macro to obtain a message and an error code that explain an I/O error.

During input or output, errors sometimes occur. When they do, one of two things happens:

- Your task ends abnormally
- Your SYNAD routine receives control, if you have provided one.

A SYNAD routine is a program that you provide to analyze the cause of any permanent I/O error your task meets. When you define the data control block (DCB) associated with a file, you can also identify a SYNAD routine for that file. See “[DCB \(BSAM/QSAM\)](#)” on page 385.

You can write your SYNAD routine to determine the cause and type of the error by examining the:

- Contents of the registers at the moment of error
- Data event control block (DECB) associated with the I/O *event* that caused the error (this applies only to BSAM files)
- Exceptional condition code
- Standard status and sense indicators.

Often it is simpler to enter the SYNADAF macro, which will return a message and error code to you describing the I/O error.

### Parameters

#### ACSMETH

Specifies the access method you are using on the file in question. Specify either BSAM or QSAM.

#### PARM1

Specifies the number of the register containing the information that was in register 1 when your SYNAD routine received control.

When the error occurred, GCS gained control. After it attempted to recover from the error, it passed control to your SYNAD routine. In so doing, GCS passed the following information to your routine in register 1.

- Status bits
- Flag bits
- The address of the data control block (DCB) associated with the file being processed when the error occurred.

If you moved this data to another register, then write the number of that register, surrounding it with parentheses. If you omit this parameter, then GCS assumes that you left this data in register 1.

**PARM2**

Specifies the number of the register containing the information that was in register 0 when your SYNAD routine received control.

When GCS passed control to your SYNAD routine, it also passed certain status and control information in register 0.

If you moved the data to another register, then write the register number, surrounding it with parentheses. If you omit this parameter, then GCS assumes that you left this data in register 0.

**Usage**

1. The SYNADAF macro returns the address of a buffer to you in register 1. This buffer contains a 120-byte message, describing the result of its error analysis. The format of this message is:

<b>Bytes</b>	<b>Contents</b>
0-43	BLANK. You can add your own comments to the message in this field, if you wish.
44-83	GCTSER306S INPUT ERROR nnn ON ddname OR GCTSER307S OUTPUT ERROR nnn ON ddname  nnn specifies an I/O error code. For more information on the explanation of messages GCT306S or GCT307S, see <a href="#">z/VM: Other Components Messages and Codes</a> .  ddname specifies the name of the file in question.
84-119	BLANK.

2. The message describing the SYNADAF macro's error analysis is a variable-length record containing EBCDIC data. If you wish, you can have this message printed.

**Return Codes and ABEND Codes**

The SYNADAF macro generates no return codes.

<b>ABEND Code</b>	<b>Meaning</b>
144	The high-order byte of register 15 should have contained X'02' or X'03' on entry to the SYNADAF SVC routine. It did not.
244	The caller provided an invalid save area address in register 13.
344	Either the DCB address or the DCB DEB address was invalid.
444	The DECB address was invalid.

## SYNADRLS (BSAM/QSAM)

### Format



### Purpose

Use the SYNADRLS macro to release the message buffer and save areas created by the SYNADAF macro.

When you enter the SYNADAF macro from your SYNAD routine, a message buffer, parameter save area, and register save area are created. See “[SYNADAF \(BSAM/QSAM\)](#)” on page 409.

These storage areas must be released after they are no longer needed. The values contained in the registers before you issued the SYNADAF macro must be restored. Use the SYNADRLS macro to effect this.

### Parameters

The SYNADRLS macro accepts no parameters.

### Usage

- Before you enter the SYNADRLS macro, be certain that register 13 contains the address of the register save area provided by the SYNADAF macro. This save area contains the values of the registers that were present before you issued the SYNADAF macro.

The SYNADRLS macro restores these registers and releases the message buffer, parameter save area, and register save area.

The SYNADRLS macro then loads register 13 with the address of another save area. This area contains the values of the supervisor's registers that were present when it passed control to your SYNAD routine. The third word of this save area is reset to zero, because the next area in the chain was just released.

Everything is then restored to its condition before you enter the SYNADAF macro.

### Return Codes and ABEND Codes

When this macro completes processing, it passes to the caller a return code in the low-order byte of register 0.

Hex Code	Decimal Code	Meaning
X'00'	0	Function completed successfully.
X'08'	8	Function completed unsuccessfully, and nothing has changed. Either register 13 does not point to the save area provided by the SYNADAF macro, or this save area is improperly chained to the save area containing the supervisor's registers.

## **SYNADRLS (BSAM/QSAM)**

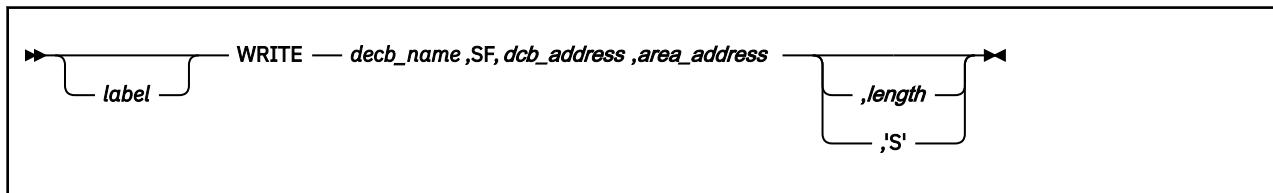
<b>ABEND Code</b>	<b>Meaning</b>
944	Either the address of SYNADAF's save area or the pointer to the caller's save area is invalid.

## WRITE (BSAM)

The WRITE macro is available in standard, list and execute formats.

### Standard Format

See also “List Format” on page 414 and “Execute Format” on page 415.



### Purpose

Use the WRITE macro to add or replace a block of data in a disk file, or to add a block to a punch or printer file.

When using BSAM to place output in a file, your application is responsible for blocking and unblocking the data.

### Parameters

#### *decb\_name*

Specifies the label that you want applied to the data event control block.

A data event control block (DECB) is created within the expansion of the WRITE macro. It contains information that describes the output *event* you want to effect. The DECB will be defined in detail later. As the DECB expands within the macro, it requires a label which you must supply.

You must write this parameter as an assembler program label.

#### **SF**

Indicates that a usual, sequential, forward retrieval access method will be used in placing the block in your BSAM file.

This is the only method of placing data in a BSAM file that GCS supports, the SF parameter is required and must be written exactly as shown.

#### *dcb\_address*

Specifies the address of the data control block (DCB) associated with the file you are processing.

A DCB contains information that defines the characteristics of the data stored in a file and describes the I/O device requirements for handling its data. You are responsible for having created a DCB for the file in question through the DCB macro. See “DCB (BSAM/QSAM)” on page 385.

You can write this parameter as an RX-type address or as register (1) through (12).

#### *area\_address*

Specifies the address in your virtual storage that contains the output block you want placed in your file.

It is your program's responsibility to provide and manage this area of storage.

You can write this parameter as an assembler program label or as register (2) through (12).

#### *length*

Specifies the number of bytes that you want placed in your file.

## **WRITE (BSAM)**

GCS will begin the block at the next available record in your file, as indicated by the file's data control block (DCB). The data placed there will be taken from the area specified by the AREA ADDRESS parameter.

You can write this parameter as any number from 1 to 32760.

### **'S'**

Indicates that the number of bytes to be placed in your file will be the number found in the DCBBLKSI field of the file's DCB.

## **Usage**

1. Control may return to your program before the WRITE macro completes execution. Therefore, it is required that you enter the CHECK macro after each WRITE macro to be certain that the latter executed properly. By using the CHECK macro you confirm whether the output to your file has failed or is incomplete. See "["CHECK \(BSAM\)" on page 380](#).
2. If you specified the UPDAT parameter in the OPEN macro when you opened your file, then both the READ and WRITE macros must use the same DECB name. See "["OPEN \(BSAM/QSAM\)" on page 398](#) and "["READ \(BSAM\)" on page 406](#).
3. If RECFM=FB (fixed block) it is the user's responsibility to ensure that the blocksize (DCBBLKSI) field in the DCB is correct before the write is issued. If the write will be for a short block, see the DCBD macro (see "["DCBD \(BSAM/QSAM\)" on page 391](#)) for how to map the DCB to update the DCBBLKSI field with the correct value.
4. The data event control block (DECB) is created as part of the WRITE macro expansion. It defines the output event using the following format.

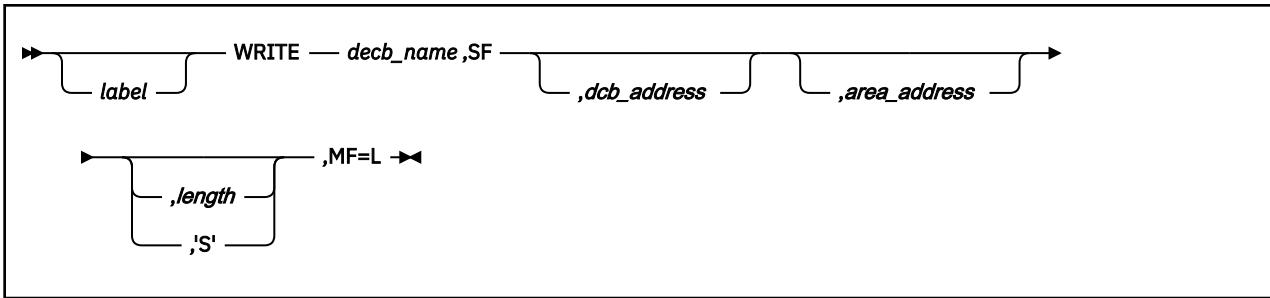
<b>Code</b>	<b>Meaning</b>
0 (0)	ECB
4 (4)	Type of I/O request, thus: 0000 0010 ---> WRITE
6 (6)	Length of the block being written
8 (8)	Address of the data control block (DCB)
12 (C)	Address in your virtual storage where the block can be found
16 (10)	Zeros

## **Return Codes and ABEND Codes**

The WRITE macro generates no return codes.

<b>ABEND Code</b>	<b>Meaning</b>
005	Either you specified an invalid address or an address was missing.

## **List Format**



### Purpose (List Format)

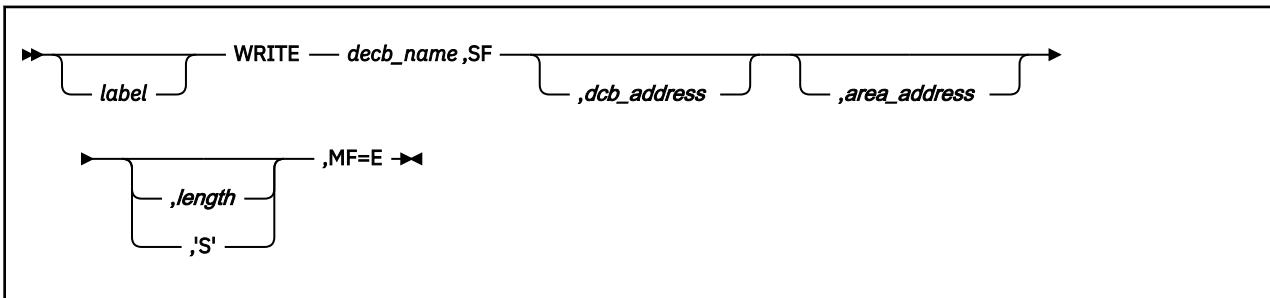
This format of the macro generates an in-line DECB, based on the parameter values that you specify. However, this format generates no executable code. Remember that you cannot specify any of the parameters using register notation.

### Added Parameter

#### **MF=L**

Specifies the list format of this macro.

### Execute Format



### Purpose (Execute Format)

This format of the macro generates code that executes the function. The access method uses the DECB whose name you specify.

### Added Parameter (Execute Format)

#### **MF=E**

Specifies the execute format of this macro.

**WRITE (BSAM)**

---

# Chapter 7. VSAM Data Management Service Macros

The VSAM data management service macros are presented in alphabetic order in the section. Additional information about VSAM is in [Appendix B, “Using VSAM,” on page 517](#). The GCS macros are described in [Chapter 5, “GCS Macros,” on page 157](#). The QSAM and BSAM data management service macros are described in [Chapter 6, “QSAM and BSAM Data Management Service Macros,” on page 379](#).

Any user applications using branch entries into VSAM data management service macros must be in AMODE 24.

The VSAM data management service macros are:

ACB  
BLDVRP  
CHECK  
CLOSE  
DLVRP  
ENDREQ  
ERASE  
EXLST  
GENCB  
GET  
MODCB  
OPEN  
POINT  
PUT  
RPL  
SHOWCAT  
SHOWCB  
TESTCB  
WRTBFR.

---

## Using VSAM

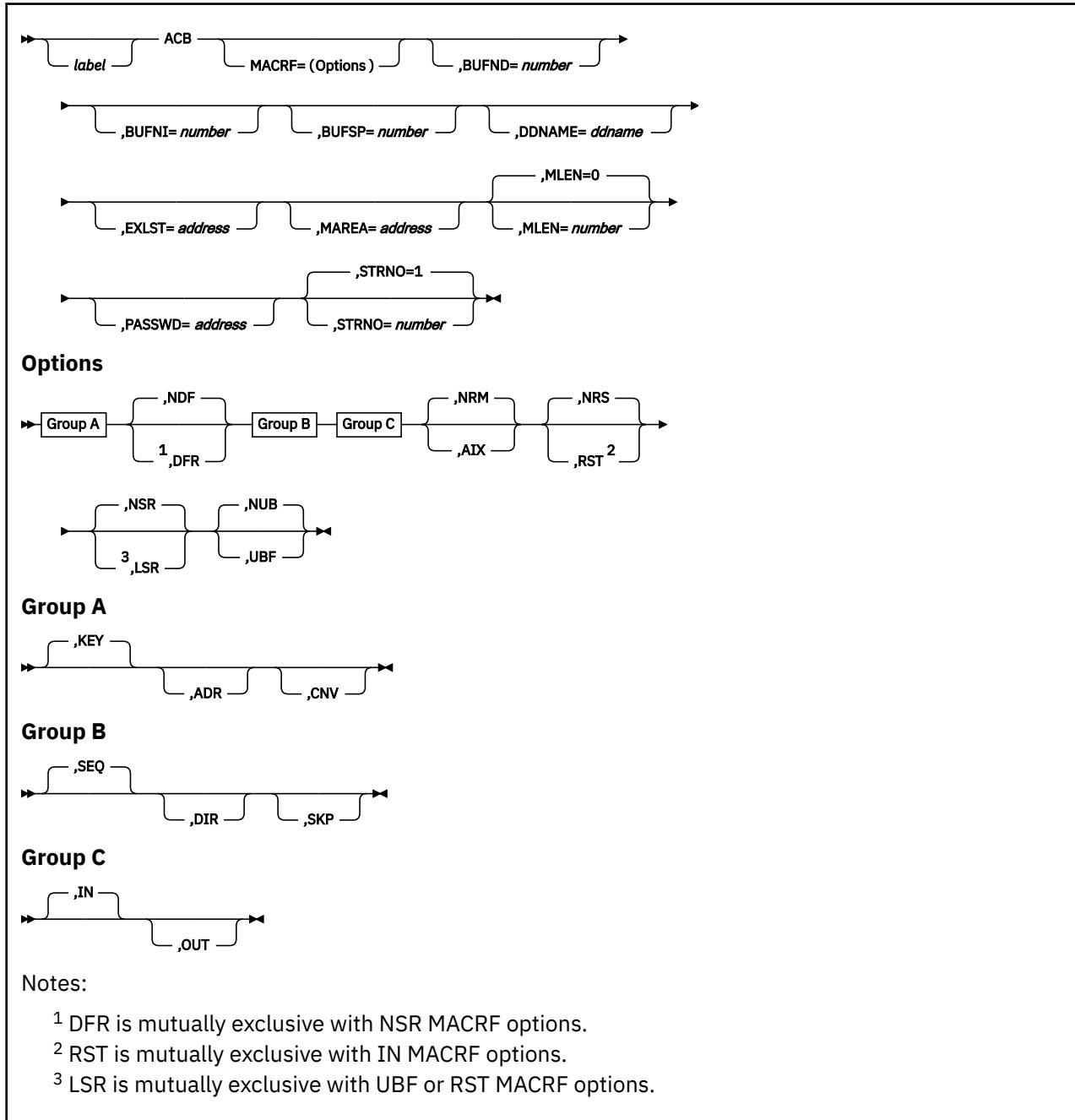
Because VSAM data management service is provided only below the 16MB line, all addresses provided through VSAM data management service macros must adhere to all:

- Branch entries to VSAM support must be in 24-bit address mode (AMODE 24)
- Calls are accepted in only the 24-bit address mode (AMODE 24)
- Addresses are accepted in only the 24-bit address mode (AMODE 24)
- Addresses must point to storage below the 16MB line
- User exits must reside in virtual storage below the 16MB line.
- Calls to any of these services cannot be made in AR mode.
- Using VSAM compression services, VSAM will compress and expand a KSDS, ESDS, or VRDS VSAM dataset to conserve DASD resources.

## ACB

---

### Format



### Purpose

Use the ACB macro to create an ACB and define certain characteristics of your file.

An access method control block (ACB) defines certain characteristics of a file that you intend to process through VSE/VSAM. When the file is opened, other characteristics of the file that you defined through the DLBL command are merged with the ACB to complete the picture. For more information on the DLBL command, see “[DLBL](#)” on page 64.

This discussion of the ACB macro deals only with those matters that involve GCS.

## Parameters

### **MACRF**

Indicates how you intend to access the file.

You must specify all of the types of processing you intend to perform on the file, whether you intend to perform them concurrently or alternately. The parameters you choose must be valid for the file in question. For example, if you specify keyed access for an entry-sequenced file, then you cannot open that file, much less process it.

Check the preceding format box. The processing options are arranged in groups, each with a value that will be assumed by default should you forget to specify from that group. Because they are not positional parameters, they can be specified in any order.

### **KEY**

Indicates access to a key-sequenced or relative record file.

Keys will be relative record numbers used as search arguments, and sequential access will be by key or relative record number.

### **ADR**

Indicates addressed access to a key-sequenced or entry-sequenced file.

RBAs will be used as search arguments, and sequential access is by entry sequence.

### **CNV**

Indicates access will be to the entire contents of a control interval rather than to an individual record.

### **NDF**

Indicates that any WRITE macro will not be deferred for a direct PUT macro.

### **DFR**

Specifies that physically writing the I/O buffers is deferred when possible.

### **SEQ**

Indicates sequential access to a key-sequenced, entry-sequenced, or relative record file.

### **DIR**

Indicates direct access to a key-sequenced, entry-sequenced, or relative record file.

### **SKP**

Indicates skip-sequential access to a key-sequenced or relative record file.

This is valid only with keyed access in a forward direction.

### **IN**

Indicates retrieval of records from key-sequenced, entry-sequenced, or relative record files.

This is not a valid form of processing for an empty file.

### **OUT**

Indicates several things:

- Storage of new records in a key-sequenced, entry-sequenced, or relative record file. This is not allowed with addressed access to a key-sequenced file.
- Update of new records in a key-sequenced, entry-sequenced, or relative record file.
- Deletion of records from a key-sequenced or relative record file.
- Retrieval of records as described under the IN parameter. To select the OUT parameter is to select the IN parameter, by implication.

### **NRM**

Indicates that the file to be processed is the one specified by the DDNAME parameter.

**AIX\***

Indicates that the object to be processed is the alternate index of the path specified by the DDNAME parameter, rather than the base cluster through the alternate index.

**NRS**

Indicates that the file is not reusable.

**RST**

Indicates that the file is reusable.

Note that the OPEN macro resets the file's catalog information to its original status. That is, it resets it to the status it had before the file was open the first time. See "["OPEN"](#) on page 465. Also, the high-used RBA is reset to zero.

The file must have been defined with the REUSE attribute for RST to be effective. Although the file is not erased, you can handle it as though it were a new file, and use it as a work file. When the OPEN macro carries out the reset operation, this parameter is equal to the OUT option. DISP=NEW specified on the DLBL command is equal to selecting this parameter and will override the NRS parameter.

**NSR**

Indicates that the resources are not shared.

**LSR**

Specifies that the resources are shared. This also indicates a VSAM resource pool will be provided opening this ACB.

**NUB**

Indicates that VSAM will manage the I/O buffers.

**UBF**

Indicates that the application will manage the I/O buffers.

The work area specified by the RPL or GENCB macros will be, in effect, the I/O buffer. The contents of a control interval is transmitted directly between the work area and DASD. This parameter is valid only when the MACRF=CNV and OPTCD=MVE parameters are specified in the RPL macro. See "["RPL"](#) on page 472 and "["GENCB"](#) on page 437.

**BUFND**

Specifies the number of I/O buffers used in transmitting data between virtual and auxiliary storage.

The size of a buffer corresponds to the size of a control interval in the data component. The minimum number you can specify is 1 plus the number specified by the STRNO parameter. If you omit the STRNO parameter, then the value of the BUFND parameter must be at least 2 because the default for the former is 1.

The default for the BUFND parameter is the minimum number required to process your file.

**BUFNI**

Specifies the number of I/O buffers to be used for transmitting the contents of index entries between virtual and auxiliary storage during keyed access.

The size of this buffer corresponds to the size of a control interval in the index. The minimum number you can specify is 1 plus the number specified by the STRNO parameter. If you omit the STRNO parameter, then the value of BUFNI parameter must be at least 2 because the default for the former is 1.

The default for the BUFNI parameter is the minimum number required to process your file.

**BUFSP**

Specifies the maximum number of bytes of virtual storage to be used for the data and index I/O buffers.

This parameter must be at least as large as the buffer size recorded in the catalog entry for your file. If the number you specify for this parameter is too small, then VSAM overrides it and uses the buffer size recorded in the catalog. VSAM, however, does not inform you of this.

If you omit this parameter, then the size of this buffer will be the largest of the following, by default:

- The buffer size specified in the catalog.

This buffer size was specified through the BUFFERSPACE parameter in the Access Method Services DEFINE command. If this parameter was omitted when your file was defined, then a default value was assigned to it. This default value, the minimum amount of buffer space allowed by VSAM, is enough to hold two data control intervals and one index control interval.

- The buffer size determined from the BUFND and BUFNI parameters.

You can also specify buffer space through the BUFSP parameter on the DLBL command that identifies your file. This value overrides the BUFSP parameter in the ACB macro. It overrides the BUFFERSPACE parameter in the DEFINE command if the latter is smaller.

If the values you specify for the BUFND, BUFNI, and BUFSP parameters are inconsistent, then VSAM increases the number of buffers to conform with the size of the buffer area. If the value in the BUFSP parameter is greater than the minimum buffer size required to process your file and greater than the values specified in the BUFND and BUFNI parameters, then the extra space is allocated between the data and index buffers if the MACRF parameter specifies:

- Direct processing, then the values in the BUFND and BUFNI parameters take effect. Any left-over space is used for index buffers.
- Sequential processing, then the values in the BUFND and BUFNI parameters take effect. Space for one additional index buffer is allocated. Any left-over space is used for data buffers. If any left-over space remains that is insufficient to hold another data buffer, then it is used for another index buffer.

If the value in the BUFSP parameter is greater than the minimum required to process your file, but less than those of the BUFND and BUFNI parameters, then enough buffer space will be made available to conform to the latter parameters.

If you provide your own pool of I/O buffers for control interval processing, then the BUFSP, BUFND, and BUFNI parameters have no effect. In such a case, the AREA and AREALEN parameters of the RPL macro determine the size of the user buffer area. See [“RPL” on page 472](#).

#### **DDNAME**

Specifies the name of the file you wish to process.

This name corresponds to that specified in the DDNAME parameter of the DLBL command associated with the file. If you omit this parameter, then you can supply it through the MODCB macro. See [“MODCB” on page 453](#).

This name must be from one to seven characters long.

#### **EXLST**

Specifies the address of a list of exit routine addresses.

This is the same list that you created through the EXLST or GENCB macro. See [“EXLST” on page 434](#) or [“GENCB” on page 437](#).

If you used the EXLST macro to create this list, then you can write this parameter as the label on that instruction. If you used the GENCB macro, then you can write this parameter as the address that the GENCB macro returned to you in register 1 or as the label associated with an area into which you have placed this address.

If you omit this parameter, then GCS assumes that you have supplied no exit routines.

#### **MAREA**

Specifies the address of an area into which GCS will place any console messages generated during processing of your file.

This area can be used by you or your exit routines to analyze any errors or problems that may arise.

#### **MLEN**

Specifies the length, in bytes, of the area whose address is given by the MAREA parameter.

The value of this parameter is zero, by default. Its maximum value is 32K.

**PASSWD**

Specifies the address of a field that contains the highest level password required for the types of access indicated by the MACRF parameter.

The first byte of the field contains the binary length of the password. Eight bytes is the maximum length. If this byte is zero, it means that you are providing no password.

If your file is password protected and you provide none, then VSAM will ask you to provide the password when it opens the file.

**STRNO**

Specifies the number of requests you will make that will require concurrent file positioning.

A request is defined by a given request parameter list or a chain thereof. If records are written in an empty file, then the value of this parameter is ignored and replaced by the value 1.

If you omit this parameter, then its value is 1, by default.

**Usage**

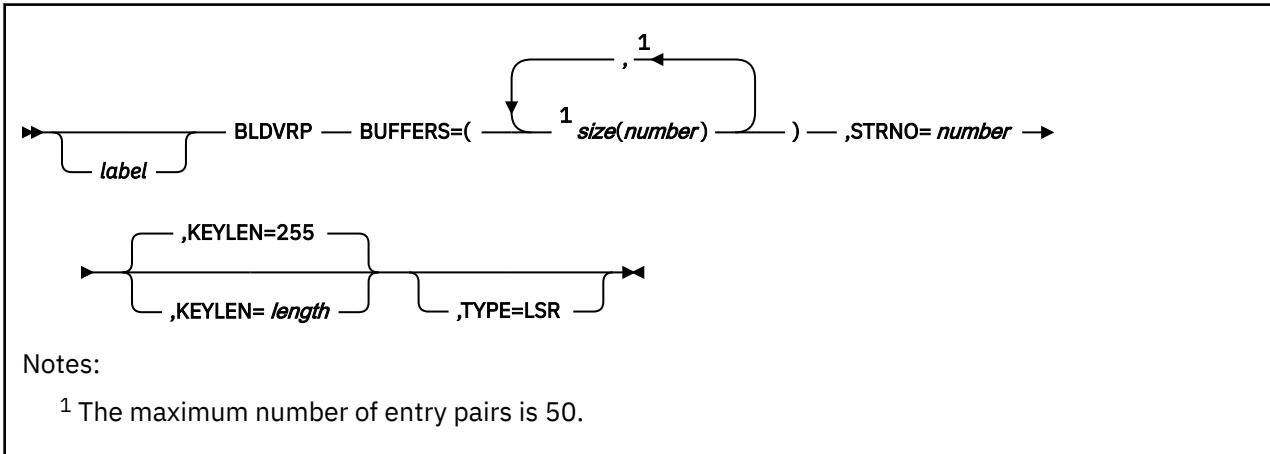
1. The ACB macro creates an access control block at assembly time. Contrast this with the GENCB macro which generates an ACB at execution time. See "[GENCB](#)" on page 437.
2. See [Appendix B, "Using VSAM," on page 517](#).
3. LSR cannot be used with a SHAREOPTION(4) file and cannot be used to initially load an empty file.
4. Appropriate macro MNOTES notify the application programmer of syntax errors in coding the ACB macro.

**Return Codes and ABEND Codes**

The ACB macro generates no return codes and no ABEND codes.

# BLDVRP

## Format



## Purpose

Use the BLDVRP macro to build a resource pool before you open any file that uses Local Shared Resources (LSR).

After a resource pool exists for a virtual machine, every OPEN of an ACB that indicates use of LSR will result in use of the resource pool to provide I/O buffers, I/O control blocks, and channel programs as required.

To specify the BUFFERS, KEYLEN, and STRNO operands of the BLDVRP macro, you must have knowledge of the size of the control intervals, data records, and key fields in the components that will use the resource pool. You must also know the way the components are processed. The SHOWCAT macro can be used to get that information before the file is opened.

### Note:

1. The SHOWCB macro lets you collect statistics about the usage of buffer pools. This information can be used in following runs of your program to optimize the characteristics of the resource pool to the program requirements.
2. VM/VSAM will support only one resource pool within a virtual machine.

## Parameters

### BUFFERS=

Specifies the size and the number of buffers for each buffer pool in the resource pool. The number of buffer pools in the resource pool is implied by the number of *size(number)* pairs you specify. You should usually set the buffer sizes equal to the control interval sizes of the file objects you want to process.

**Note:** If you do not specify the exact buffer size required by a component of the file, VSAM will use buffers from the buffer pool with the next larger buffer size.

### size

Specifies the number of bytes in the buffer (512, 1024, 2048, 4096, 8192, 12288, 16384, 20480, 24576, 28672, or 32768).

**Note:** The macro interface does not support use of the K notation in specifying buffer size.

***number***

Specifies the number of buffers of a given size which must be at least three.

**STRNO**

Specifies the maximum number of requests that may be issued concurrently for all of the files that are to share the resource pool. The number must be at least one and no more than 255.

**Note:** To make sure you are using the resource pool effectively you can enter SHOWCB ACB=addr,FIELDS=(STRMAX) in your application program. Depending on the result, you may want to redefine STRNO=number the next time you build your resource pool.

**KEYLEN**

Specifies the maximum key length (relative record number) of the files that are to share the resource pool. The default is 255.

The key length of a relative record file is four. If the buffer pool contains buffers for entry-sequenced files only, specify KEYLEN=0. To find out the key length of a file, enter the SHOWCAT macro for that file.

**TYPE=LSR**

Specifies the resource pool is used for Local Shared Resources.

**Usage**

1. When using this macro you must make sure that register 13 contains the address of a 72-byte save area. If you enter the macro from within one of your exit routines (LERAD or SYNAD) you must provide a second 72-byte save area because the original one is still in use by the external VSAM routine.
2. When VSAM returns to the application after a BLDVRP request, register 15 contains one of the following completion codes:

Completion Code	Meaning
0	VSAM completed the request successfully.
4	A resource pool already exists for this virtual machine.
8	An error was detected while VSAM routines were being loaded.
16	<ul style="list-style-type: none"> <li>• An unsupported parameter was specified on macro.</li> <li>• Either the FIX keyword or TYPE=GSR was specified on the macro call.</li> </ul>
20	STRNO was specified as less than one or greater than 255.
24	Size or number specified with BUFFERS is invalid.

# CHECK

---

## Format



## Purpose

Use the CHECK macro to place your task in the WAIT state while it waits for a certain VSAM request to take place.

## Parameters

### RPL

Specifies the address of the request parameter list (RPL) associated with the VSAM request in question.

This is the same request parameter list that you created through the RPL macro. See “[RPL](#)” on page [472](#).

You can write this parameter as an assembler program label or as register (2) through (12).

## Usage

1. VSAM requests are associated with these macros:

- ENDREQ
- ERASE
- GET
- POINT
- PUT.

If you specified asynchronous processing (OPTCD=ASY) in the RPL macro, enter the CHECK macro after each of these instructions.

2. The request parameter list associated with your VSAM request can specify the ASY option.

This indicates that you want your request processed asynchronously. Remember, though, that asynchronous processing is merely simulated by GCS. Disk I/O in GCS is always synchronous.

3. See [Appendix B, “Using VSAM,” on page 517](#).

## Return Codes and ABEND Codes

When this macro completes processing, it passes to the caller a return code in register 15. If the return code is 0, 8, or 12, then the macro also returns a feedback code in the FDBK field of the RPL associated with the request. This field can be checked through the SHOWCB or TESTCB macros. See “[SHOWCB](#)” on page [480](#) or “[TESTCB](#)” on page [490](#).

---

Hex Code	Decimal Code	Meaning
X'00'	0	Function completed successfully.

---

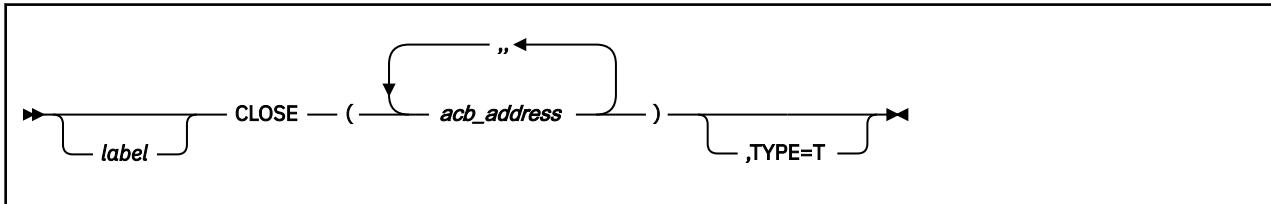
**CHECK**

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'04'	4	Your request was not accepted. The RPL in question is active for another request.
X'08'	8	A logical error occurred.
X'0C'	12	A physical error occurred.
<b>ABEND Code</b>	<b>Meaning</b>	
035	An error occurred in the macro associated with the request. The message preceding the ABEND code explains the problem further.	
03B	An invalid address was detected in a VSAM control block or a VSAM parameter list. This means that your program tried to use an address to which it has no access.	

# CLOSE

---

## Format



## Purpose

Use the CLOSE macro to close a VSAM file that your program has finished with. When a file is no longer needed by your program, the file must be closed. Closing a file involves recording pending updates in the file, logically disconnecting it from the program that was processing it, freeing storage that is no longer needed, updating the catalog with any changes in the attributes of the file, and restoring control blocks to their condition before the file was opened. This discussion of the CLOSE macro deals only with those matters that involve GCS.

## Parameters

### *acb\_address*

Specifies the address of the access method control block (ACB) associated with the file you wish to close.

You can specify the address of more than one, and thereby close more than one file. If you do specify more than one ACB address, be certain to separate each by a comma.

You can write this parameter as an assembler program label or as register (2) through (12). If you specify the address using a register, then be certain that each register in the list is surrounded by a pair of parentheses. And, always be certain that the list itself is surrounded by a pair of parentheses.

### **TYPE=T**

Indicates that you want all closing operations performed on the file in question, except that you do not want your program logically disconnected from the file.

## Usage

1. The CLOSE macro completes any outstanding operations on a file. For example, the CLOSE macro may cause VSAM to write any index or data buffers that have been updated but not yet recorded in the file.
2. The CLOSE macro updates the catalog with any changes made to the attributes of the file, including pointers that mark the end of the file and statistics on its processing. (This does not apply to catalogs that reside on READ ONLY disks.) It restores all pertinent control blocks to their condition before the file was opened. It then completes any outstanding I/O operations that the file may have pending. The CLOSE macro restores the ACB to the status it had before the file was opened and frees the storage that the OPEN macro used to construct VSAM control blocks. If you load records into a file and retrieve records all in the same run, then you must enter a CLOSE macro between these two activities.
3. If an abnormal termination occurs, then GCS will attempt to close the ACB. If GCS is unable to do so, then you should use the Access Method Services VERIFY command to correct the file's catalog information.
4. Under no circumstances will GCS attempt to close ACBs during normal task termination. This is the program's responsibility.

5. The parameters in the CLOSE macro are positional. Therefore, write them in the order indicated in the preceding syntax box and provide a comma for any that you omit.
6. See [Appendix B, “Using VSAM,” on page 517](#).
7. If you have data control blocks (DCBs) that you wish to close, and ACBs, you can specify a combination of both in the same CLOSE macro. GCS is able to distinguish the address of one from the address of the other, if you separate each with a comma. This macro and the one described in [“CLOSE \(BSAM/QSAM\)” on page 382](#) are similar. However, note that neither of these macros, as presented herein, pertains to VTAM.

## Return Codes and ABEND Codes

When this macro completes processing, it passes to the caller a return code in register 15.

Hex Code	Decimal Code	Meaning
X'00'	0	All files were successfully closed.
X'04'	4	At least one file was not closed successfully.
X'08'	8	Either one or more CLOSE routines could not be loaded because insufficient virtual storage was available, or the modules could not be found. Processing cannot continue.

If register 15 contains the return code 4, then you can use the SHOWCB macro to display the ERROR field of each access method control block. See [“SHOWCB” on page 480](#). The following table describes the possible values this field can contain.

Error Code	Meaning
0	No error occurred.
4	The file associated with this ACB is already closed.
136	Insufficient virtual storage was available to execute the CLOSE macro.
144	An irrecoverable I/O error occurred while VSAM was reading or writing a catalog record.
148	An unidentified error occurred while VSAM was searching the catalog.
184	An irrecoverable I/O error occurred while VSAM was completing outstanding I/O requests.
246	Compression Management Services error during CLOSE.
247	Compression Control error during CLOSE.

ABEND Code	Meaning
035	An error occurred in the CLOSE macro. The message preceding the ABEND describes this further.
03B	An invalid address was detected in a VSAM control block or a VSAM parameter list. This means that your program tried to use an address to which it has no access.

# DLVRP

---

## Format



## Purpose

Use the DLVRP macro to delete the resource pool after all the files using the resource pool have been closed.

## Parameters

### TYPE=LSR

Specifies the resource pool is used for Local Shared Resources.

## Usage

When using this macro you must make sure that register 13 contains the address of a 72-byte save area. If you enter the macro from within one of your exit routines (LERAD or SYNAD) you must provide a second 72-byte save area because the original one is still in use by the external VSAM routine.

## Completion Codes

When VSAM returns to the application after a DLVRP request, register 15 contains one of the following completion codes:

Completion Code	Meaning
0	VSAM completed the request successfully.
4	There is not a resource pool to delete.
8	An error was detected while VSAM routines were being loaded.
12	There is at least one open data set using the resource pool.
16	A TYPE other than LSR was specified.

## ENDREQ

---

### Format



### Purpose

Use the ENDREQ macro to cancel a certain VSAM request.

A VSAM request is associated with one of the following macros: CHECK, ENDREQ, ERASE, GET, POINT, and PUT. See “[CHECK](#)” on page 425, “[ERASE](#)” on page 432, “[GET](#)” on page 451, “[POINT](#)” on page 468, or “[PUT](#)” on page 470). You may wish to cancel one such request that you previously made.

This discussion of the ENDREQ macro deals only with those matters that involve GCS.

### Parameters

#### RPL

Specifies the address of the request parameter list (RPL) associated with the VSAM request you wish to cancel.

This is the same request parameter list that you defined through the RPL macro. See “[RPL](#)” on page 472.

You can write this parameter as an assembler program label or as register (2) through (12).

### Usage

1. The ENDREQ macro causes VSAM to end a request — VSAM will forget its position for the specified RPL and will release its associated buffers to another RPL. Therefore, before you enter the ENDREQ macro specifying an RPL for which the ENDREQ macro was previously executed, you must reposition VSAM.
2. Each time you enter the ENDREQ macro, you must provide the system with a 72-byte save area. Be certain that before you enter the instruction you place the address of this save area in register 13.
3. You are limited to as many concurrent active requests as you have specified in the STRNO parameter of the ACB macro. See “[ACB](#)” on page 418. If you want to initiate more requests, then you must start the ENDREQ macro first.
4. If an I/O operation is in progress when you enter the ENDREQ macro, it will complete. This includes operations that are necessary to maintain the integrity of the file.
5. If your request involves a chain of RPLs, then all records specified in the request may not be processed. For example, two RPLs are chained in a PUT request to add two new records to a file. Then, an ENDREQ macro is issued after VSAM started the I/O operation to add the first new record. That operation will be completed. If the operation causes a control-interval split, subsequent I/O operations occur to complete the split and update the index. However, VSAM will then return control to the processing program without adding the second new record.
6. The ENDREQ macro causes VSAM to cancel the position in the file established for that request. It also invalidates data and index buffers to force refreshing of all requests subsequent to the end request.
7. See [Appendix B, “Using VSAM,” on page 517](#).

## Completion Codes, Return Codes, and ABEND Codes

When this macro completes processing, it passes to the caller a completion code in register 15. If register 15 contains 8 or 12, then the specific error is indicated in the FDBK field of the appropriate RPL. This field can be displayed through the SHOWCB or TESTCB macros. See “[SHOWCB](#)” on page 480 or “[TESTCB](#)” on page 490.

<b>Completion Code</b>	<b>Meaning</b>
0	Function completed successfully.
4	The ENDREQ macro could not terminate the request. The specified RPL was active for another request.
8	A logical error occurred.
12	A physical error occurred.
<b>ABEND Code</b>	<b>Meaning</b>
035	An error occurred in the ENDREQ macro. The message preceding the ABEND explains this further.
03B	An invalid address was found in a VSAM control block or a VSAM parameter list. This means your program tried to use an address to which it has no access.

## ERASE

---

### Format



### Purpose

Use the ERASE macro to delete a record from a VSAM file.

This record must be one that you have retrieved through the GET macro with the OPTCD=UPD parameter specified. You can delete records in a key-sequenced file by keyed or addressed access. However, you cannot delete records in an entry sequenced file. You can delete records in a relative-record file by keyed access, but you cannot delete control intervals. See [“GET” on page 451](#).

This discussion of the ERASE macro deals only with those matters that involve GCS.

### Parameters

#### RPL

Specifies the address of the request parameter list (RPL) associated with your ERASE request.

This is the same request parameter list that you defined through the RPL macro. (If necessary, review the entry titled [“RPL” on page 472](#).)

You can write this parameter as an assembler program label or as register (2) through (12).

### Usage

1. Each time you enter the ERASE macro, you must provide the system with a 72-byte save area. Be certain that before you enter the macro you place the address of this save area in register 13.
2. See [Appendix B, “Using VSAM,” on page 517](#).

### Return Codes and ABEND Codes

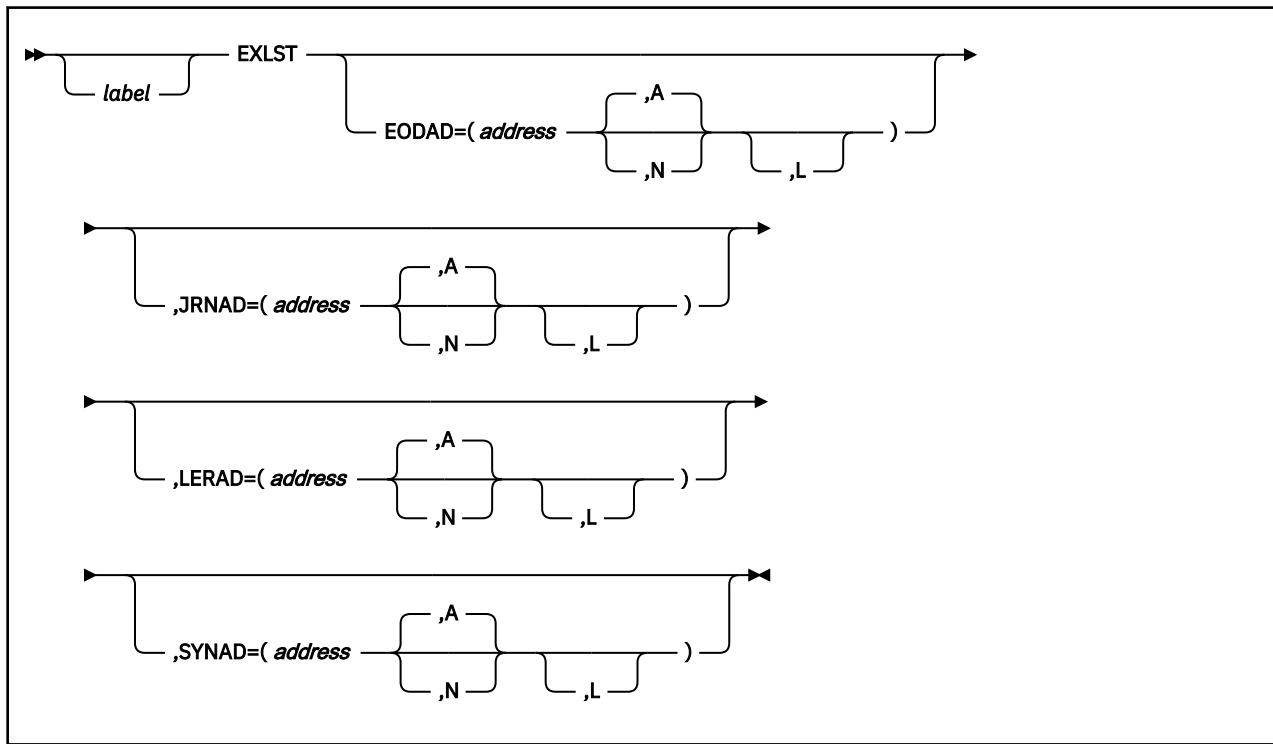
When this macro completes processing, it passes to the caller a return code in register 15. If register 15 contains 8 or 12, then the specific error is indicated in the FDBK field of the appropriate RPL. This field can be displayed through the SHOWCB or TESTCB macros. See [“SHOWCB” on page 480](#) or [“TESTCB” on page 490](#).

Hex Code	Decimal Code	Meaning
X'00'	0	Your request was accepted.
X'04'	4	Your request was not accepted because the RPL you specified in the ERASE macro is already active for another request.
X'08'	8	A logical error occurred.
X'0C'	12	A physical error occurred.

<b>ABEND Code</b>	<b>Meaning</b>
035	An error occurred in the ERASE macro.
03B	An invalid address was found in a VSAM control block or a VSAM parameter list. This means your program tried to use an address to which it has no access.

# EXLST

## Format



## Purpose

Use the EXLST macro to create a list of the addresses of your exit routines.

During VSAM processing, unusual conditions sometimes occur. If you wish, you can supply one or more exit routines to handle such conditions. You can then associate them with one or more access method control blocks (ACBs) that define the characteristics of the VSAM files you plan to process.

This discussion of the EXLST macro deals only with those matters that involve GCS.

## Parameters

### EODAD

Indicates that you are providing an exit routine to handle the END-OF-FILE condition during sequential or skip-sequential access.

### JRNAD

Indicates that you are providing an exit routine to handle journaling.

### LERAD

Indicates that you are providing an exit routine that will analyze logical errors.

### SYNAD

Indicates that you are providing an exit routine that will analyze physical errors.

### address

Specifies the address of the exit routine in question.

You can write this parameter as an assembler program label or as register (2) through (12).

**A**

Indicates that the exit routine in question will be active.

This is the case by default.

**N**

Indicates that the exit routine in question will not be active.

Even if the condition to which this exit routine applies arises, it will not receive control.

**L**

Indicates that the address given in the ADDRESS parameter is an 8-byte field that contains the name of the exit routine in question. It is to be loaded into virtual storage by GCS.

If you omit this parameter, then GCS assumes that the address you specify is the routine's entry point in virtual storage.

## Usage

1. You can create a list of exit routine addresses, but for them to be useful you must specify the address of this list in the EXLST parameter of the ACB or MODCB macro. See [“ACB” on page 418](#) or [“MODCB” on page 453](#).

The address of this list is the same as the address of the EXLST macro in your program, see the label on this macro when you create or modify the access method control block (ACB).

2. When VSAM enters one of your exit routines, the registers contain information that may be helpful in analyzing the situation.
3. The EXLST macro generates an exit list at assembly time. Contrast this with the GENCB macro, which generates an exit list at execution time. See [“GENCB” on page 437](#).
4. You can define no more than 128 exits per GCS virtual machine.
5. See [Appendix B, “Using VSAM,” on page 517](#).

### When VSAM enters an EODAD routine, the registers contain the following:

Register	Contents
0	Unpredictable.
1	The address of the request parameter list that defines the request that occasioned VSAM's reaching the end of the file. The register must contain this address if you return to VSAM.
2 - 13	The same values as when the macro was issued. Register 13, by convention, contains the address of your program's 72-byte save area. This save area cannot be used as a save area by the EODAD routine if it returns control to VSAM.
14	The return address within VSAM.
15	The entry address to the EODAD routine.

### When VSAM enters a JRNAD routine, the registers contain the following:

Register	Contents
0	Unpredictable.
1	The address of a parameter list. For more information on the format of the list, see <i>VSE/VSAM Commands and Macros</i> .
2 - 13	Unpredictable.
14	The return address within VSAM.
15	The entry address in the JRNAD routine.

**When VSAM enters a LERAD routine, the registers contain the following:**

<b>Register</b>	<b>Contents</b>
0	Unpredictable.
1	The address of the request parameter list that contains the feedback field that the routine should examine. The register must contain this address if you return to VSAM.
2 - 13	The same values as when the macro was issued. Register 13, by convention, contains the address of your program's 72-byte save area. This save area cannot be used as a save area by the LERAD routine if it returns control to VSAM.
14	The return address within VSAM.
15	The entry address to the LERAD routine. This register does not contain the logical-error indicator.

**When VSAM enters a SYNAD routine, the registers contain the following:**

<b>Register</b>	<b>Contents</b>
0	Unpredictable.
1	The address of the request parameter list that contains the feedback return code and the address of the message area, if any, that the routine should examine. If you issued a request instruction, then the request macro points to the RPL. If you issued a CLOSE macro, then the RPL was built by VSAM so it could close the file. Register 1 must contain one of these addresses if you return to VSAM.
2 - 13	The same values as when the macro was issued. Register 13, by convention, contains the address of your program's 72-byte save area. This save area cannot be used as a save area by the LERAD routine if it returns control to VSAM.
14	The return address within VSAM.
15	The entry address to the LERAD routine. This register does not contain the physical-error indicator.

**Return Codes and ABEND Codes**

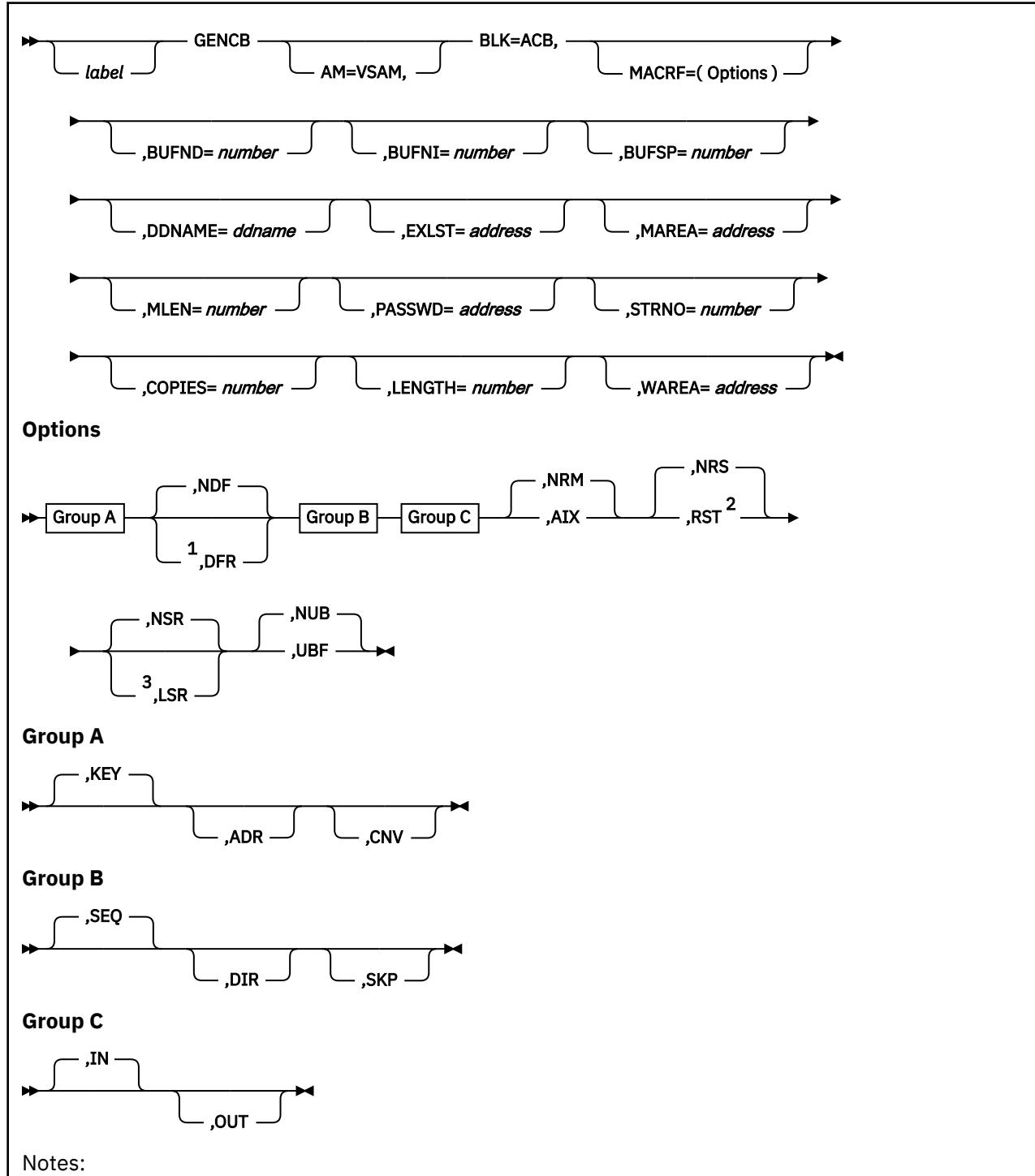
The EXLST macro generates no return codes and no ABEND codes.

## GENCB

The GENCB macro is available in Access Control Block (ACB), Exit List (EXLST) and Request Parameter List (RPL) formats.

### Access Control Block Format

See also [“Exit List Format” on page 442](#) and [“Request Parameter List Format” on page 445](#).



<sup>1</sup> DFR is mutually exclusive with NSR MACRF options.

<sup>2</sup> RST is mutually exclusive with IN MACRF options.

<sup>3</sup> LSR is mutually exclusive with UBF or RST MACRF options.

## Purpose (ACB)

An access method control block (ACB) defines certain characteristics of a file that you intend to process through VSE/VSAM. When the file is opened, other characteristics of the file, which you defined through the DLBL command, are merged with the ACB. For more information on the DLBL command, see “[DLBL](#)” on page 64.

This discussion of the GENCB macro deals only with those matters that involve GCS.

## Parameters (ACB)

### AM=VSAM

Indicates that you are using VSAM to process the file associated with the ACB.

### BLK=ACB

Indicates that you wish to generate an access method control block.

This parameter is required to distinguish this macro from the other two GENCB macros. See “[Exit List Format](#)” on page 442 and “[Request Parameter List Format](#)” on page 445.

### MACRF

Indicates how you intend to process the file.

You must specify all of the types of processing you intend to perform on the file, whether you intend to perform them concurrently or sequentially. The parameters you choose must be valid for the file in question. For example, if you specify keyed access for an entry-sequenced file, then you cannot open that file, or process it.

Check the preceding format box. The processing options are arranged in groups, each with a default value. They are not positional parameters — they can be specified in any order.

### KEY

Indicates access to a key-sequenced or relative record file.

Keys will be relative record numbers used as search arguments, and sequential access will be by key or relative record number.

### ADR

Indicates addressed access to a key-sequenced or entry-sequenced file.

RBAs are used as search arguments, and sequential access is by entry sequence.

### CNV

Indicates access is to the entire contents of a control interval, rather than to an individual record.

### NDF

Indicates that any WRITE macro is deferred for a direct PUT macro.

### DFR

Specifies that physically writing the I/O buffers is deferred when possible.

### SEQ

Indicates sequential access to a key-sequenced, entry-sequenced, or relative record file.

### DIR

Indicates direct access to a key-sequenced, entry-sequenced, or relative record file.

### SKP

Indicates skip-sequential access to a key-sequenced or relative record file.

This is valid only with keyed access in a forward direction.

**IN**

Indicates retrieval of records from key-sequenced, entry-sequenced, or relative record files.

This is not a valid form of processing for an empty file.

**OUT**

Indicates three things:

- Storage of new records in a key-sequenced, entry-sequenced, or relative record file. This is not allowed with addressed access to a key-sequenced file.
- Update of new records in a key-sequenced, entry-sequenced, or relative record file.
- Deletion of records from a key-sequenced or relative record file.

**NRM**

Indicates that the file to be processed is the one specified by the DDNAME parameter.

**AIX**

Indicates that the object to be processed is the alternate index of the path specified by the DDNAME parameter, rather than the base cluster through the alternate index.

**NRS**

Indicates that the file is not reusable.

**RST**

Indicates that the file is reusable.

The OPEN macro resets the file's catalog information to its original status. It resets it to the status it had before the file was open the first time. See [“OPEN” on page 465](#). Also, the high-used RBA is reset to zero.

The file must have been defined with the REUSE attribute for RST to be effective. Although the file is not erased, you can handle it as though it were a new file, and use it as a work file. When the OPEN macro does the reset operation, this parameter is equivalent to the OUT option. DISP=NEW specified on the DLBL command is equivalent to selecting this parameter and will override the NRS parameter.

**NSR**

Indicates that the resources are not shared.

**LSR**

Specifies that the resources are shared. This also indicates a VSAM resource pool will be provided when opening this ACB.

**NUB**

Indicates that VSAM will manage the I/O buffers.

**UBF**

Indicates that the application will manage the I/O buffers.

The work area specified by the RPL or GENCB macros will be, in effect, the I/O buffer. The contents of a control interval are transmitted directly between the work area and DASD. This parameter is valid only when the MACRF=CNV and OPTCD=MVE parameters are specified in the RPL macro. See [“RPL” on page 472](#) and [“Request Parameter List Format” on page 445](#).

**BUFND**

Specifies the number of I/O buffers to be used for transmitting data between virtual and auxiliary storage.

The size of a buffer corresponds to the size of a control interval in the data component. The minimum number you can specify is 1 plus the number specified by the STRNO parameter. If you omit the STRNO parameter, then the value of the BUFND parameter must be at least 2 because the default for the former is 1.

The default for the BUFND parameter is the minimum number required to process your file.

**BUFNI**

Specifies the number of I/O buffers to be used for transmitting the contents of index entries between virtual and auxiliary storage during keyed access.

The size of this buffer corresponds to the size of a control interval in the index. The minimum number you can specify is 1 plus the number specified by the STRNO parameter. If you omit the STRNO parameter, then the value of BUFNI parameter must be at least 2 because the default for the former is 1.

The default for the BUFNI parameter is the minimum number required to process your file.

**BUFSP**

Specifies the maximum number of bytes of virtual storage to be used for the data and index I/O buffers.

This parameter must be at least as large as the buffer size recorded in the catalog entry for your file. If the number you specify for this parameter is too small, then VSAM overrides it and uses the buffer size recorded in the catalog. VSAM, however, does not inform you of this.

If you omit this parameter, then the size of this buffer will be the larger of the following, by default:

- The buffer size specified in the catalog. This buffer size was specified through the BUFFERSPACE parameter in the Access Method Services DEFINE command. If this parameter was omitted when your file was defined, then a default value was assigned to it. This default value, the minimum amount of buffer space allowed by VSAM, is enough to hold two data control intervals and one index control interval.
- The buffer size determined from the BUFND and BUFNI parameters.

You can also specify buffer space through the BUFSP parameter on the DLBL command that identifies your file. This value overrides the BUFSP parameter in the ACB macro. It overrides the BUFFERSPACE parameter in the DEFINE command if the latter is smaller.

If the values you specify for the BUFND, BUFNI, and BUFSP parameters are inconsistent, then VSAM increases the number of buffers to conform with the size of the buffer area. If the value in the BUFSP parameter is greater than the minimum buffer size required to process your file and greater than the values specified in the BUFND and BUFNI parameters, then the extra space is allocated between the data and index buffers as follows:

- If the MACRF parameter specifies direct processing, then the values in the BUFND and BUFNI parameters take effect. Any left-over space is used for index buffers.
- If the MACRF parameter specifies sequential processing, then the values in the BUFND and BUFNI parameters take effect. Space for one additional index buffer is allocated. Any left-over space is used for data buffers. If any left-over space remains that is insufficient to accept another data buffer, then it is used for another index buffer.

If the value in the BUFSP parameter is greater than the minimum required to process your file, but less than those of the BUFND and BUFNI parameters, then enough buffer space will be made available to conform to the latter parameters.

If you provide your own pool of I/O buffers for control interval processing, then the BUFSP, BUFND, and BUFNI parameters have no effect. In such a case, the AREA and AREALEN parameters of the RPL macro determine the size of the user buffer area. See “[RPL](#)” on page 472.

**DDNAME**

Specifies the name of the file you wish to process.

This name corresponds to that specified in the DDNAME parameter of the DLBL command associated with the file. If you omit this parameter, then you can supply it through the MODCB macro. See “[MODCB](#)” on page 453.

This name must be from one to seven characters long.

**EXLST**

Specifies the address of a list of exit routine addresses.

This is the same list that you created through the EXLST or GENCB macro. See “[EXLST](#)” on page 434 or “[Purpose \(EXLST\)](#)” on page 443.

If you used the EXLST macro to create this list, then you can write this parameter as the label on that macro. If you used the GENCB macro, then you can write this parameter as the address that the GENCB macro returned to you in register 1 or as the label associated with an area where you have placed this address.

If you omit this parameter, then GCS assumes that you have supplied no exit routines.

#### **MAREA**

Specifies the address of an area where GCS will place any console messages generated during processing of your file.

This area can be used by you or your exit routines to analyze any errors or problems that may arise.

#### **MLEN**

Specifies the length, in bytes, of the area whose address is given by the MAREA parameter.

The value of this parameter is zero, by default. Its maximum value is 32K.

#### **PASSWD**

Specifies the address of a field that contains the highest level password required for the types of access indicated by the MACRF parameter.

The first byte of the field contains the binary length of the password. Eight bytes is the maximum length. If this byte is 0, it means that you are providing no password.

#### **STRNO**

Specifies the number of requests you will make that will require concurrent file positioning.

A request is defined by a given request parameter list or a chain thereof. If records are written in an empty file, then the value of this parameter is ignored and replaced by the value 1.

If you omit this parameter, then its value is 1, by default.

#### **COPIES**

Specifies the number of copies of the access method control block you want generated.

GCS will generate as many ACBs as you wish. Each will be identical. You can use the MODCB macro to tailor each ACB to the specific file and type of processing you wish. See “[MODCB](#)” on page 453. However, unless you specify otherwise, GCS will generate just one copy.

#### **LENGTH**

Specifies the length of the area you are supplying in virtual storage to hold the ACBs you want to generate. Express this figure in bytes.

#### **WAREA**

Specifies the address of the area you are supplying in virtual storage to accept the ACBs you want to generate.

This area must begin on a fullword boundary.

If you omit this parameter, then the address of an ACB area set up by GCS is returned to you in register 1. GCS returns the length of the area in register 0. To find the length of each ACB, just divide the length of the area supplied by the number of ACBs you specified in the COPIES parameter. Then, to access each ACB in the area, use this quotient as an offset from the address in register 1.

### **Usage (ACB)**

1. The GENCB macro generates an ACB at execution time. Contrast this with the ACB macro, which generates an ACB at assembly time. See “[ACB](#)” on page 418.
2. Each time you enter the GENCB macro, you must provide the system with a 72-byte save area. Before you enter the macro, place the address of this save area in register 13.
3. See [Appendix B, “Using VSAM,” on page 517](#).

4. Appropriate macro MNOTES notify the application programmer of syntax errors in coding the ACB macro.

### Completion Codes, Return Codes, and ABEND Codes (ACB)

When this macro completes execution, it passes to the caller a completion code in register 15.

Completion Code	Meaning
0	Function completed successfully.
4	Function completed unsuccessfully.
8	You attempted to use the execute form of the macro to modify a keyword that is not in the parameter list.
12	The GENCB macro was not executed because an error occurred while a VSAM module was being loaded.

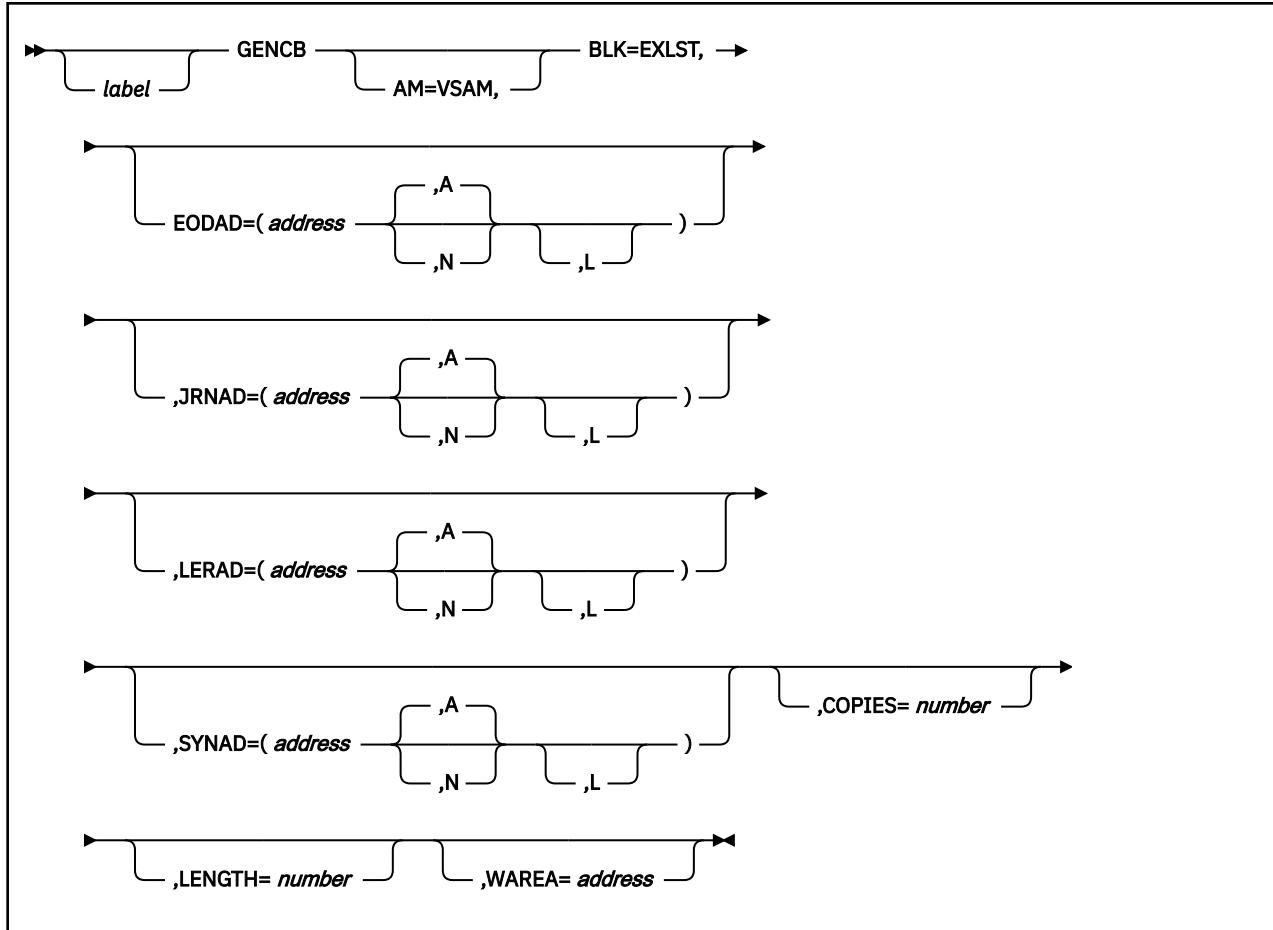
If register 15 contains 0 and if the WAREA parameter was not specified, then register 0 contains the length of the area which GCS builds the ACBs. Register 1 contains the address of this area.

If register 15 contains 4, then register 0 contains a return code, further describing the condition.

Hex Code	Decimal Code	Meaning
X'01'	1	The type of your request is invalid.
X'02'	2	The block type is invalid.
X'03'	3	One of the keyword codes in the parameter list is invalid.
X'08'	8	There is not enough virtual storage to generate the ACB.
X'09'	9	The area you specified in the WAREA parameter is not large enough to hold the ACB.
X'0E'	14	You have specified an invalid combination of options in the MACRF parameter.
X'0F'	15	The storage you specified in the WAREA parameter does not fall on a fullword boundary.

ABEND Code	Meaning
03B	An invalid address was found in a VSAM control block or a VSAM parameter list. This means your program tried to use an address to which it has no access.

### Exit List Format



## Purpose (EXLST)

During VSAM processing, unusual conditions sometimes occur. You can supply one or more exit routines to handle such conditions. You can then associate them with one or more access method control blocks (ACBs) that define the characteristics of the VSAM files you plan to process. See “[ACB](#)” on page 418.

This discussion of the GENCB macro deals only with those matters that involve GCS.

Use the GENCB macro to create a list of the addresses of your exit routines.

## Parameters (EXLST)

### **AM=VSAM**

Indicates that you are using VSAM to process your files.

### **BLK=EXLST**

Indicates that you wish to generate an exit list.

This parameter is required to distinguish this macro from the other two GENCB macros. See “[Access Control Block Format](#)” on page 437 and “[Request Parameter List Format](#)” on page 445.

### **EODAD**

Indicates that you are providing an exit routine to handle the END-OF-FILE condition during sequential or skip sequential access.

### **JRNAD**

Indicates that you are providing an exit routine to handle journaling.

### **LERAD**

Indicates that you are providing an exit routine that will analyze logical errors.

**SYNAD**

Indicates that you are providing an exit routine that will analyze physical errors.

**address**

Specifies the address of the exit routine in question.

You can write this parameter as an assembler program label or as register (2) through (12).

**A**

Indicates that the exit routine in question will be active.

This is the case by default.

**N**

Indicates that the exit routine in question will not be active.

Even if the condition which this exit routine applies arises, it will not receive control.

**L**

Indicates that the address given in the ADDRESS parameter is the address of an 8-byte field that contains the name of the exit routine in question. It is to be loaded into virtual storage by GCS.

If you omit this parameter, then GCS assumes that the address you specify is the routine's entry point in virtual storage.

**COPIES**

Specifies the number of copies of the exit list you want generated.

GCS will generate as many exit lists as you wish. Each will be identical. You can use the MODCB macro to modify the addresses in any of the exit lists. See "[MODCB](#)" on page 453. However, unless you specify otherwise, GCS will generate only one copy.

**LENGTH**

Specifies the length of the area you are supplying in virtual storage to hold the exit lists you want to generate. Express this figure in bytes.

**WAREA**

Specifies the address of the area you are supplying in virtual storage to accept the exit lists you want to generate.

This area must begin on a fullword boundary.

If you omit this parameter, then the address of an exit list area set up by GCS is returned to you in register 1. GCS returns the length of the area in register 0. To find the length of each exit list, just divide the length of the area supplied by the number of lists you specified in the COPIES parameter. Then, to access each list in the area, use this quotient as an offset from the address in register 1.

**Usage (EXLST)**

1. Note that the GENCB macro generates an exit list at execution time. Contrast this with the EXLST macro which generates an exit list at assembly time. See "[EXLST](#)" on page 434.
2. Each time you enter the GENCB macro, you must provide the system with a 72-byte save area. Before you enter the instruction, place the address of this save area in register 13.
3. See [Appendix B, "Using VSAM," on page 517](#).

**Completion Codes, Return Codes, and ABEND Codes (EXLST)**

When this macro completes execution, it passes to the caller a completion code in register 15.

Completion Code	Meaning
0	Function completed successfully.
4	Function completed unsuccessfully.

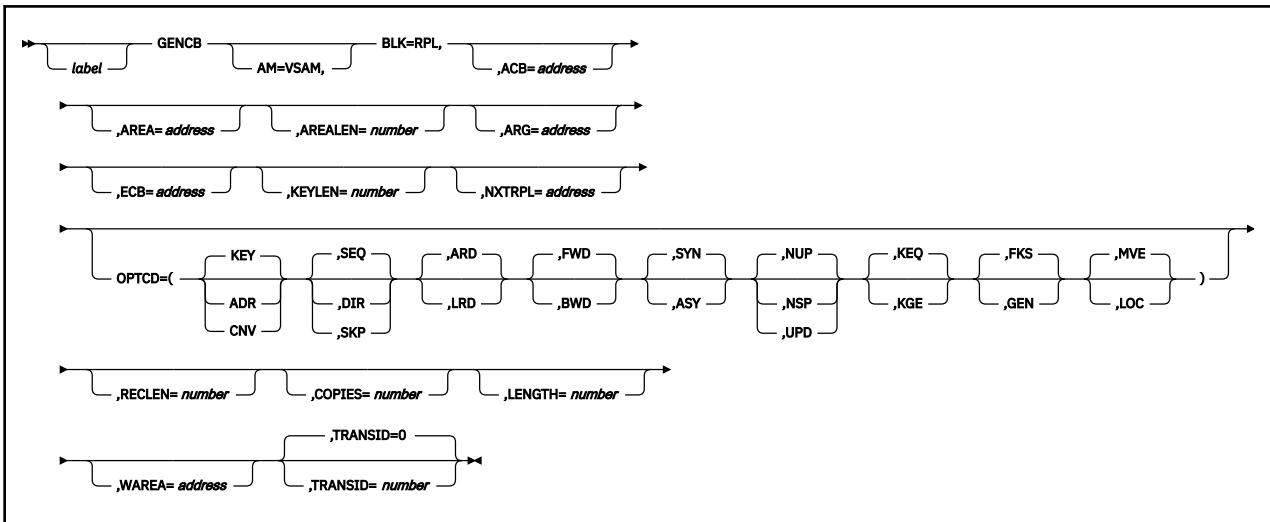
Completion Code	Meaning
8	You attempted to use the execute form of the macro to modify a keyword that is not in the parameter list.
12	The GENCB macro was not executed because an error occurred while the module was being loaded.

If register 15 contains 0 and if the WAREA parameter was not specified, then register 0 contains the length of the area in which GCS builds the ACBs. Furthermore, register 1 contains the address of this area. If register 15 contains 4, then register 0 contains a return code, further describing the condition.

Hex Code	Decimal Code	Meaning
X'01'	1	The type of your request is invalid.
X'02'	2	You selected an access method control block. This is invalid.
X'03'	3	One of the keyword codes in the parameter list is invalid.
X'08'	8	There is not enough virtual storage to generate the exit list.
X'09'	9	The area you specified in the WAREA parameter is not large enough to generate the exit list.
X'0A'	10	You specified an exit without giving an address.
X'0F'	15	The storage you specified in the WAREA parameter does not fall on a fullword boundary, as it must.

ABEND Code	Meaning
03B	An invalid address was found in a VSAM control block or a VSAM parameter list. This means your program tried to use an address to which it has no access.

## Request Parameter List Format



## Purpose (RPL)

All VSAM functions require that you set up a request parameter list (RPL) that describes the characteristics of your request. These VSAM functions are associated with the following macros: CHECK, ENDREQ, ERASE, GET, POINT, and PUT. See “[CHECK](#)” on page 425, “[ENDREQ](#)” on page 430, “[ERASE](#)” on page 432, “[GET](#)” on page 451, “[POINT](#)” on page 468, or “[PUT](#)” on page 470.

This discussion of the GENCB macro deals only with those matters that involve GCS.

## Parameters (RPL)

### **AM=VSAM**

Indicates that you are using VSAM to process the file associated with the ACB.

### **BLK=RPL**

Indicates that you wish to generate a request parameter list.

This parameter is required to distinguish it from the other two GENCB macros. See “[Access Control Block Format](#)” on page 437 and “[Exit List Format](#)” on page 442.

### **ACB**

Specifies the address of the access method control block (ACB) associated with the file you are processing.

If you created the access method control block through the ACB macro, you can write this parameter as the assembler program label on that macro. If no ACB associated with your file exists, then you must create one through another GENCB macro before issuing this GENCB macro. See “[Access Control Block Format](#)” on page 437.

### **AREA**

Specifies one of two things:

- If you select the OPTCD=MVE parameter, then the AREA parameter specifies the address of a work area to which a data record is moved to be processed and from which it is moved after processing.
- If you select the OPTCD=LOC parameter, then the AREA parameter will specify the address of a work area. The address of the I/O buffer in which you process your file will be placed in this work area (GET only).

### **AREALEN**

Specifies the length, in bytes, of the work area whose address you specified in the AREA parameter.

If you selected the OPTCD=MVE parameter, then this length must be no less than the size of a data record. For variable-length records, you must allow for the largest record in the file.

If you selected the OPTCD=LOC parameter, then you must specify a length of 4 bytes to accept the address of the I/O buffer where you will process each record.

### **ARG**

Specifies the address of a field that contains the search argument for one of the following:

- Direct or skip sequential retrieval (GET).
- Sequential positioning (POINT).
- Direct or skip sequential storage (PUT) for a relative record file.

For keyed access (OPTCD=KEY), the search argument may be a

- Full key (OPTCD=FKS).
- Generic key (OPTCD=GEN). Here, you must also specify its size through the KEYLEN parameter.
- Relative record number (which is treated as a key).

For addressed access (OPTCD=ADR), the search argument is always an RBA. To determine the RBA of a record where you have gained access sequentially or directly by key, you can enter the SHOWCB macro. See “[SHOWCB](#)” on page 480.

For control interval access with user buffering and a user supplied RBA, the record is written only to this RBA if positioning is not established by a previous request.

When records are inserted into a key sequenced file, either sequentially or directly, VSAM obtains the key from the record itself. When the records are inserted sequentially into a relative record file, VSAM returns the assigned relative record number in the ARG field.

#### **ECB**

Specifies the address of the event control block associated with the VSAM request you will make.

#### **KEYLEN**

Specifies the length, in bytes, of the generic key that you are using as a search argument.

You specify the search argument in the ARG parameter. However, you must specify its length when it is a generic key.

You can write this parameter as any number from 1 to 255.

#### **NXTRPL**

Specifies the address of the next request parameter list in the chain.

Omit this parameter from the RPL macro that generates the last RPL in the chain. When you enter a request that is defined by a chain of RPLs, specify the address of the first RPL in the chain in the instruction associated with the request.

#### **OPTCD**

Indicates the options that will govern the request defined by the request parameter list you are creating.

Carefully check the preceding format box. Note that the parameters are arranged in groups, each with a value that will be assumed by default should you forget to specify from that group. Because they are not positional parameters, they can be specified in any order.

#### **KEY**

Indicates access to a key-sequenced or relative record file.

#### **ADR**

Indicates addressed access to a key-sequenced or entry-sequenced file.

#### **CNV**

Indicates access will be to the entire contents of a control interval, rather than to an individual record.

#### **DIR**

Indicates direct processing.

#### **SEQ**

Indicates sequential processing.

#### **SKP**

Indicates skip-sequential processing.

This is valid only with keyed access.

#### **ARD**

Indicates that the user's argument determines the record to be located, retrieved, or stored.

#### **LRD**

Indicates that the last record in the file will be located or retrieved.

If you choose this parameter, then you must also choose the BWD parameter.

#### **FWD**

Indicates that processing is to go through the file in a forward direction.

#### **BWD**

Indicates that processing is to go through the file in a backward direction for keyed or addressed access, and for sequential or direct processing.

#### **SYN**

Specifies that you want your file processed synchronously.

This means that control will return to your program only after the request associated with the RPL you are creating has been carried out.

**ASY**

Specifies that you want your file processed asynchronously.

This means that when the request associated with the RPL you are creating is scheduled, control will return to your program so it can continue processing. Meanwhile, your request is being carried out.

Remember that asynchronous processing is merely simulated by GCS. Disk I/O in GCS is always synchronous. Even so, you must enter the CHECK macro to obtain the results of the operation. See "["CHECK" on page 425](#)".

**NUP**

Indicates that any record retrieved will not be updated or deleted. Moreover, any record that is stored is a new record.

On direct access requests, GCS does not remember the record's position.

**NSP**

Indicates that, for direct processing only, your request is not for update. VSAM will be positioned at the next record for subsequent sequential processing.

**UPD**

Indicates that any record retrieved can be updated or deleted.

**KEQ**

Indicates that the key you provide as a search argument must equal the key of the record.

**KGE**

Indicates that if the key you specify as a search argument does not equal a certain record, then the request will affect the record with the next highest key.

**FKS**

Indicates that you are providing a full key as a search argument.

**GEN**

Indicates that you are providing a generic key as a search argument.

If you select this parameter, then you must also specify the length of the generic key in the KEYLEN parameter.

**MVE**

Indicates that, during retrieval, the record will be moved to a work area for processing. For storage, it will be moved from the work area to VSAM's I/O buffer.

**LOC**

Indicates that, during retrieval, the record will be put in VSAM's I/O buffer to be processed.

**RECLEN**

Specifies the length, in bytes, of a record that is to be stored.

If you intend to enter the PUT macro, then this parameter is required. If you enter a GET macro, then the length of the record involved is placed in the RPL field associated with this parameter. This is for the benefit of any subsequent update or store requests.

**COPIES**

Specifies the number of copies of the request parameter list you want to generate.

GCS will generate as many RPLs as you wish. Each will be identical. You can use the MODCB macro to tailor each RPL to the specific file and type of processing you wish. See "["MODCB" on page 453](#)".

Unless you specify otherwise, GCS will generate only one copy.

**LENGTH**

Specifies the length of the area you are supplying in virtual storage to hold the RPLs you want to generate. Express this figure in bytes.

**WAREA**

Specifies the address of the area you are supplying in virtual storage to accept the RPLs you want to generate.

This area must begin on a fullword boundary.

If you omit this parameter, then the address of an RPL area set up by GCS is returned to you in register 1. GCS returns the length of the area in register 0. To find the length of each RPL, just divide the length of the area supplied by the number of RPLs you specified in the COPIES parameter. Then, to access each RPL in the area, use this quotient as an offset from the address in register 1.

**TRANSID**

Specifies a number from 0 to 31 when BLK=RPL.

**Number****Description****0**

Default value. Indicates that the request defined by this RPL is not associated with other requests.

**1-31**

Relates the requests defined by this RPL to the requests defined by other RPLs with the same TRANSID value.

**Usage**

1. The GENCB macro generates an RPL at execution time. Contrast this with the RPL macro, which generates an RPL at assembly time. See “[RPL](#)” on page 472.
2. Each time you enter the GENCB macro, you must provide the system with a 72-byte save area. Before you enter the instruction, place the address of this save area in register 13.
3. See [Appendix B, “Using VSAM,”](#) on page 517.

**Completion Codes, Return Codes, and ABEND Codes (RPL)**

When this macro completes execution, it passes to the caller a completion code in register 15.

<b>Completion Code</b>	<b>Meaning</b>
0	Function completed successfully.
4	Function completed unsuccessfully.
8	You attempted to use the execute form of the macro to modify a keyword that is not in the parameter list.
12	The GENCB macro was not executed because an error occurred while a VSAM module was being loaded.

If register 15 contains 0 and if the WAREA parameter were not specified, then register 0 contains the length of the area which GCS builds the RPLs. Furthermore, register 1 contains the address of this area.

If register 15 contains 4, then register 0 contains a return code, further describing the condition.

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'01'	1	The type of your request is invalid.
X'02'	2	You selected a request parameter list. This is invalid.
X'03'	3	One of the keyword codes in the parameter list is invalid.
X'08'	8	There is not enough virtual storage to generate the RPL.

Hex Code	Decimal Code	Meaning
X'09'	9	The area you specified in the WAREA parameter is not large enough to generate the RPL.
X'0F'	15	The storage you specified in the WAREA parameter does not fall on a fullword boundary, as it should.
ABEND Code	Meaning	
03B	An invalid address was found in a VSAM control block or a VSAM parameter list. This means that your program tried to use an address to which it has no access.	

# GET

---

## Format



## Purpose

Use the GET macro to retrieve a record from a VSAM file and place it in either an I/O buffer or a work area. This discussion of the GET macro deals only with those matters that involve GCS.

## Parameters

### RPL

Specifies the address of the request parameter list (RPL) associated with your GET request. This is the same request parameter list that you defined through the RPL macro. See “[RPL](#)” on page 472.

You can write this parameter as an assembler program label or as register (1) through (12).

## Usage

1. Each time you enter the GET macro, you must provide the system with a 72-byte save area. Before you enter the macro, place the address of this save area in register 13.
2. See [Appendix B, “Using VSAM,” on page 517](#).

## Return Codes and ABEND Codes

When this macro completes processing, it passes to the caller a return code in register 15. If register 15 contains 8 or 12, then the specific error is indicated in the FDBK field of the appropriate RPL. This field can be displayed through the SHOWCB or TESTCB macros. See “[SHOWCB](#)” on page 480 or “[TESTCB](#)” on page 490.

Hex Code	Decimal Code	Meaning
X'00'	0	Your request was accepted.
X'04'	4	Your request was not accepted because the RPL you specified in the GET macro is already active for another request.
X'08'	8	A logical error occurred.
X'0C'	12	A physical error occurred.

ABEND Code	Meaning
035	An error occurred in the GET macro. The message preceding the ABEND describes this further.
03B	An invalid address was found in a VSAM control block or a VSAM parameter list. This means that your program tried to use an address to which it has no access.

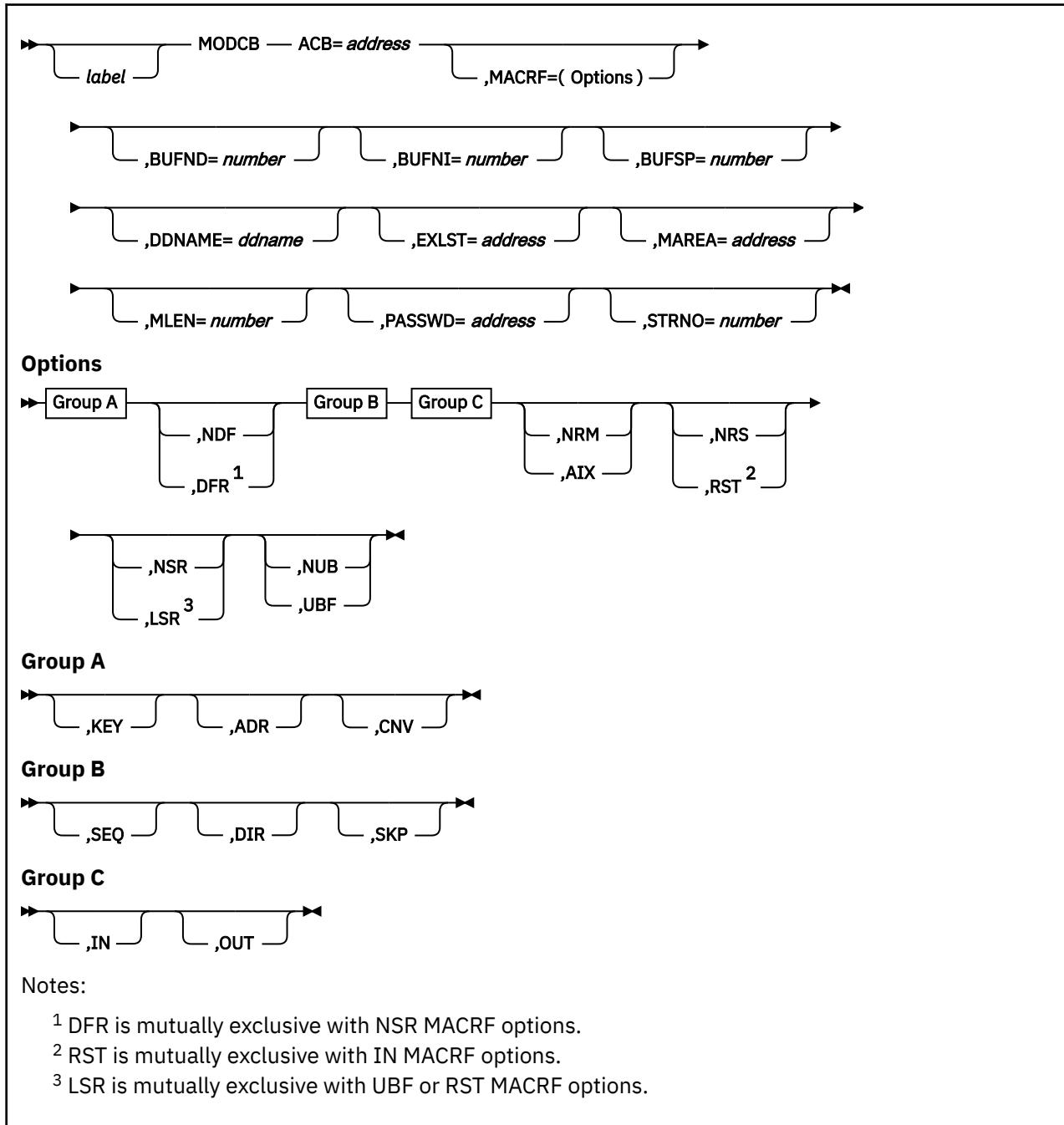
<b>ABEND Code</b>	<b>Meaning</b>
03B	You specified a TYPE parameter of CHK or DRBA, and those parameters are not supported.

## MODCB

The MODCB macro is available in Access Control Block (ACB), Exit List (EXLST) and Request Parameter List (RPL) formats.

### Access Control Block Format

See also “[Exit List Format](#)” on page 458 and “[Request Parameter List Format](#)” on page 460.



### Purpose (ACB)

An access method control block (ACB) defines certain characteristics of a file that you intend to process through VSAM. When the file is opened, other characteristics of the file that you defined through the DLBL

command are merged with the ACB to complete the picture. For more information, see “ACB” on page 418 and “DLBL” on page 64.

This discussion of the MODCB macro deals only with those matters that involve GCS.

## **Parameters (ACB)**

### **ACB**

Specifies the address of the access method control block whose contents you want to modify.

### **MACRF**

Indicates how you intend to process the file.

You must specify all of the types of processing you intend to perform on the file, whether you intend to perform them concurrently or alternately. Moreover, the parameters you choose must be valid for the file in question. For example, if you specify keyed access for an entry-sequenced file, then you cannot open that file.

Check the preceding format box. The processing options are arranged in groups. They are not positional parameters, they can be specified in any order.

### **ADR**

Indicates addressed access to a key-sequenced or entry-sequenced file.

RBAs will be used as search arguments, and sequential access is by entry sequence.

### **CNV**

Indicates access will be to the entire contents of a control interval, rather than to an individual record.

### **KEY**

Indicates access to a key-sequenced or relative record file.

Keys will be relative record numbers used as search arguments, and sequential access will be by key or relative record number.

### **NDF**

Indicates that any WRITE macro will not be deferred for a direct PUT macro.

### **DFR**

Specifies that physically writing the I/O buffers is deferred when possible.

### **DIR**

Indicates direct access to a key-sequenced, entry-sequenced, or relative record file.

### **SEQ**

Indicates sequential access to a key-sequenced, entry-sequenced, or relative record file.

### **SKP**

Indicates skip-sequential access to a key-sequenced or relative record file.

This is valid only with keyed access in a forward direction.

### **IN**

Indicates retrieval of records from key-sequenced, entry-sequenced, or relative record files.

This is not a valid form of processing for an empty file.

### **OUT**

Indicates three things:

- Storage of new records in a key-sequenced, entry-sequenced, or relative record file. This is not allowed with addressed access to a key-sequenced file.
- Update of new records in a key-sequenced, entry-sequenced, or relative record file.
- Deletion of records from a key-sequenced or relative record file.

**NRM**

Indicates that the file to be processed is the one specified by the DDNAME parameter.

**AIX**

Indicates that the object to be processed is the alternate index of the path specified by the DDNAME parameter, rather than the base cluster through the alternate index.

**NRS**

Indicates that the file is not reusable.

**RST**

Indicates that the file is reusable.

The OPEN macro resets the file's catalog information to its original status – it resets it to the status it had before the file was open the first time. See “[OPEN](#)” on page 465. Also, the high-used RBA is reset to zero.

The file must have been defined with the REUSE attribute for RST to be effective. Although the file is not erased, you can handle it as though it were a new file, and use it as a work file. When the OPEN macro carries out the reset operation, this parameter is equivalent to the OUT option. DISP=NEW specified on the DLBL command is equivalent to selecting this parameter, and will override the NRS parameter.

**NSR**

Indicates that the resources are not shared.

**LSR**

Specifies that the resources are shared. This also indicates a VSAM resource pool will be provided opening this ACB.

**NUB**

Indicates that VSAM will manage the I/O buffers.

**UBF**

Indicates that the application will manage the I/O buffers.

The work area specified by the RPL or GENCB macros will be, in effect, the I/O buffer. The contents of a control interval is transmitted directly between the work area and DASD. This parameter is valid only when the MACRF=CNV and OPTCD=MVE parameters are specified in the RPL macro. See “[RPL](#)” on page 472 and “[GENCB](#)” on page 437.

**BUFND**

Specifies the number of I/O buffers to be used for transmitting data between virtual and auxiliary storage.

The size of a buffer corresponds to the size of a control interval in the data component. The minimum number you can specify is 1 plus the number specified by the STRNO parameter.

**BUFNI**

Specifies the number of I/O buffers to be used for transmitting the contents of index entries between virtual and auxiliary storage during keyed access.

The size of this buffer corresponds to the size of a control interval in the index. The minimum number you can specify is 1 plus the number specified by the STRNO parameter.

The default for the BUFNI parameter is the minimum number required to process your file.

**BUFSP**

Specifies the maximum number of bytes of virtual storage to be used for the data and index I/O buffers.

This parameter must be at least as large as the buffer size recorded in the catalog entry for your file. If the number you specify for this parameter is too small, then VSAM overrides it and uses the buffer size recorded in the catalog. VSAM, however, does not inform you of this.

If you omit this parameter, then the size of this buffer will be the largest of the following, by default:

- The buffer size specified in the catalog.

This buffer size was specified through the **BUFFERSPACE** parameter in the Access Method Services **DEFINE** command. If this parameter were omitted when your file was defined, then a default value was assigned to it. This default value, the minimum amount of buffer space allowed by VSAM, is enough to hold two data control intervals and one index control interval.

- Or, the buffer size determined from the **BUFND** and **BUFNI** parameters.

You can also specify buffer space through the **BUFSP** parameter on the **DLBL** command that identifies your file. This value overrides the **BUFSP** parameter in the **ACB** macro. It overrides the **BUFFERSPACE** parameter in the **DEFINE** command if the latter is smaller.

If the values you specify for the **BUFND**, **BUFNI**, and **BUFSP** parameters are inconsistent, then VSAM increases the number of buffers to conform with the size of the buffer area. If the value in the **BUFSP** parameter is greater than the minimum buffer size required to process your file and greater than the values specified in the **BUFND** and **BUFNI** parameters, then the extra space is allocated between the data and index buffers as follows:

- If the **MACRF** parameter specifies direct processing, then the values in the **BUFND** and **BUFNI** parameters take effect. Any left-over space is used for index buffers.
- If the **MACRF** parameter specifies sequential processing, then the values in the **BUFND** and **BUFNI** parameters take effect. Space for one additional index buffer is allocated. Any left-over space is used for data buffers. If any left-over space remains that is insufficient to accept another data buffer, then it is used for another index buffer.

If the value in the **BUFSP** parameter is greater than the minimum required to process your file, but less than those of the **BUFND** and **BUFNI** parameters, then enough buffer space will be made available to conform to the latter parameters.

If you provide your own pool of I/O buffers for control interval processing, then the **BUFSP**, **BUFND**, and **BUFNI** parameters have no effect. In such a case, the **AREA** and **AREALEN** parameters of the **RPL** macro determine the size of the user buffer area. See “[RPL](#)” on page 472.

**DDNAME**

Specifies the name of the file you wish to process.

This name corresponds to that specified in the **DDNAME** parameter of the **DLBL** command associated with the file. If you omit this parameter, then you can supply it through the **MODCB** macro.

This name must be from one to seven characters long.

**EXLST**

Specifies the address of a list of exit routine addresses.

This is the same list that you created through the **EXLST** or **GENCB** macro. See “[EXLST](#)” on page 434 or “[GENCB](#)” on page 437.

If you used the **EXLST** macro to create this list, then you can write this parameter as the label on that instruction. If you used the **GENCB** macro, then you can write this parameter as the address that the **GENCB** macro returned to you in register 1 or as the label associated with an area into which you have placed this address.

If you omit this parameter, then GCS assumes that you have supplied no exit routines.

**MAREA**

Specifies the address of an area where GCS will place any console messages generated during processing of your file.

This area can be used by you or your exit routines to analyze any errors or problems that may arise.

**MLEN**

Specifies the length, in bytes, of the area whose address is given by the **MAREA** parameter.

The minimum value of this parameter is 0 and the maximum value is 32K.

**PASSWD**

Specifies the address of a field that contains the highest level password required for the type(s) of access indicated by the MACRF parameter.

The first byte of the field contains the binary length of the password. Eight bytes is the maximum length. If this byte is 0, it means that you are providing no password.

If the file is password protected, and you provide none, then VSAM will prompt you for the password when it opens the file.

**STRNO**

Specifies the number of requests you will make that will require concurrent file positioning.

A request is defined by a given request parameter list or a chain thereof. If records are written in an empty file, then the value of this parameter is ignored and replaced by the value 1.

**Usage (ACB)**

1. You can add or modify any preceding parameter listed. However, be certain that the additions or modifications you make are consistent and non-conflicting. If you assign a value to a parameter and that new value conflicts or is inconsistent with another value, then the new value replaces the old. For example, if the ACB now specifies the MACRF=UBF parameter, and you specify the MACRF=NUB parameter in the MODCB macro, then NUB replaces UBF.
2. You must never try to modify the ACB of a file that is already open. If you do, it is an error. If you must modify an ACB for a file that is already open, then close the file first.
3. Each time you enter the MODCB macro, you must provide the system with a 72-byte save area. Before you enter the instruction place the address of this save area in register 13.
4. See [Appendix B, “Using VSAM,” on page 517](#).

**Completion Codes, Return Codes, and ABEND Codes (ACB)**

When this macro completes execution, it passes to the caller a completion code in register 15.

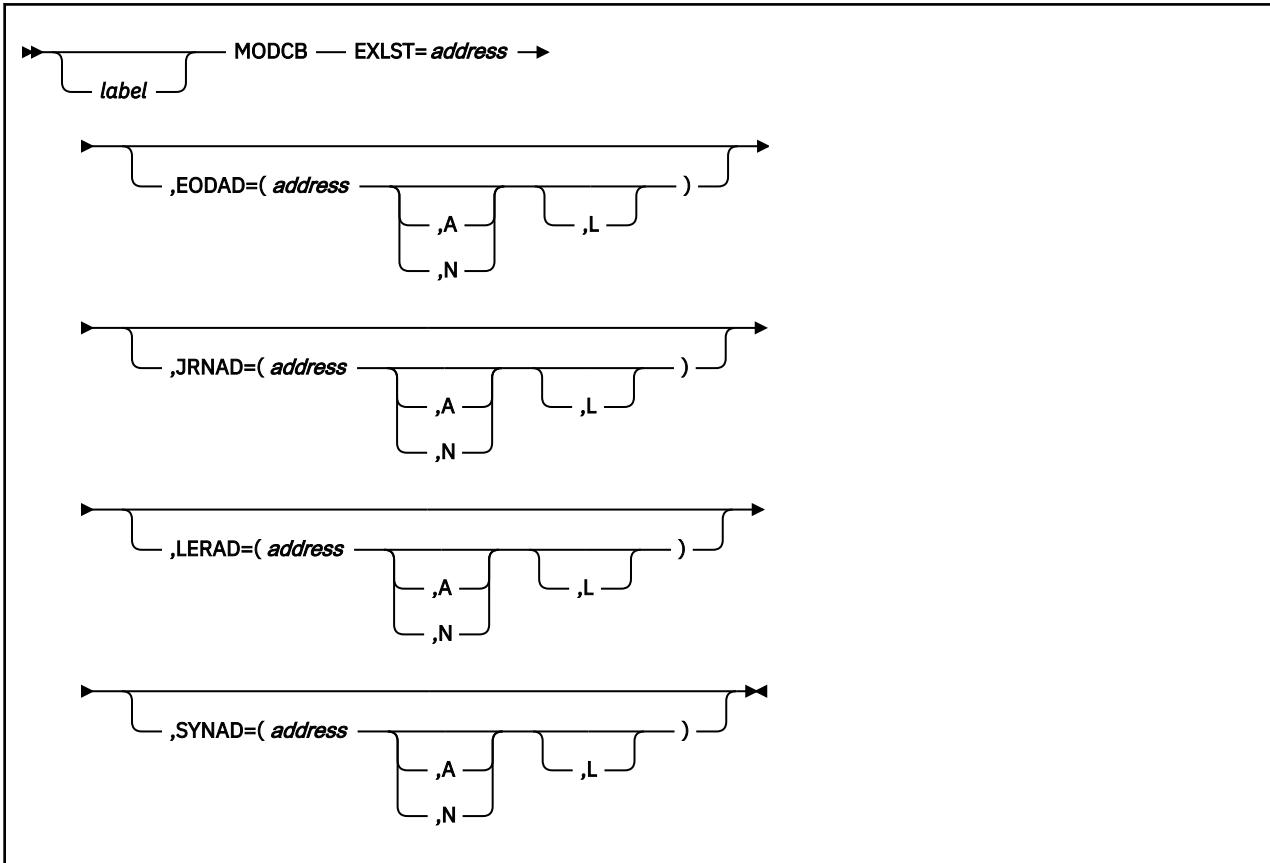
<b>Completion Code</b>	<b>Meaning</b>
0	Function completed successfully.
4	Function completed unsuccessfully.
8	You attempted to use the execute form of this macro to modify a keyword that is not in the parameter list.
12	The MODCB macro was not executed because an error occurred while a VSAM module was being loaded.

When register 15 contains 4, then register 0 contains one of the following return codes.

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'01'	1	The type of request was invalid.
X'02'	2	The block type was invalid.
X'03'	3	One of the keywords in the parameter list is invalid.
X'04'	4	The block at the address you specified was not of the type you indicated.
X'0C'	12	The file associated with the ACB in question is open. It cannot be modified.

Hex Code	Decimal Code	Meaning
X'0E'	14	You specified an incompatible set of parameters for MACRF.
X'10'	16	You specified an invalid control block address in the ACB parameter.
ABEND Code	Meaning	
03B	An invalid address was found in a VSAM control block or a VSAM parameter list. This means that your program tried to use an address to which it has no access.	

## Exit List Format



## Purpose (EXLST)

During VSAM processing, unusual conditions sometimes occur. You can supply one or more exit routines to handle such conditions. You can then associate them with one or more access method control blocks (ACBs) that define the characteristics of the VSAM files you plan to process. See “[Access Control Block Format](#)” on page 453.

This discussion of the MODCB macro deals only with those matters that involve GCS.

Use the MODCB macro at execution time to modify a previously created list that contains the addresses of your exit routines.

## Parameters (EXLST)

**EXLST**

Specifies the address of the list of exit routine addresses that you wish to modify.

**EODAD**

Indicates that you are modifying the address of the exit routine that will handle the END-OF-FILE condition during sequential access.

**JRNAD**

Indicates that you are modifying the address of the exit routine that will handle journaling.

**LERAD**

Indicates that you are modifying the address of the exit routine that will analyze logical errors.

**SYNAD**

Indicates that you are modifying the address of the exit routine that will analyze physical errors.

**address**

Specifies the new address of the exit routine in question.

You can write this parameter as an assembler program label or as register (2) through (12).

**A**

Indicates that the exit routine in question will be active.

**N**

Indicates that the exit routine in question will not be active.

Even if the condition which this exit routine applies arises, it will not receive control.

**L**

Indicates that the address given in the ADDRESS parameter is the address of an 8-byte field that contains the name of the exit routine in question. It is to be loaded into virtual storage by GCS.

If you omit this parameter, then GCS assumes that the address you specify in the ADDRESS parameter is the routine's entry point in virtual storage.

**Usage (EXLST)**

1. It does not matter whether the file whose exit list you are trying to modify is opened or closed. You can enter the MODCB macro in either case.
2. The exit list you want to modify is a certain length. You cannot make any modification to the list that would change its length. For example, if there are already three addresses in the list, you cannot add a fourth. You can, however, modify one of the existing three addresses.

Remember also, that exit list addresses are stored in the exit list control block in the following order: EODAD, SYNAD, LERAD, JRNAD. Given this and the fact that you cannot lengthen an existing exit address list, you must be very careful how you modify it. For example, if your original exit list contained only an address for the LERAD parameter, then you could add addresses for the EODAD and SYNAD parameters. But, to add one for the JRNAD parameter would increase the length of the list and is an error.

3. Each time you enter the MODCB macro, you must provide the system with a 72-byte save area. Before you enter the instruction, place the address of this save area in register 13.
4. See [Appendix B, "Using VSAM," on page 517](#).

**Completion Codes, Return Codes, and ABEND Codes (EXLST)**

When this macro completes execution, it passes to the caller a completion code in register 15.

Completion Code	Meaning
0	Function completed successfully.
4	Function completed unsuccessfully.

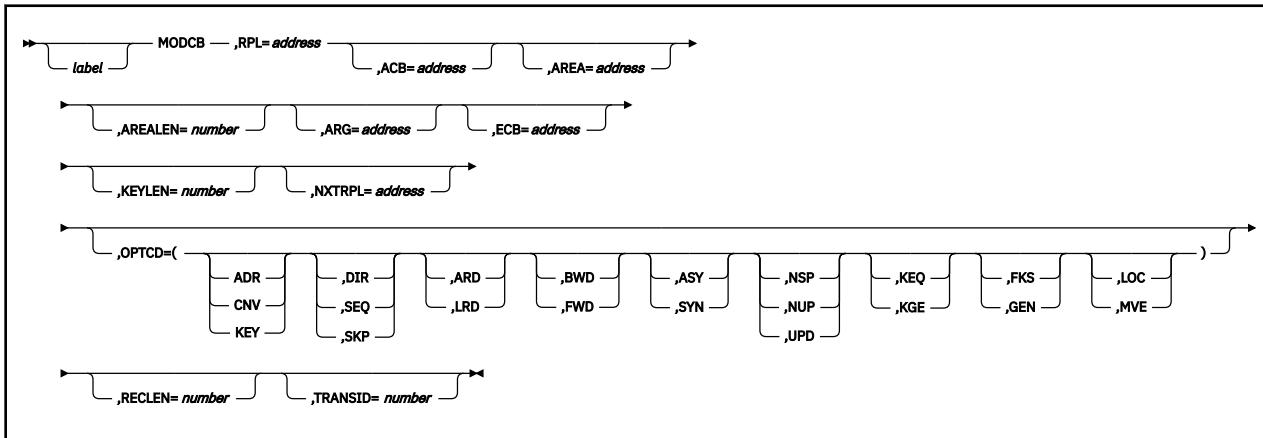
Completion Code	Meaning
8	You attempted to use the execute form of this macro to modify a keyword that is not in the parameter list.
12	The MODCB macro was not executed because an error occurred while a VSAM module was being loaded.

When register 15 contains 4, then register 0 contains one of the following return codes.

Hex Code	Decimal Code	Meaning
X'01'	1	The type of request was invalid.
X'02'	2	The block type was invalid.
X'03'	3	One of the keywords in the parameter list is invalid.
X'04'	4	The block at the address you specified was not the type you indicated.
X'07'	7	Either the exit list is not large enough to accept your modification or the exit entry you tried to modify was not in the list at all.
X'0A'	10	You failed to specify an address for one of your exit routines. Also, you must specify either the A or N parameter.
X'0D'	13	You attempted to activate an exit, but did not provide an address for it.
X'10'	16	You specified an invalid control block address in the EXLST parameter.

ABEND Code	Meaning
03A	The number of exits defined in the system has reached the maximum of 128.
03B	An invalid address was found in a VSAM control block or a VSAM parameter list. This means that your program tried to use an address to which it has no access.

## Request Parameter List Format



## Purpose (RPL)

All VSAM functions require that you set up a request parameter list (RPL) that describes the characteristics of your request. These VSAM functions are associated with the following macros: CHECK, ENDREQ, ERASE, GET, POINT, and PUT. See “[CHECK](#)” on page 425, “[ENDREQ](#)” on page 430, “[ERASE](#)” on page 432, “[GET](#)” on page 451, “[POINT](#)” on page 468, or “[PUT](#)” on page 470.

This discussion of the MODCB macro deals only with those matters that involve GCS.

## Parameters (RPL)

### RPL

Specifies the address of the request parameter list whose fields you want to modify.

You cannot modify the fields of any RPL that is active — an RPL that defines a request that has been issued, but is not yet completed. To confirm whether an active request is complete, enter the CHECK macro. To cancel an active request, enter the ENDREQ macro. See “[CHECK](#)” on page 425 and “[ENDREQ](#)” on page 430.

### ACB

Specifies the address of the access method control block (ACB) associated with the file you are processing.

If you created the access method control block through the ACB macro, you can write this parameter as the assembler program label on that instruction. If no ACB associated with your file exists, then you must create one through the ACB or GENCB macro before issuing the RPL macro. If necessary, review the entry titled “[ACB](#)” on page 418 or “[GENCB](#)” on page 437.

### AREA

Specifies one of two things:

- If you select the OPTCD=MVE parameter, then the AREA parameter specifies the address of a work area to which a data record is moved to be processed and from which it is moved after processing.
- If you select the OPTCD=LOC parameter, then the AREA parameter will specify the address of a work area. The address of the I/O buffer in which you process your file will be placed in this work area (GET only).

### AREALEN

Specifies the length, in bytes, of the work area whose address you specified in the AREA parameter.

If you selected the OPTCD=MVE parameter, then this length must be no less than the size of a data record. For variable-length records, you must allow for the largest record in the file.

If you selected the OPTCD=LOC parameter, then you must specify a length of 4 bytes to accept the address of the I/O buffer where you will process each record.

### ARG

Specifies the address of a field that contains the search argument for one of the following:

- Direct or skip sequential retrieval (GET).
- Sequential positioning (POINT).
- Direct or skip sequential storage (PUT) for a relative record file.

For keyed access (OPTCD=KEY), the search argument may be a

- Full key (OPTCD=FKS).
- Generic key (OPTCD=GEN). Here, you must also specify its size through the KEYLEN parameter.
- Relative record number (which is treated as a key).

For addressed access (OPTCD=ADR), the search argument is always an RBA. To determine the RBA of a record where you have gained access sequentially or directly by key, you can enter the SHOWCB macro. See “[SHOWCB](#)” on page 480.

For control interval access with user buffering and a user supplied RBA, the record is written only to this RBA if positioning is not established by a previous request.

When records are inserted into a key sequenced file, either sequentially or directly, VSAM obtains the key from the record itself. When the records are inserted sequentially into a relative record file, VSAM returns the assigned relative record number in the ARG field.

**ECB**

Specifies the address of the event control block associated with the VSAM request you will make.

**KEYLEN**

Specifies the length, in bytes, of the generic key that you are using as a search argument (OPTCD=GEN).

You specify the search argument in the ARG parameter. You must specify its length when it is a generic key.

You can write this parameter as any number from 1 to 255.

**NXTRPL**

Specifies the address of the next request parameter list in the chain.

Omit this parameter from the RPL macro that generates the last RPL in the chain. When you enter a request that is defined by a chain of RPLs, specify the address of the first RPL in the chain in the instruction associated with the request.

**OPTCD**

Indicates the options that will govern the request defined by the request parameter list you are creating.

**ADR**

Indicates addressed access to a key-sequenced or entry-sequenced file.

**CNV**

Indicates access will be to the entire contents of a control interval, rather than to an individual record.

**KEY**

Indicates access to a key-sequenced or relative record file.

**DIR**

Indicates direct processing.

**SEQ**

Indicates sequential processing.

**SKP**

Indicates skip-sequential processing.

This is valid only with keyed access.

**ARD**

Indicates that the user's argument determines the record to be located, retrieved, or stored.

**LRD**

Indicates that the last record in the file will be located or retrieved.

If you choose this parameter, then you must also choose the BWD parameter.

**BWD**

Indicates that processing is to go through the file in a backward direction for keyed or addressed access and for sequential or direct processing.

**FWD**

Indicates that processing is to go through the file in a forward direction.

**ASY**

Specifies that you want your file processed asynchronously.

This means that when the request associated with the RPL you are creating is scheduled, control will return to your program so it can continue processing. Meanwhile, your request is being carried out.

Remember that asynchronous processing is merely simulated by GCS. Disk I/O in GCS is always synchronous. Even so, you must enter the CHECK macro to obtain the results of the operation. See “CHECK” on page 425.

#### **SYN**

Specifies that you want your file processed synchronously.

This means that control will return to your program only after the request associated with the RPL you are creating has been carried out.

#### **NSP**

Indicates that, for direct processing only, the request is not for update. VSAM will be positioned at the next record for subsequent sequential processing.

#### **NUP**

Indicates that any record retrieved will not be updated or deleted. Moreover, any record that is stored is a new record.

On direct access requests, GCS does not remember the record's position.

#### **UPD**

Indicates that any record retrieved can be updated or deleted.

#### **KEQ**

Indicates that the key you provide as a search argument must equal the key of the record.

#### **KGE**

Indicates that if the key you specify as a search argument does not equal a certain record, then the request will affect the record with the next highest key.

#### **FKS**

Indicates that you are providing a full key as a search argument.

#### **GEN**

Indicates that you are providing a generic key as a search argument.

If you select this parameter, then you must also specify the length of the generic key in the KEYLEN parameter.

#### **LOC**

Indicates that during retrieval, the record will be put in VSAM's I/O buffer to be processed.

This parameter is not valid if you intend to call the PUT or ERASE macros, though it is valid with the GET macro. However, to update the record, you must build a new version of it in a work area and modify the RPL from LOCATE MODE to MOVE MODE before you enter any PUT macro. For keyed-sequential retrieval, modifying key fields in the I/O buffer may cause erroneous results for subsequent GET requests until the record is reread.

#### **MVE**

Indicates that, during retrieval, the record will be moved to a work area for processing. For storage, it will be moved from the work area to the I/O buffer.

#### **RECLEN**

Specifies the length, in bytes, of a record that is to be stored.

If you intend to enter the PUT macro, then this parameter is required. If you enter a GET macro, then the length of the record involved is placed in the RPL field associated with this parameter. This is for the benefit of any subsequent update or store requests.

#### **TRANSID**

Specifies a number from 0 to 31 when RPL= is specified.

##### **Number**

##### **Description**

##### **0**

Indicates that the request defined by this RPL is not associated with other requests.

**1-31**

Relates the requests defined by this RPL to the requests defined by other RPLs with the same TRANSID value.

**Usage (RPL)**

1. Whatever value you assign to a parameter in this instruction is the value that will replace the one currently associated with the parameter in the RPL.
2. Notice that the options under the OPTCD parameter are divided into groups. Only one option per group can be in effect at one time. If you specify one option from a group in the MODCB macro, then that option overrides any other option from that group that might be specified in the RPL.
3. Each time you enter the MODCB macro, you must provide the system with a 72-byte save area. Before you enter the instruction, place the address of this save area in register 13.
4. See [Appendix B, "Using VSAM," on page 517](#).

**Completion Codes, Return Codes, and ABEND Codes (RPL)**

When this macro completes execution, it passes to the caller a completion code in register 15.

<b>Completion Code</b>	<b>Meaning</b>
0	Function completed successfully.
4	Function completed unsuccessfully.
8	You attempted to use the execute form of this macro to modify a keyword that is not in the parameter list.
12	The MODCB macro was not executed because an error occurred while a VSAM module was being loaded.

When register 15 contains 4, then register 0 contains one of the following return codes.

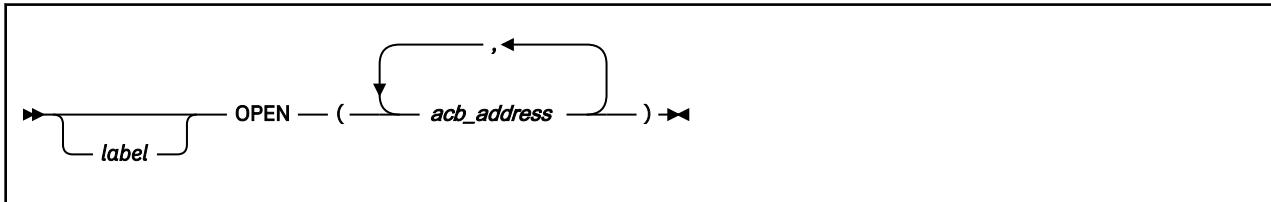
<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'01'	1	The type of request was invalid.
X'02'	2	The block type was invalid.
X'03'	3	One of the keywords in the parameter list is invalid.
X'04'	4	The block at the address you specified was not of the type you indicated.
X'0B'	11	The MODCB macro is already active on the specified control block.
X'0E'	14	You specified an incompatible set of parameters.
X'10'	16	You specified an invalid control block address in the RPL parameter.

<b>ABEND Code</b>	<b>Meaning</b>
03B	An invalid address was found in a VSAM control block or a VSAM parameter list. This means that your program tried to use an address to which it has no access.

# OPEN

---

## Format



## Purpose

Use the OPEN macro to prepare a VSAM file for processing.

Before your program can access a file, the file must be opened for processing. The process of preparing a file includes:

- Logically connecting your program to the file.
- Building various control blocks needed by VSAM to process the file.
- Verifying that the file matches the one you described through the ACB or GENCB macros.
- Verifying any necessary passwords to the file. See “[ACB](#)” on page 418 or “[GENCB](#)” on page 437.

This discussion of the OPEN macro deals only with those matters that involve GCS.

## Parameters

### *acb\_address*

Specifies the address of the access method control block (ACB) associated with the file you wish to open.

You can specify the address of more than one, and open more than one file. If you do specify more than one ACB address, be certain to separate each by a comma.

You can write this parameter as an assembler program label or as register (2) through (12). If you specify the address using a register, each register in the list is surrounded by a pair of parentheses, and the list itself is surrounded by a set of parentheses.

## Usage

1. See [Appendix B, “Using VSAM,” on page 517](#).
2. If you have data control blocks (DCBs) that you wish to open, and ACBs, you can specify a combination of both in the same OPEN macro. GCS is able to distinguish the address of one from the address of the other, if you separate each with a comma. This macro and the one described in “[OPEN \(BSAM/QSAM\)](#)” on page 398 are similar.

## Completion Codes, Return Codes, and ABEND Codes

When this macro completes processing, it passes to the caller a completion code in register 15. If register 15 contains 4 or 8, then the specific error is indicated in the ERROR field of the appropriate ACB. This field can be displayed through the SHOWCB macro. See “[SHOWCB](#)” on page 480.

Completion Code	Meaning
0	All the files specified are now opened.

<b>Completion Code</b>	<b>Meaning</b>	
4	All the files specified are now opened. However, one or more warning messages have been issued.	
8	At least one of the files specified was not opened. The ACB associated with the file(s) not opened have been restored to their original condition. If any of these files were already opened, then their ACBs remain open, usable, and unchanged.	
<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'04'	4	The file indicated by the access method control block is already open.
X'5C'	92	MACRF=LSR is specified but the OPEN/CLOSE/TCLOSE message area was not specified; therefore, it is impossible to capture error messages produced during implied buffer write.
X'60'	96	Catalog recovery for this file failed. Therefore, the file is not usable.
X'64'	100	The OPEN macro found an empty alternate index that is part of an upgrade set.
X'68'	104	The time stamp of the volume on which a file is stored does not match the system time stamp in the file's catalog record. This indicates that extent information in the catalog may not agree with the extents indicated in the volume's VTOC.
X'6C'	108	The time stamps of a data and index component do not match. This indicates that the data and the index were not updated at the same time.
X'74'	116	The file was not properly closed.
X'80'	128	Either the DLBL command for the file or for the catalog is missing, or the file specified in that statement does not match the name of the ACB.
X'84'	132	A permanent I/O error occurred while VSAM was reading label information.
X'88'	136	Insufficient virtual storage is available for work areas, control blocks, or buffers.
X'90'	144	An I/O error, which cannot be corrected, occurred while VSAM was reading or writing a catalog record.
X'94'	148	Either no record for the file to be opened was found in the available catalog(s), or an unidentified error occurred while VSAM was searching the catalog.
X'98'	152	Security verification failed. The password specified in the access method control block for a specified level of access does not match the password in the catalog for that level of access.

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'A0'	160	<p>The parameters specified in the ACB or GENCB macro are either inconsistent with each other, or inconsistent with the information in the catalog record.</p> <p>This means:</p> <ul style="list-style-type: none"> <li>• MACRF options are inconsistent.</li> <li>• MACRF DFR is specified for a data set that was defined with SHAREOPTION (4).</li> <li>• Attempt to open a compressed cluster in control interval mode.</li> </ul>
X'A8'	168	<p>Either the file is not available for the type of processing you specified, or an attempt was made to open a reusable file with the RESET option while another user had the file open.</p>
X'B4'	180	An error occurred in opening a catalog.
X'BC'	188	The file specified by the access method control block is not one that can be specified by an ACB.
X'C0'	192	An unusable file was opened for output.
X'C2'	194	Attempt to open data component of a compressed cluster.
X'C4'	196	Access to data was requested through an empty alternate index.
X'D4'	212	MACRF=LSR is specified, but the data set opened is empty.
X'D8'	216	MACRF=LSR is specified, but the key length of the data set opened is greater than the maximum key length specified in the BLDVRP for the resource pool.
X'DC'	220	MACRF=LSR is specified, but the control interval size of the data set opened is greater than the largest buffer size specified in BLDVRP for the resource pool.
X'E4'	228	MACRF=LSR is specified, but the VSAM Shared Resources Table does not exist.
X'E8'	232	RESET was specified for a non-reusable file, but the file is not empty.
X'F6'	246	Compression Management Services error during OPEN.
X'F7'	247	Compression Control error during OPEN.
<b>ABEND Code</b>	<b>Meaning</b>	
013	An error occurred during the execution of the OPEN macro. You will receive a message explaining this further.	
035	An error occurred in the OPEN macro.	
03A	The number of exits defined in the system has reached the maximum of 128.	
03B	An invalid address was found in a VSAM control block or a VSAM parameter list. This means that your program tried to use an address to which it has no access.	

## **POINT**

### **Format**



### **Purpose**

Use the POINT macro to position yourself forward or backward within the VSAM file to a specific record.

To access a certain record within a VSAM file, you must *position yourself* within that file and *point* to the record in question.

This discussion of the POINT macro deals only with those matters that involve GCS.

### **Parameters**

#### **RPL**

Specifies the address of the request parameter list (RPL) associated with your POINT request.

This is the same request parameter list that you defined through the RPL macro. See “[RPL](#)” on page [472](#). You specify the record to which you want to point in the ARG parameter of that instruction.

You can write this parameter as an assembler program label or as register (2) through (12).

### **Usage**

1. If you specify the OPTCD=KEY parameter in the appropriate RPL instruction, then the POINT macro establishes a pointer indicating the record whose key or relative-record number you specified in the search argument field. You can use the POINT macro to position either forward or backward within the file.
2. If you specify the OPTCD=ADR or OPTCD=CNV parameter in the appropriate RPL instruction, then the POINT macro establishes a pointer indicating the record or control interval whose RBA you specified in the search argument field. You can use the POINT macro to position either forward or backward within the file.
3. VSAM can also be positioned for sequential processing by either a direct GET or PUT macro.
4. Each time you enter the POINT macro, you must provide the system with a 72-byte save area. Before you enter the instruction place the address of this save area in register 13.
5. See [Appendix B, “Using VSAM,” on page 517](#).

### **Return Codes and ABEND Codes**

When this macro completes processing, it passes to the caller a return code in register 15. If register 15 contains 8 or 12, then the specific error is indicated in the FDBK field of the appropriate RPL. This field can be displayed through the SHOWCB or TESTCB macros. See “[SHOWCB](#)” on page [480](#) or “[TESTCB](#)” on page [490](#).

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'00'	0	Your request was accepted.

Hex Code	Decimal Code	Meaning
X'04'	4	Your request was not accepted because the RPL you specified in the POINT macro is already active for another request.
X'08'	8	A logical error occurred.
X'0C'	12	A physical error occurred.
ABEND Code	Meaning	
035	An error occurred in the POINT macro. The message preceding the ABEND describes this further.	
03B	An invalid address was found in a VSAM control block or a VSAM parameter list. This means that your program tried to use an address to which it has no access.	

# PUT

---

## Format



## Purpose

Use the PUT macro to store a record from an I/O buffer or work area into a VSAM file.

This discussion of the PUT macro deals only with those matters that involve GCS.

## Parameters

### RPL

Specifies the address of the request parameter list (RPL) associated with your PUT request.

This is the same request parameter list that you defined through the RPL macro. See “[RPL](#)” on page [472](#).

You can write this parameter as an assembler program label or as register (2) through (12).

## Usage

1. Each time you enter the PUT macro, you must provide the system with a 72-byte save area. Before you enter the instruction, place the address of this save area in register 13.
2. See [Appendix B, “Using VSAM,” on page 517](#).

## Return Codes and ABEND Codes

When this macro completes processing, it passes to the caller a return code in register 15. If register 15 contains 8 or 12, then the specific error is indicated in the FDBK field of the appropriate RPL. This field can be displayed through the SHOWCB or TESTCB macros. See “[SHOWCB](#)” on page [480](#) or “[TESTCB](#)” on page [490](#).

Hex Code	Decimal Code	Meaning
X'00'	0	Your request was accepted.
X'04'	4	Your request was not accepted because the RPL you specified in the PUT macro is already active for another request.
X'08'	8	A logical error occurred.
X'0C'	12	A physical error occurred.

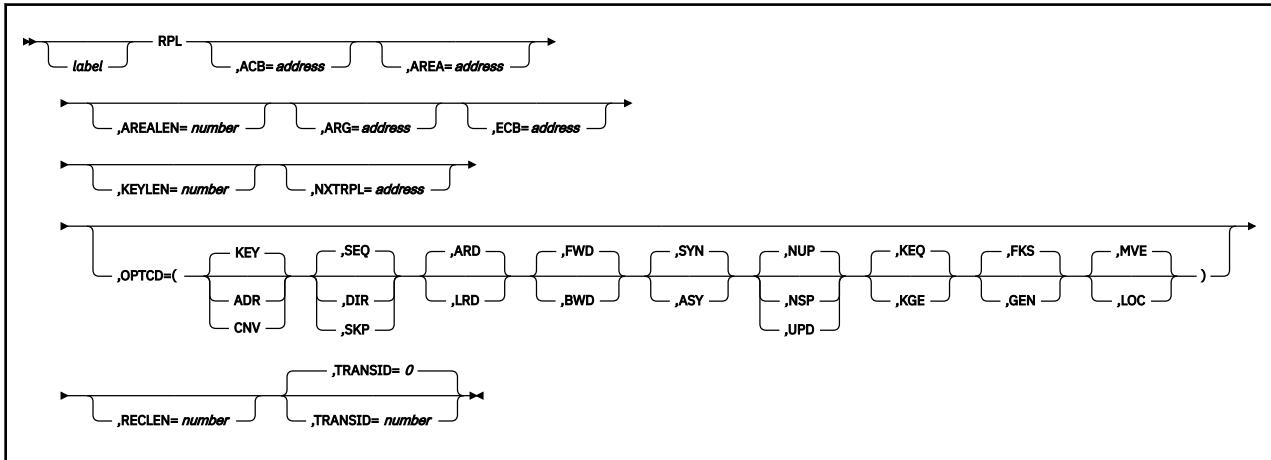
ABEND Code	Meaning
035	An error occurred in the PUT macro.
03B	An invalid address was found in a VSAM control block or a VSAM parameter list. This means that your program tried to use an address to which it has no access.

<b>ABEND Code</b>	<b>Meaning</b>
03B	You specified a TYPE parameter of CHK or DRBA and those parameters are not supported.

## RPL

---

### Format



### Purpose

Use the RPL macro to create a Request Parameter List (RPL) at assembly time describing the characteristics of your VSAM request.

Certain VSAM functions require that you set up a request parameter list (RPL). These VSAM functions are associated with the following macros: CHECK, ENDREQ, ERASE, GET, POINT, PUT, and WRTBFR. See “[CHECK](#)” on page 425, “[ENDREQ](#)” on page 430, “[ERASE](#)” on page 432, “[GET](#)” on page 451, “[POINT](#)” on page 468, “[PUT](#)” on page 470, or “[WRTBFR](#)” on page 505.

This discussion of the RPL macro deals only with those matters that involve GCS.

### Parameters

#### ACB

Specifies the address of the access method control block (ACB) associated with the file you are processing.

If you created the access method control block through the ACB macro, you can write this parameter as the assembler program label on that instruction. If no ACB associated with your file exists, then you must create one through the GENCB macro. If necessary, review the entry titled “[GENCB](#)” on page 437.

#### AREA

Specifies one of two things:

- If you select the OPTCD=MVE parameter, then the AREA parameter specifies the address of a work area where data records are moved for processing and moved from after processing.
- If you select the OPTCD=LOC parameter, then the AREA parameter will specify the address of a work area. The address of the I/O buffer which you process your file will be placed in this work area.

#### AREALEN

Specifies the length, in bytes, of the work area whose address you specified in the AREA parameter.

If you selected the OPTCD=MVE parameter, then this length must be no less than the size of a data record. For variable-length records, you must allow for the largest record in the file.

If you selected the OPTCD=LOC parameter, then you must specify a length of 4 bytes to hold the address of the I/O buffer which you will process each record.

#### **ARG**

Specifies the address of a field that contains the search argument for one of the following:

- Direct or skip sequential retrieval (GET).
- Sequential positioning (POINT).
- Direct or skip sequential storage (PUT) for a relative record file.

For keyed access (OPTCD=KEY), the search argument may be a

- Full key (OPTCD=FKS).
- Generic key (OPTCD=GEN). You must also specify its size through the KEYLEN parameter.
- Relative record number (which is treated as a key).

For addressed access (OPTCD=ADR), the search argument is always an RBA. To determine the RBA of a record to which you have gained access sequentially or directly by key, you can enter the SHOWCB macro. See “[SHOWCB](#)” on page 480.

For control interval access with user buffering and a user-supplied RBA, the record is written only to this RBA if positioning is not established by a previous request.

When records are inserted into a key sequenced file, either sequentially or directly, VSAM obtains the key from the record itself. When the records are inserted sequentially into a relative record file, VSAM returns the assigned relative record number in the ARG field.

#### **ECB**

Specifies the address of the event control block associated with the VSAM request you will make.

#### **KEYLEN**

Specifies the length, in bytes, of the generic key that you are using as a search argument.

You specify the search argument in the ARG parameter. However, you must specify its length when it is a generic key.

You can write this parameter as any number from 1 to 255.

#### **NXTRPL**

Specifies the address of the next request parameter list in the chain.

Omit this parameter from the RPL instruction that generates the last RPL in the chain. When you enter a request that is defined by a chain of RPLs, specify the address of the first RPL in the chain in the instruction associated with the request.

#### **OPTCD**

Indicates the options that will govern the request defined by the request parameter list you are creating.

Carefully check the preceding format box. Note that the parameters are arranged in groups, each with a value that will be assumed by default should you forget to specify from that group. Because they are not positional parameters, they can be specified in any order.

#### **KEY**

Indicates access to a key-sequenced or relative record file.

#### **ADR**

Indicates addressed access to a key-sequenced or entry-sequenced file.

#### **CNV**

Indicates access will be to the entire contents of a control interval, rather than to an individual record.

#### **SEQ**

Indicates sequential processing.

**DIR**

Indicates direct processing.

**SKP**

Indicates skip-sequential processing

This is valid only with keyed access.

**ARD**

Indicates that the user's argument determines the record to be located, retrieved, or stored.

**LRD**

Indicates that the last record in the file will be located or retrieved.

If you choose this parameter, then you must also choose the BWD parameter.

**FWD**

Indicates that processing is to go through the file in a forward direction.

**BWD**

Indicates that processing is to go through the file in a backward direction for keyed or addressed access, and for sequential or direct processing.

**SYN**

Specifies that you want your file processed synchronously.

This means that control will return to your program only after the request associated with the RPL you are creating has been carried out.

**ASY**

Specifies that you want your file processed asynchronously.

This means that when the request associated with the RPL you are creating is scheduled, control will return to your program so it can continue processing. Meanwhile, your request is carried out.

Remember that asynchronous processing is merely simulated by GCS. Disk I/O in GCS is always synchronous. Even so, you must enter the CHECK macro to obtain the results of the operation.

**NUP**

Indicates that any record retrieved will not be updated or deleted. Any record that is stored is a new record.

On direct access requests, GCS does not remember the record's position.

**NSP**

Indicates that, for direct processing only, your request is not for update. VSAM will be positioned at the next record for further sequential processing.

**UPD**

Indicates that any record retrieved can be updated or deleted.

**KEQ**

Indicates that the key you provide as a search argument must equal the key of the record.

**KGE**

Indicates that if the key you specify as a search argument does not equal a certain record, then the request will affect the record with the next highest key.

**FKS**

Indicates that you are providing a full key as a search argument.

**GEN**

Indicates that you are providing a generic key as a search argument.

If you select this parameter, then you must also specify the length of the generic key in the KEYLEN parameter.

**MVE**

Indicates that, during retrieval, the record will be moved to a work area for processing. For storage, it will be moved from the work area to VSAM's I/O buffer.

**LOC**

Indicates that, during retrieval, the record will be put in VSAM's I/O buffer to be processed.

**RECLEN**

Specifies the length, in bytes, of a record that is to be stored.

If you intend to enter the PUT macro, then this parameter is required. If you enter a GET instruction, then the length of the record involved is placed in the RPL field associated with this parameter. This is for the benefit of any subsequent update or store requests.

**TRANSID**

Provides a means of associating buffers that contain deferred write buffer updates that represent parts of the same transaction. By using TRANSID with WRTBFR you may synchronize update commitment by transaction. TRANSID also specifies a number from 0 to 31.

**Number****Description****0**

Default value. Indicates that the request defined by this RPL is not associated with other requests.

**1-31**

Relates the requests defined by this RPL to the requests defined by other RPLs with the same TRANSID value.

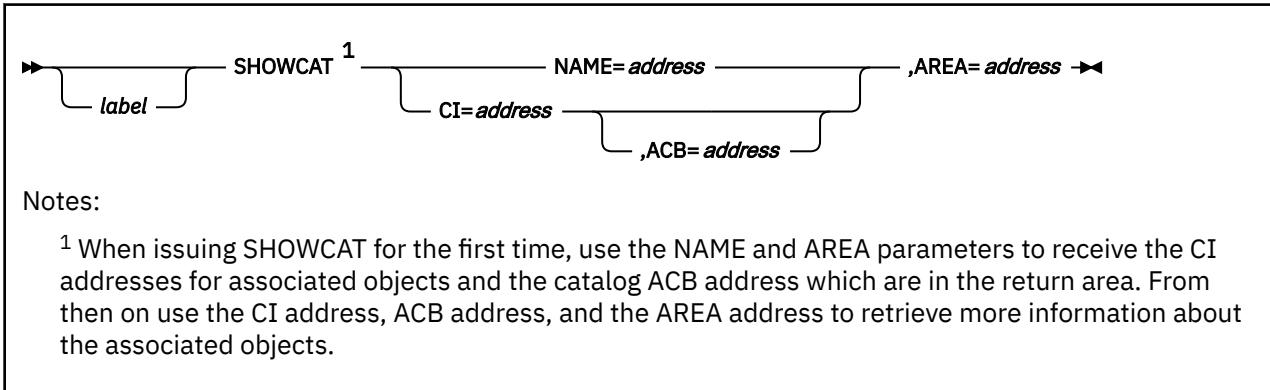
**Usage**

1. SHOWCB or TESTCB can tell you what TRANSID an RPL has.
2. MODCB can change the TRANSID of an RPL.

# SHOWCAT

---

## Format



## Purpose

Use the SHOWCAT macro to allow an application to retrieve information from the VSAM catalog. The information retrieved can be used to calculate parameters for building the resource pool for Local Shared Resources.

To use the SHOWCAT macro you must enter SHOWCAT on:

1. The name of the object.

The information returned to you includes the control interval numbers of catalog records in entries that describe associated objects.

2. A control interval number to retrieve information from one of these other entries.

## Parameters

### NAME

Specifies the address of a 44-byte area containing the data set name of the VSAM object for which you are retrieving information. The name is left justified, padded with blanks on the right, and must be an object other than upgrade set (Y).

### CI

Specifies the address of a 3-byte area that contains the control interval number of the catalog entry for the object you want displayed. All object entries are valid. The area you specify with this parameter must be distinct from the area you specify with the AREA parameter.

**Note:** When you enter the first SHOWCAT for an object using the NAME parameter, you receive the CI numbers for associated objects and the catalog ACB address in the return area. You then use these pointers to retrieve more information about the associated objects.

### ACB

Specifies the address of the ACB that defines the catalog containing the entry you want displayed.

#### Note:

1. The first time you enter SHOWCAT, do not specify an ACB. VSAM indicates the address of the ACB in the return area. Use this address and the entry CI numbers to retrieve information for the associated objects.
2. You must specify the ACB address in register notation.

**AREA**

Specifies the address of a work area where the catalog information is to be displayed.

**Note:**

1. The first 2 bytes of this area must contain the length of the work area, which includes the 2-byte length indicator.
2. The minimum size of the area is 64 bytes.

**Usage**

When using this macro you must make sure that register 13 contains the address of a 72-byte save area.

If you enter the macro from within one of your exit routines (LERAD or SYNAD) you must provide a second 72-byte save area because the original one is still in use by the external VSAM routine.

**Completion Codes**

When VSAM returns to the application after a SHOWCAT request, register 15 contains one of the following completion codes:

<b>Completion Code</b>	<b>Meaning</b>
0	VSAM completed the request successfully.
4	The area specified in the AREA parameter is less than the minimum required (64 bytes) or the area is too small to display all associated objects.
8	An error was detected while VSAM routines were being loaded.
12	Either the ACB address is invalid or the VSAM master catalog does not exist or cannot be opened.
20	The named object or control interval does not exist.
24	There was an I/O error in accessing the catalog.
28	The CI number specified is invalid.
32	The catalog record does not describe an acceptable type of object (C, D, G, I, R, or Y).
36	The information in the catalog is at a different level than that in the catalog recovery area.
40	There was an unexpected error code returned from catalog management to the SHOWCAT processor.

**Information and Format in the Catalog Display**

If the completion code returned was zero, the requested catalog information is returned in the work area that you specified with the AREA operand. The format of the returned information is as follows:

<b>Offset</b>	<b>Length (bytes)</b>	<b>Meaning</b>
0(0)	2	Length of the work area, including the length of this length field (provided by requester).
2(2)	2	Length of the work area actually used by VSAM, including the length of this field and the preceding field.
4(4)	4	The address of the ACB that defines the catalog that contains the entry which will be displayed.

Offset	Length (bytes)	Meaning
8(8)	1	Type of object about which information is returned: <ul style="list-style-type: none"> <li>• Cluster (C)</li> <li>• Data Component (D)</li> <li>• Index Component (I)</li> <li>• Alternate Index (G)</li> <li>• Path (R)</li> <li>• Upgrade Set (Y).</li> </ul>
For C, G, R, and Y types:		
Offset	Length (bytes)	Meaning
9(9)	1	For C and Y types: Reserved. For G type:  <b>X.... ....</b> The alternate index can be (1) or cannot be (0) a member of an upgrade set. The way to find out for sure is to display information for the upgrade set of the base cluster and check whether it contains control interval numbers of entries that describe the components of an alternate index.  <b>.XXX XXXX</b> Reserved.  For R type:  <b>X.... ....</b> The path is (1) or is not (0) defined with the UPDATE attribute (for upgrading alternate indexes).  <b>.XXX XXXX</b> Reserved.
10(A)	2	The number of pairs of fields that follow. Each pair of fields identifies another catalog entry that describes an object associated with this C, G, R, or Y object. The possible types of associated objects are:  With C: D, G, I, R With G: C, D, I, R With R: C, D, G, I With Y: D, I.
12(C)	1	Type of associated object the entry describes.
13(D)	3	The control interval number of its first record.
16(10)		Next pair of fields, and so on.
<b>Note:</b>		
<ol style="list-style-type: none"> <li>1. VSAM displays as many pairs as possible and returns a code of 4 in register 15 if the area is too small.</li> <li>2. Each pair of fields occupies 4 bytes, except Y-type (8 bytes).</li> </ol>		
For D and I types:		

<b>Offset</b>	<b>Length (bytes)</b>	<b>Meaning</b>
9(9)	1	Reserved.
10(A)	2	Relative position of the prime key in records in the data component. For the data component of an entry-sequenced or a relative record file there is no prime key and this field is 0.
12(C)	2	Length of the prime key.
14(E)	4	Control interval size of the data or index component.
18(12)	4	Maximum record size of the data or index component.
22(16)	2	The number of pairs of fields that follow. Each pair of fields identifies another catalog entry that describes an object associated with this D or I object. The possible types of associated objects are:  With D: C, G, Y With I: C, G.
24(18)	1	Type of associated object the entry describes.
25(19)	1	The control interval number of its first record.
28(1C)		Next pair of fields, and so on. If the minimum AREA size is specified, fields for all associated objects can always be displayed.

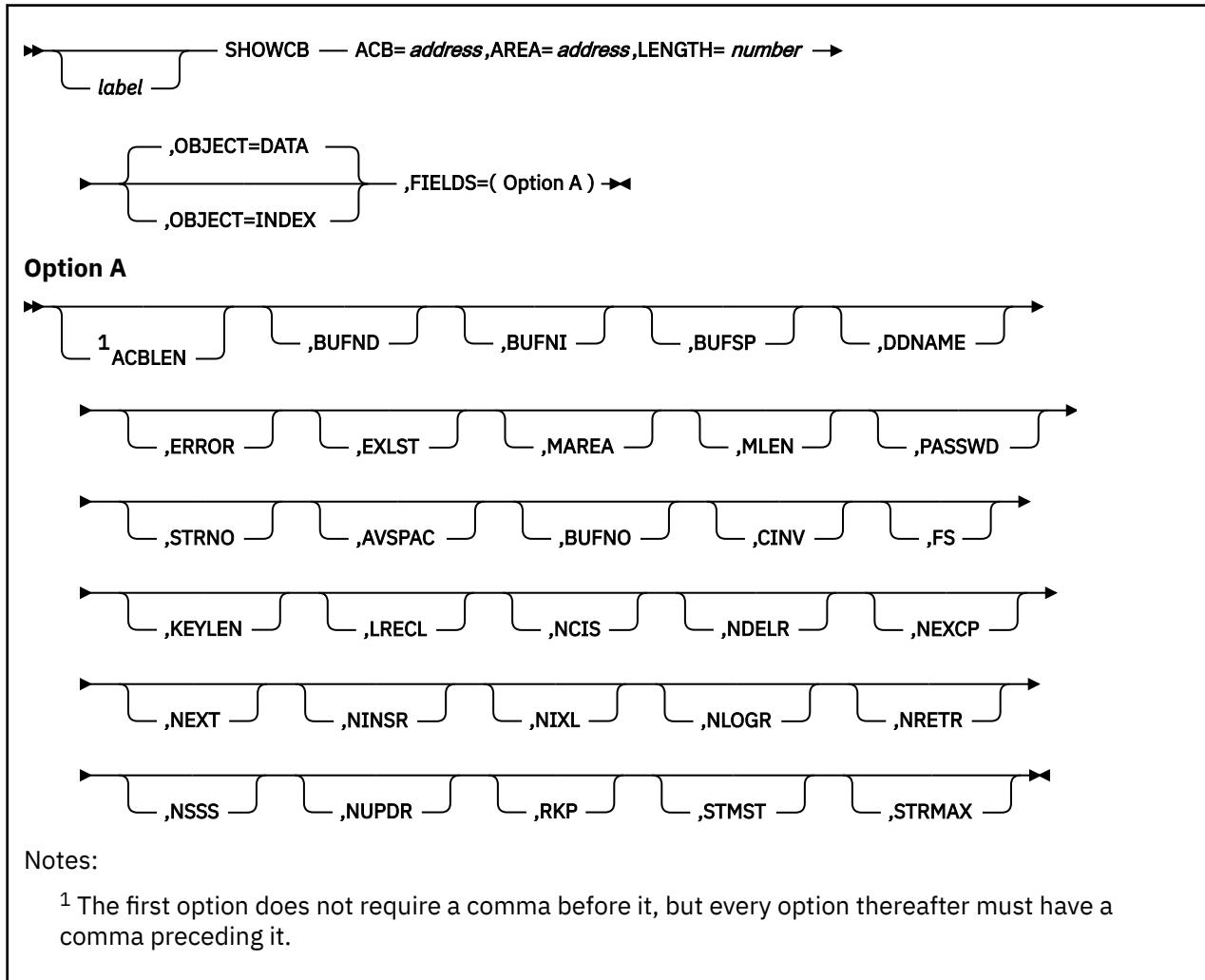
<b>ABEND Code</b>	<b>Meaning</b>
03B	An invalid address was found in a VSAM control block or a VSAM parameter list. This means that your program tried to use an address to which it has no access.

## SHOWCB

The SHOWCB macro is available in Access Control Block (ACB), Exit List (EXLST) and Request Parameter List (RPL) formats.

### Access Control Block Format

See also [“Exit List Format” on page 484](#) and [“Request Parameter List Format” on page 486](#).



### Purpose (ACB)

An access method control block (ACB) defines certain characteristics of a file that you intend to process through VSE/VSAM. When the file is opened, other characteristics of the file that you defined through the DLBL command are merged with the ACB to complete the picture.

The contents of each field (except the ACBLEN field) is determined by the corresponding parameter in the ACB macro, the GENCB macro, or the DLBL command. See [“ACB” on page 418](#) and [“GENCB” on page 437](#). For more information on the DLBL command, see [“DLBL” on page 64](#). Also, see [Appendix B, “Using VSAM,” on page 517](#).

This discussion of the SHOWCB macro deals only with those matters that involve GCS.

## Parameters (ACB)

### **ACB**

Specifies the address of the ACB containing the fields you want displayed.

All ACBs are the same length. So, if you only want the ACBLEN field displayed, you need give no ACB address.

If you entered the ACB macro with a label attached to it, then you can write this parameter as that label.

### **AREA**

Specifies the address of a work area that you have provided in virtual storage to accept the ACB fields to be displayed.

The contents of the fields are displayed in this area in the order which you list them in the SHOWCB macro.

This work area must begin on a fullword boundary.

### **LENGTH**

Specifies the length, in bytes, of the work area that you have provided in virtual storage to hold the ACB fields to be displayed.

Check the following field parameters listed to determine the necessary length for this work area. If the area is not large enough to accept all the fields you specify, then you will receive an error code.

### **OBJECT**

Indicates the scope of your request.

### **DATA**

Indicates that the fields you specify pertain to the data contained in the file. This is the case by default.

### **INDEX**

Indicates that the fields you specify pertain to the index.

### **FIELDS**

Indicates which fields in the ACB you want displayed.

Some of the ACB fields can be displayed at any time. Others can be displayed only after the file in question has been opened. As with a key-sequenced file opened for keyed access, the fields can pertain to either the data or the index.

The number following each field name specifies the number of fullwords needed in the work area to accept the field.

#### **The following fields can be displayed at any time:**

##### **ACBLEN (1)**

The length of the access method control block in question.

##### **BUFND (1)**

The number of I/O buffers used for data.

##### **BUFNI (1)**

The number of I/O buffers used for the index.

##### **BUFSP (1)**

The amount of space allocated for I/O buffers.

##### **DDNAME (2)**

The logical name of the file associated with the ACB in question.

##### **ERROR (1)**

The code returned after opening or closing the file associated with the ACB in question.

**EXLST (1)**

The address of the list of exit routine addresses. If none was specified, then this field contains 0.

**MAREA (1)**

The address of the message area. If none was specified, then this field contains 0.

**MLEN (1)**

The length of the message area. If none was specified, then this field contains 0.

**PASSWD (1)**

The address of the field containing the password to the file associated with the ACB in question.

The first byte of the field contains the binary length of the password.

**STRNO (1)**

The number of requests for which the position in the file is to be remembered.

**The following fields can be displayed only after the file is open:****AVSPAC (1)**

The amount of available space, in bytes, in the data component or index component.

**BUFN0 (1)**

The number of I/O buffers actually in use by the data component or index component.

**CINV (1)**

The control interval size for the data component or index component.

**FS (1)**

The number of free control intervals per control area in the data component. If you specified the OBJECT=INDEX parameter, then this field contains 0.

**KEYLEN (1)**

Either the full length of the prime key field or the alternate key field in each logical record. Which it is depends on whether you access the base cluster through a path.

**LRECL (1)**

The length of the records in the data component or the index component. For the former, with variable-length records, this is the maximum length of any record. For the latter, this is the control interval length minus seven.

**NCIS (1)**

The number of control intervals that have been split in the data component. If you specified the OBJECT=INDEX parameter, then this field contains 0.

**NDELR (1)**

The number of records that have been deleted from the data component. If you specified the OBJECT=INDEX parameter, then this field contains 0.

**NEXCP (1)**

The number of EXCP macros that have been issued to obtain access to the data component or index component.

**NEXT (1)**

The number of extents currently allocated to the data component or the index component.

**NINSR (1)**

The number of records that have been inserted into the data component. If you specified the OBJECT=INDEX parameter, then this field contains 0.

**NIXL (1)**

The number of levels in the index component. If you specified the OBJECT=DATA parameter, then this field contains 0.

**NLOGR (1)**

The number of records in the data component. If you specified the OBJECT=INDEX parameter, then this field contains 0.

**NRETR (1)**

The number of records that have ever been retrieved from the data component. If you specified the OBJECT=INDEX parameter, then this field contains 0.

**NSSS (1)**

The number of control areas that have been split in the data component. If you specified the OBJECT=INDEX parameter, then this field contains 0.

**NUPDR (1)**

The number of records in the data component that have ever been updated. If you specified the OBJECT=INDEX parameter, then this field contains 0.

**RKP (1)**

The displacement of either the prime key field or alternate key field from the beginning of a data record. Which it is depends on whether you access the base cluster through a path. The same value is displayed whether the object is index or data.

**STMST (2)**

The system time stamp, which specifies the time and date on which the data component or index component was closed. Bit 51 is equivalent to one microsecond and bits 52 through 63 are unused.

**STRMAX (1)**

Specifies the maximum number of request which were concurrently active because the resource pool was built. The ACB specified must be associated with a resource pool, that is the parameter MACRF=(LSR) must have been specified for the ACB.

**Usage (ACB)**

1. Each time you enter the SHOWCB macro, you must provide the system with a 72-byte save area. Before you enter the instruction, place the address of this save area in register 13.

**Completion Codes, Return Codes, and ABEND Codes (ACB)**

When this macro completes execution, it passes to the caller a completion code in register 15.

Completion Code	Meaning
0	Function completed successfully.
4	Function completed unsuccessfully.
8	You attempted to use the execute form of this macro to modify a keyword that is not in the parameter list.
12	The SHOWCB macro was not executed because an error occurred while a VSAM module was being loaded.

When register 15 contains 4, then register 0 contains one of the following return codes.

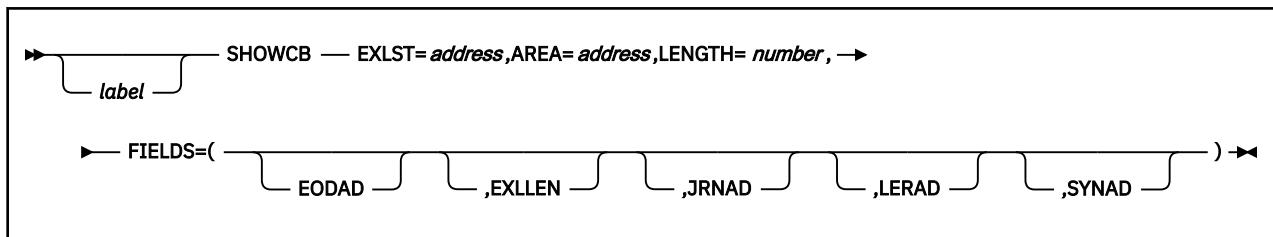
Hex Code	Decimal Code	Meaning
X'01'	1	The type of request was invalid.
X'02'	2	The block type was invalid.
X'03'	3	One of the keywords in the parameter list is invalid.
X'04'	4	The block at the address you specified was not of the type you indicated.
X'05'	5	Either the file associated with the ACB in question is not open or is not a VSAM file.

Hex Code	Decimal Code	Meaning
X'06'	6	Index information was requested, but no index was opened for the file in question.
X'09'	9	The work area you provided to hold the fields to be displayed is too small.
X'0F'	15	The work area you provided to hold the fields to be displayed is not on a fullword boundary.
X'10'	16	You specified an invalid control block address in the ACB parameter.
X'14'	20	You specified certain parameters that can apply only if MACRF=LSR or MACRF=GSR. MACRF=GSR is not supported by GCS. TRANSID was specified, but LSR was not specified in the ACB.

ABEND Code	Meaning
03B	An invalid address was found in a VSAM control block or a VSAM parameter list. This means that your program tried to use an address to which it has no access.

## Exit List Format



## Purpose (EXLST)

During VSAM processing, unusual conditions sometimes occur. If you wish, you can supply one or more exit routines to handle such conditions. See “[EXLST](#)” on page 434 or “[GENCB](#)” on page 437. You can then associate them with one or more access method control blocks (ACBs) that define the characteristics of the VSAM files you plan to process. See “[ACB](#)” on page 418 or “[MODCB](#)” on page 453.

This discussion of the SHOWCB macro deals only with those matters that involve GCS.

Use the SHOWCB macro to display certain fields of an exit list. This display appears in a virtual storage work area that you set aside for this purpose.

## Parameters (EXLST)

### EXLST

Specifies the address of the exit list whose fields you want to display.

If you omit this parameter and specify the EXLLEN parameter, then the EXLLEN field will display the maximum allowable length of any exit list.

If you used the EXLST macro to generate the exit list, and you applied a label to that instruction, then you can write this parameter as that label.

### AREA

Specifies the address of a work area in virtual storage you have set aside for the display of the exit list fields.

This area must begin on a fullword boundary. The fields are displayed in the order which you specify them in the SHOWCB macro.

#### **LENGTH**

Specifies the length, in bytes, of a work area in virtual storage you have set aside for the display of the exit list fields.

Each exit list field requires one fullword. Therefore, allow 4 bytes for each field you specify in the FIELD parameter. If the work area is not large enough to accept all the fields you specify, then you will receive an error code.

#### **FIELDS**

Indicates the scope of your request.

##### **EODAD**

Indicates that the address of the END-OF-FILE routine will be displayed.

##### **EXLLEN**

Specifies one of two things:

- If the EXLST parameter is specified, then the length of the exit list will be displayed.
- If the EXLST parameter is not specified, then the maximum allowable length of any exit list will be displayed.

##### **JRNAD**

Specifies that the address of the journaling routine will be displayed.

##### **LERAD**

Specifies that the address of the logical error analysis routine will be displayed.

##### **SYNAD**

Specifies that the address of the physical error analysis routine will be displayed.

#### **Usage (EXLST)**

1. Use the SHOWCB macro to display a certain field in an exit list only if that field exists.

GCS will display the fields in the order which you request them.

2. Each time you enter the SHOWCB macro, you must provide the system with a 72-byte save area. Before you enter the macro, place the address of this save area in register 13.

3. See [Appendix B, “Using VSAM,” on page 517](#).

#### **Completion Codes, Return Codes, and ABEND Codes (EXLST)**

When this macro completes execution, it passes to the caller a completion code in register 15.

<b>Completion Code</b>	<b>Meaning</b>
0	Function completed successfully.
4	Function completed unsuccessfully.
8	You attempted to use the execute form of this macro to modify a keyword that is not in the parameter list.
12	The SHOWCB macro was not executed because an error occurred while a VSAM module was being loaded.

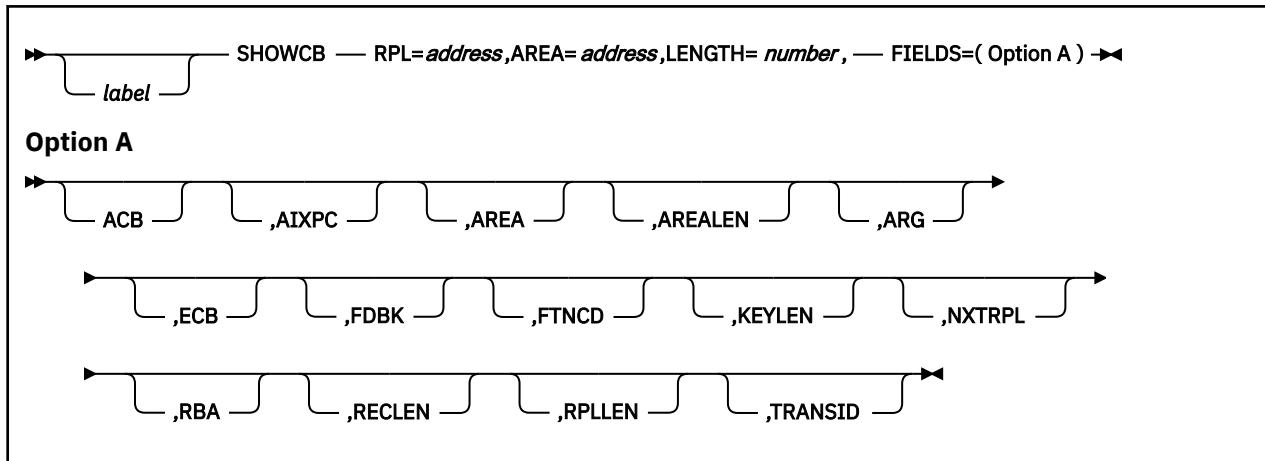
When register 15 contains 4, then register 0 contains one of the following return codes.

Hex Code	Decimal Code	Meaning
X'01'	1	The type of request was invalid.
X'02'	2	The block type was invalid.
X'03'	3	One of the keywords in the parameter list is invalid.
X'04'	4	The block at the address you specified was not of the type you indicated.
X'07'	7	The type of exit you specified is not in the exit list.
X'09'	9	The work area you provided to accommodate the fields to be displayed is too small. No fields were displayed.
X'0F'	15	The work area you provided to accommodate the fields to be displayed is not on a fullword boundary. No fields were displayed.
X'10'	16	You specified an invalid control block address in the EXLST parameter.

ABEND Code	Meaning
03B	An invalid address was found in a VSAM control block or a VSAM parameter list. This means that your program tried to use an address to which it has no access.

## Request Parameter List Format



## Purpose (RPL)

All VSAM functions require that you set up a request parameter list (RPL) that describes the characteristics of your request. These VSAM functions are associated with the following macros: CHECK, ENDREQ, ERASE, GET, POINT, and PUT. See “CHECK” on page 425, “ENDREQ” on page 430, “ERASE” on page 432, “GET” on page 451, “POINT” on page 468, or “PUT” on page 470. Also, see Appendix B, “Using VSAM,” on page 517.

You create a request parameter list through the RPL or GENCB macros. See “RPL” on page 472 or “GENCB” on page 437.

This discussion of the SHOWCB macro deals only with those matters that involve GCS.

Use the SHOWCB macro to display certain fields of a request parameter list. This display appears in a work area that you have set aside for this purpose.

## Parameters (RPL)

### RPL

Specifies the address of the request parameter list whose fields you want to display.

Because all RPLs are the same length, you can omit this parameter if the only field you are interested in displaying is the RPLLEN field.

If you used the RPL macro to create this request parameter list, and you applied a label to that instruction, then you can write this parameter as that label.

### AREA

Specifies the address of a work area in virtual storage you have set aside to accommodate the RPL fields you want to display.

This work area must begin on a fullword boundary. The fields are displayed in this work area in the order which you list them in the SHOWCB macro.

### LENGTH

Specifies the length, in bytes, of the work area in virtual storage you have set aside to accept the RPL fields you want to display.

Each RPL field requires one fullword. Therefore, allow 4 bytes for each field you specify in the FIELDS parameter.

### FIELDS

Indicates which fields you want to display.

#### ACB

The address of the access method control block that relates the RPL to the file you are processing.

#### AIXPC

The number of alternate index pointers.

#### AREA

The address of the work area that your program uses to process the file records. Access to this file is defined by the RPL.

#### AREALEN

The length of the work area whose address is specified in the AREA field.

#### ARG

If you are using search arguments to process your file, the address of the field containing that search argument.

#### ECB

The address of the event control block associated with the RPL in question. It is in this ECB that the completion of the request associated with the RPL is posted.

#### FDBK

The address of the feedback field that will contain the return code from the request associated with this RPL.

For asynchronous requests, you must enter the CHECK macro to place the return code in this field. See “[CHECK](#)” on page 425. The significance of this return code depends on the contents of register 15, which indicates whether the request was successful or unsuccessful because of logical or physical error.

#### FTNCD

The code that describes the function which a logical or physical error occurred.

#### KEYLEN

If you are using a generic key as a search argument, the length of that argument.

**NXTRPL**

The address of the next request parameter list in the chain, if one exists.

**RBA**

The relative byte address of the most recently processed record in the file.

**RECLEN**

The length of the file record, access to which is defined by the request parameter list.

**RPLLEN**

The length, in bytes, of any request parameter list.

**TRANSID**

Specifies that you want to have the TRANSID displayed in your work area when RPL= is specified.

**Usage (RPL)**

1. Each time you enter the SHOWCB macro, you must provide the system with a 72-byte save area. Before you enter the instruction, place the address of this save area in register 13.
2. Display of TRANSID requires one fullword in your work area.

**Completion Codes, Return Codes, and ABEND Codes (RPL)**

When this macro completes execution, it passes to the caller a completion code in register 15.

<b>Completion Code</b>	<b>Meaning</b>
0	Function completed successfully.
4	Function completed unsuccessfully.
8	You attempted to use the execute form of this macro to modify a keyword that is not in the parameter list.
12	The SHOWCB macro was not executed because an error occurred while a VSAM module was being loaded.

When register 15 contains 4, then register 0 contains one of the following return codes.

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'01'	1	The type of request was invalid.
X'02'	2	The block type was invalid.
X'03'	3	One of the keywords in the parameter list is invalid.
X'04'	4	The block at the address you specified was not of the type you indicated.
X'09'	9	The work area you provided to hold the fields to be displayed is too small. No fields were displayed.
X'0F'	15	The work area you provided to hold the fields to be displayed is not on a fullword boundary. No fields were displayed.
X'10'	16	You specified an invalid control block address in the RPL parameter.
X'14'	20	TRANSID was specified, but LSR was not specified in the ACB.

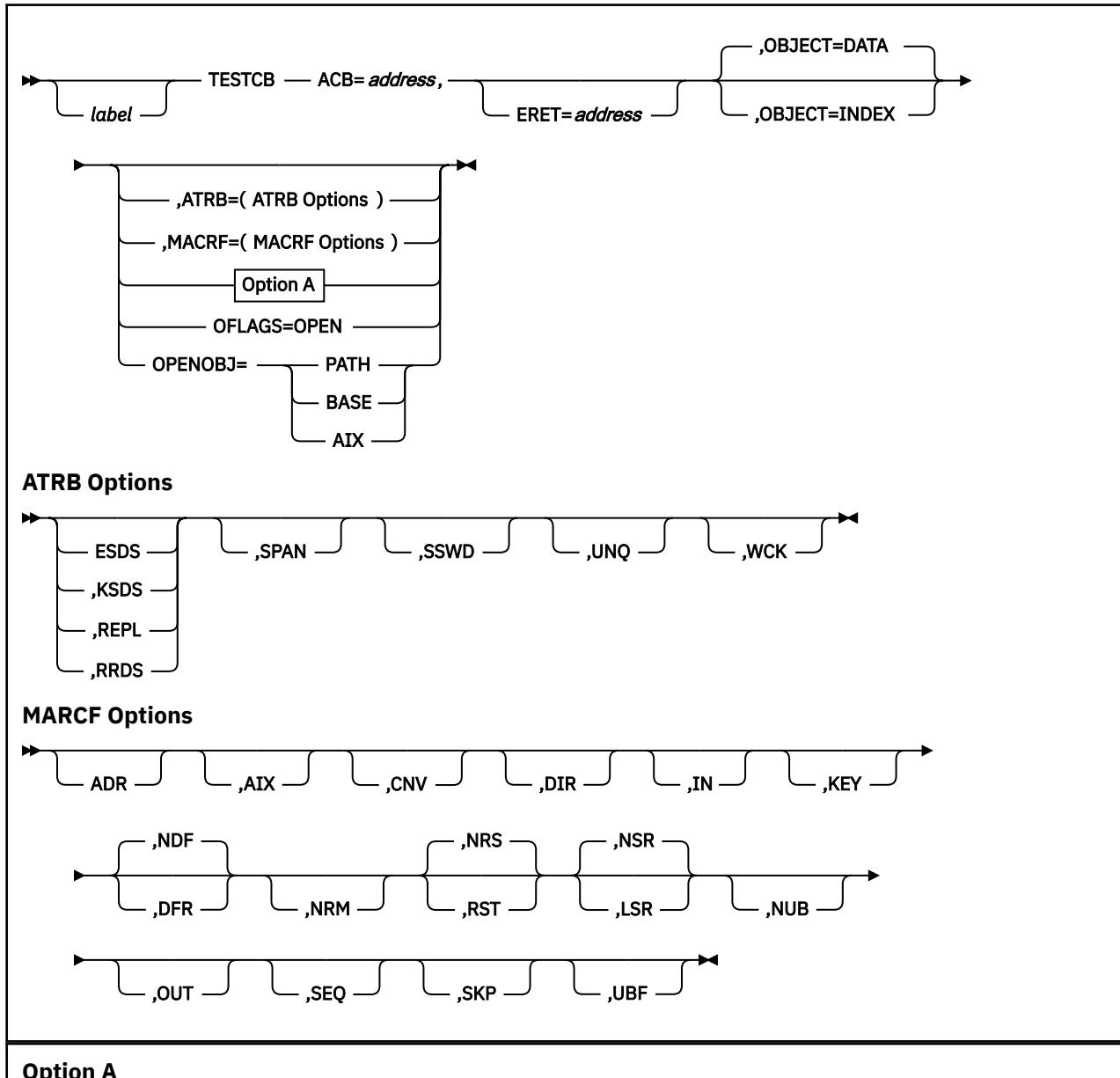
<b>ABEND Code</b>	<b>Meaning</b>
03B	An invalid address was found in a VSAM control block or a VSAM parameter list. This means that your program tried to use an address to which it has no access.

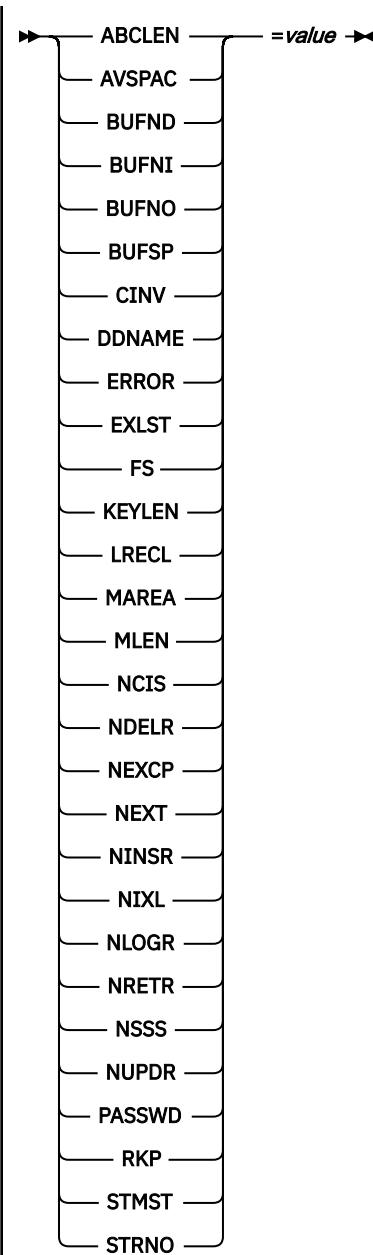
## TESTCB

The TESTCB macro is available in Access Control Block (ACB), Exit List (EXLST) and Request Parameter List (RPL) formats.

### Access Control Block Format

See also [“Exit List Format” on page 496](#) and [“Request Parameter List Format” on page 499](#).





## Purpose (ACB)

An access method control block (ACB) defines certain characteristics of a file that you intend to process through VSAM. When the file is opened, other characteristics of the file that you defined through the DLBL command are merged with the ACB to complete the picture.

The contents of each field (except the ACBLEN field) is determined by the corresponding parameter in the ACB macro, the GENCB macro, or the DLBL command. See “[ACB](#)” on page 418 and “[GENCB](#)” on page 437. For more information on the DLBL command, see “[DLBL](#)” on page 64.

This discussion of the TESTCB macro deals only with those matters that involve GCS.

## Parameters (ACB)

### ACB

Specifies the address of the ACB that contains the information you want to test.

Because all ACBs have the same length, you can omit this parameter if the field you want to test is the ACBLEN field.

**ERET**

Specifies the address of a routine that will receive control if the condition you want to test for cannot be tested.

This routine receives control if the TESTCB macro places a return code of 4 in register 15. Upon entry to this routine, register 0 contains additional information describing the error.

The ERET routine probably should issue an ABEND macro, because a failure to carry out a test is probably the result of a program logic error. If the ERET routine allows the program to continue, then it must transfer control to the continuation point, though it must not return to VSAM.

**OBJECT**

Indicates the scope of the test.

**DATA**

Indicates that the test will affect the data component. This is the case by default.

**INDEX**

Indicates that the test will affect the index component.

**ATRB**

Indicates the attribute that will be tested on the open file. Select from among the following attributes for which you can test.

**ATRB Options:****ESDS**

Whether an entry-sequenced file.

**KSDS**

Whether a key-sequenced file.

**REPL**

Whether some portion of the index is replicated.

**RRDS**

Whether a relative record file.

**SPAN**

Whether the file contains spanned records.

**SSWD**

Whether a sequence set is adjacent to the data.

**UNQ**

Whether the alternate index requires unique keys.

**WCK**

Whether write operations for the file are being verified.

**MACRF**

Indicates that a test be made to determine whether certain processing options are being used. The following describes the various processing options available for which you can test.

**MARCF Options:****ADR**

Indicates addressed access to a key-sequenced or entry-sequenced file.

RBAs will be used as search arguments, and sequential access is by entry sequence.

**AIX**

Indicates that the object to be processed is the alternate index of the path specified by the DDNAME parameter, rather than the base cluster through the alternate index.

**CNV**

Indicates access will be to the entire contents of a control interval, rather than to an individual record.

**DIR**

Indicates direct access to a key-sequenced, entry-sequenced, or relative record file.

**IN**

Indicates retrieval of records from key-sequenced, entry-sequenced, or relative record files.

This is not a valid form of processing for an empty file.

**KEY**

Indicates access to a key-sequenced or relative record file.

Keys will be relative record numbers used as search arguments, and sequential access will be by key or relative record number.

**NDF**

Indicates that any WRITE macro will not be deferred for a direct PUT macro.

**DFR**

Specifies that physically writing the I/O buffers is deferred when possible.

**NRM**

Indicates that the file to be processed is the one specified by the DDNAME parameter.

**NRS**

Indicates that the file is not reusable.

**RST**

Indicates that the file is reusable.

The OPEN macro resets the file's catalog information to its original status — it resets it to the status it had before the file was first opened. See “[OPEN](#)” on page 465. Also, the high-used RBA is reset to zero.

The file must have been defined with the REUSE attribute for RST to be effective. Although the file is not erased, you can handle it as though it were a new file, and use it as a work file. When the OPEN macro carries out the reset operation, this parameter is equivalent to the OUT option. DISP=NEW specified on the DLBL command is equivalent to selecting this parameter, and will override the NRS parameter.

**NSR**

Indicates that the resources are not shared.

**LSR**

Specifies that the resources are shared. This also indicates a VSAM resource pool will be provided when opening this ACB.

**NUB**

Indicates that VSAM will manage the I/O buffers.

**OUT**

Indicates three things:

- Storage of new records in a key-sequenced, entry-sequenced, or relative record file. This is not allowed with addressed access to a key-sequenced file.
- Update of new records in a key-sequenced, entry-sequenced, or relative record file.
- Deletion of records from a key-sequenced or relative record file.

**SEQ**

Indicates sequential access to a key-sequenced, entry-sequenced, or relative record file.

**SKP**

Indicates skip-sequential access to a key-sequenced or relative record file.

This is valid only with keyed access in a forward direction.

**UBF**

Indicates that the application will manage the I/O buffers.

The work area specified by the RPL or GENCB macros will be, in effect, the I/O buffer. The contents of a control interval is transmitted directly between the work area and DASD. This parameter is valid only when the MACRF=CNV and OPTCD=MVE parameters are specified in the RPL macro. See “[RPL](#)” on page 472 and “[GENCB](#)” on page 437.

**Option A:****ACBLEN**

The length of the access method control block in question.

**AVSPAC**

The amount of available space, in bytes, in the data component or index component.

**BUFND**

The number of I/O buffers used for data.

**BUFNI**

The number of I/O buffers used for the index.

**BUFNO**

The number of I/O buffers actually in use by the data component or index component.

**BUFSP**

The amount of space allocated for I/O buffers.

**CINV**

The control interval size for the data component or index component.

**DDNAME**

The logical name of the file associated with the ACB in question.

**ERROR**

The code returned after opening or closing the file associated with the ACB in question.

**EXLST**

The address of the list of exit routine addresses. If none was specified, then this field contains 0.

**FS**

The percentage of free control intervals per control area in the data component. If you specified the OBJECT=INDEX parameter, then this field contains 0.

**KEYLEN**

The full length of the prime key field or alternate key field in each logical record. Which it is depends on whether you access the base cluster through a path.

**LRECL**

The length of the records in the data component or the index component. For the former, with variable-length records, this is the maximum length of any record. For the latter, this is the control interval length minus seven.

**MAREA**

The address of the message area. If none was specified, then this field contains 0.

**MLEN**

The length of the message area. If none was specified, then this field contains 0.

**NCIS**

The number of control intervals that have been split in the data component. If you specified the OBJECT=INDEX parameter, then this field contains 0.

**NDELR**

The number of records that have been deleted from the data component. If you specified the OBJECT=INDEX parameter, then this field contains 0.

**NEXCP**

The number of EXCP macros that have been issued to obtain access to the data component or index component.

**NEXT**

The number of extents currently allocated to the data component or the index component.

**NINSR**

The number of records that have been inserted into the data component. If you specified the OBJECT=INDEX parameter, then this field contains 0.

**NIXL**

The number of levels in the index component. If you specified the OBJECT=DATA parameter, then this field contains 0.

**NLOGR**

The number of records in the data component. If you specified the OBJECT=INDEX parameter, then this field contains 0.

**NRETR**

The number of records that have ever been retrieved from the data component. If you specified the OBJECT=INDEX parameter, then this field contains 0.

**NSSS**

The number of control areas that have been split in the data component. If you specified the OBJECT=INDEX parameter, then this field contains 0.

**NUPDR**

The number of records in the data component that have ever been updated. If you specified the OBJECT=INDEX parameter, then this field contains 0.

**PASSWD**

The address of the field containing the password to the file associated with the ACB in question. The first byte of the field contains the binary length of the password.

**RKP**

Depending on whether you access the base cluster through a path, the displacement of the prime key field or alternate key field from the beginning of a data record. The same value is displayed whether the object is index or data.

**STMST**

The system time stamp, which specifies the time and date on which the data component or index component was closed. Bit 51 is equal to one microsecond and bits 52 through 63 are unused.

**STRNO**

The number of requests for which the position in the file is to be remembered.

**value**

Is the expression you can use depending on the keyword you specify. For complete description of these expressions, see [“Parameter Notation for GENCB, MODCB, SHOWCB, and TESTCB Macros”](#) on page 519.

**OFLAGS**

Indicates that a test will be made to determine whether a file for which the OPEN macro has been issued is in fact open.

**OPENOBJ=PATH****OPENOBJ=BASE****OPENOBJ=AIX**

Indicates that a test will be made to determine whether the open object is a path, base cluster, or an alternative index. Select one.

**Usage (ACB)**

1. You can use the TESTCB macro to test only one field at a time. After the test, analyze the CONDITION CODE field of the PSW. It will indicate one of the following conditions:

- EQUAL TO
- GREATER THAN
- LESS THAN.

You can then proceed, based upon this condition code.

2. Each time you enter the TESTCB macro, you must provide the system with a 72-byte save area. Before you enter the macro, place the address of this save area in register 13.
3. See [Appendix B, “Using VSAM,” on page 517](#).

### **Completion Codes, Return Codes, and ABEND Codes (ACB)**

When this macro completes execution, it passes to the caller a completion code in register 15.

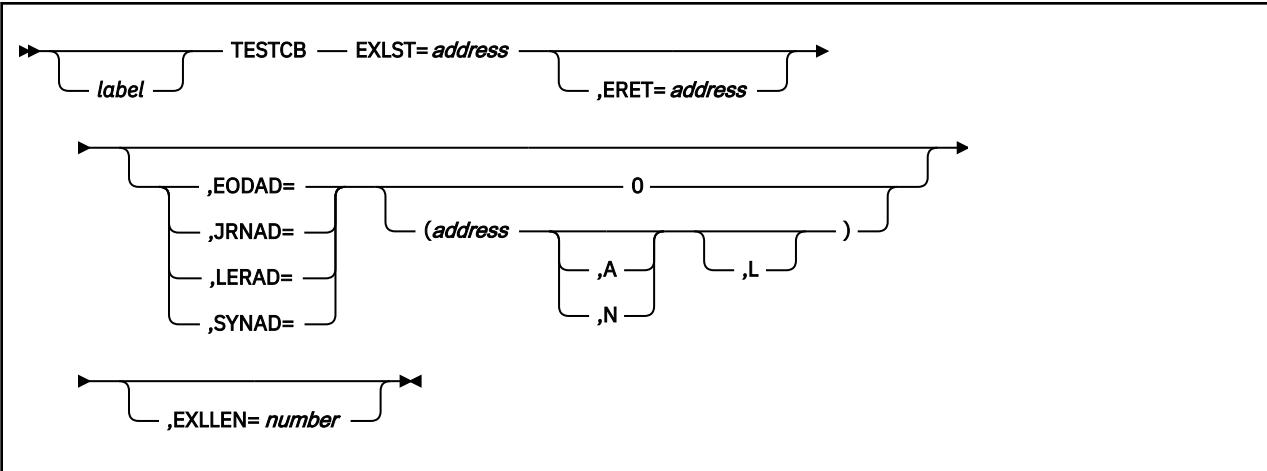
<b>Completion Code</b>	<b>Meaning</b>
0	Function completed successfully.
4	Function completed unsuccessfully.
8	You attempted to use the execute form of this macro to modify a keyword that is not in the parameter list.
12	The TESTCB macro was not executed because an error occurred while a VSAM module was being loaded.

When register 15 contains 4, then register 0 contains one of the following return codes.

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'01'	1	The type of request was invalid.
X'02'	2	The block type was invalid.
X'03'	3	One of the keywords in the parameter list is invalid.
X'04'	4	The block at the address you specified was not of the type you indicated.
X'05'	5	Either the file associated with the ACB in question is not open or is not a VSAM file.
X'06'	6	Index information was requested, but no index was opened for the file in question.
X'0E'	14	The MACRF or ATRB parameters contain incompatible options.
X'10'	16	You specified an invalid control block address in the ACB parameter.

<b>ABEND Code</b>	<b>Meaning</b>
03B	An invalid address was found in a VSAM control block or a VSAM parameter list. This means that your program tried to use an address to which it has no access.

### **Exit List Format**



## Purpose (EXLST)

During VSAM processing, unusual conditions sometimes occur. You can supply one or more exit routines to handle such conditions. You can then associate them with one or more access method control blocks (ACBs) that define the characteristics of the VSAM files you plan to process. See “[MODCB](#)” on page 453.

This discussion of the TESTCB macro deals only with those matters that involve GCS.

## Parameters (EXLST)

### **EXLST**

Specifies the address of the exit list whose information you want to test.

### **ERET**

Specifies the address of a routine that will receive control if the condition you want to test for cannot be tested.

This routine will receive control if the TESTCB macro places a return code of 4 in register 15. Upon entry to this routine, register 0 contains further information describing the error.

The ERET routine probably should issue an ABEND macro, because a failure to carry out a test is probably the result of a program logic error. If the ERET routine allows the program to continue, then it must transfer control to the continuation point, though it must not return to VSAM.

### **EODAD**

### **JRNAD**

### **LERAD**

### **SYNAD**

Specifies the exit routine about which you are asking a YES/NO question.

If you specify more than one operand following one of these parameters, each must equal the corresponding value in the exit list for you to receive an EQUAL CONDITION.

Because the same maximum length applies to every exit identifier, you can omit this parameter if you want to test the EXLLEN field.

The tests you can make are as follows:

### **0**

Test whether an entry is provided for the specified type of exit routine.

### **address**

Specifies a certain address in virtual storage.

If this parameter is specified by itself, it means test to see if this address is the address of the specified exit routine. Otherwise, it specifies the object address of the following test descriptions:

**A**

Test to see if the exit routine at the address specified is active.

**N**

Test to see if the exit routine at the address specified is inactive.

**L**

Test to see if the address specified is the address of an 8-byte field containing the name of the module containing the exit routine, rather than the entry point of the exit routine.

**EXLLEN**

Specifies one of two things:

- If you do not also specify the EXIT routine, then this parameter specifies the maximum length of an exit list.
- If you do specify the EXLST parameter, then this parameter specifies the actual length of the exit list.

**Usage (EXLST)**

1. You can use the TESTCB macro to test for only one attribute at a time. After the test, analyze the CONDITION CODE field of the PSW. It will indicate one of the following conditions:

- EQUAL TO
- GREATER THAN
- LESS THAN.

You can then proceed, based upon the condition.

2. Each time you enter the TESTCB macro, you must provide the system with a 72-byte save area. Before you enter the macro, place the address of this save area in register 13.
3. See [Appendix B, "Using VSAM," on page 517](#).

**Completion Codes, Return Codes, and ABEND Codes (EXLST)**

When this macro completes execution, it passes to the caller a completion code in register 15.

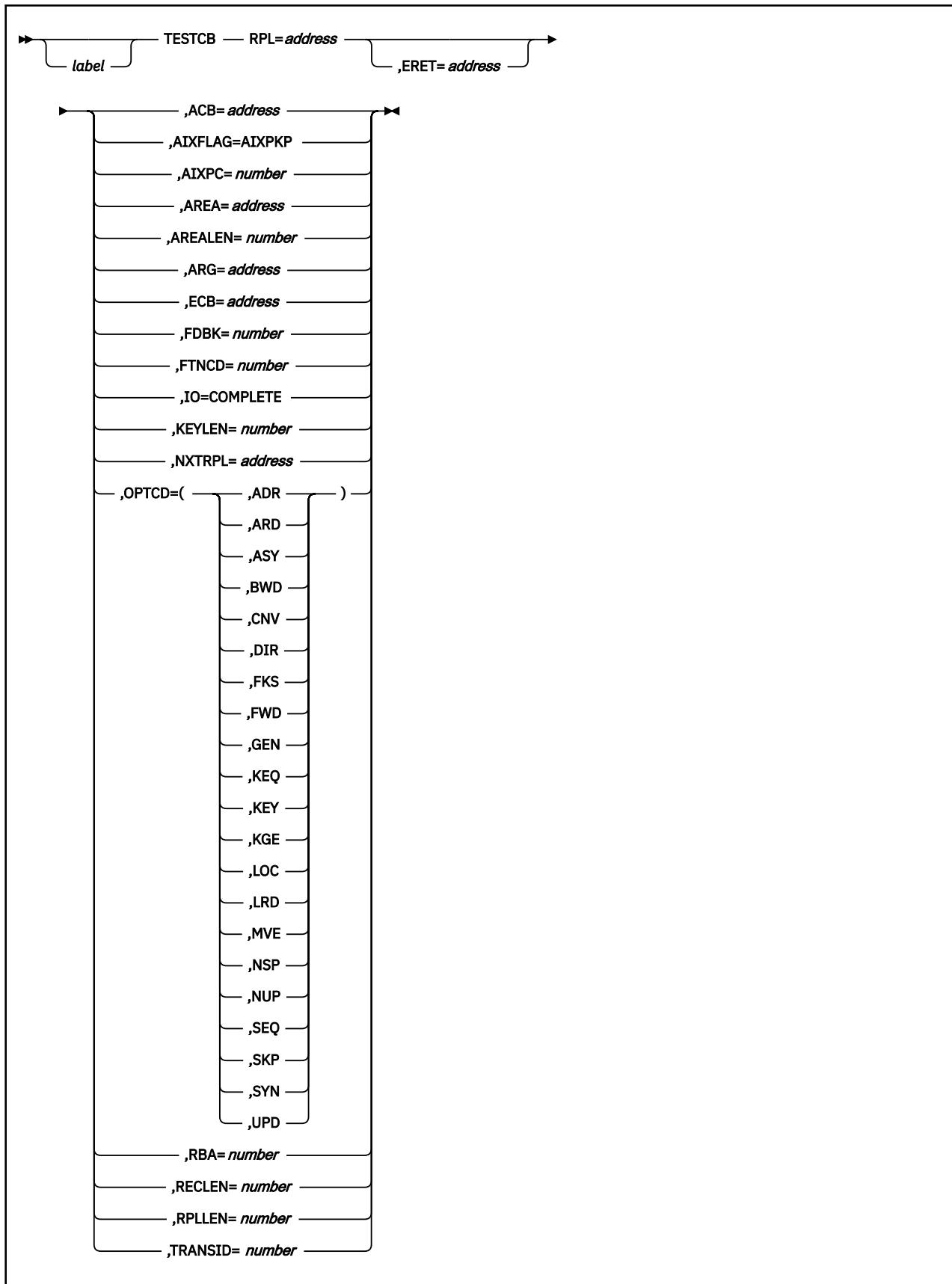
<b>Completion Code</b>	<b>Meaning</b>
0	Function completed successfully.
4	Function completed unsuccessfully.
8	You attempted to use the execute form of this macro to modify a keyword that is not in the parameter list.
12	The TESTCB macro was not executed because an error occurred while a VSAM module was being loaded.

When register 15 contains 4, then register 0 contains one of the following return codes.

<b>Hex Code</b>	<b>Decimal Code</b>	<b>Meaning</b>
X'01'	1	The type of request was invalid.
X'02'	2	The block type was invalid.
X'03'	3	One of the keywords in the parameter list is invalid.
X'04'	4	The block at the address you specified was not of the type you indicated.

Hex Code	Decimal Code	Meaning
X'10'	16	You specified an invalid control block address in the EXLST parameter.
ABEND Code	Meaning	
03B	An invalid address was found in a VSAM control block or a VSAM parameter list. This means that your program tried to use an address to which it has no access.	

## Request Parameter List Format



### Purpose (RPL)

All VSAM functions require that you set up a request parameter list (RPL) that describes the characteristics of your request. These VSAM functions are associated with the following macros: CHECK, ENDREQ, ERASE, GET, POINT, and PUT. See “[CHECK](#)” on page 425, “[ENDREQ](#)” on page 430, “[ERASE](#)” on page 432, “[GET](#)” on page 451, “[POINT](#)” on page 468, or “[PUT](#)” on page 470.

This discussion of the TESTCB macro deals only with those matters that involve GCS.

Use the TESTCB macro to test a certain field in a request parameter list.

## Parameters (RPL)

### RPL

Specifies the address of the RPL whose field you want to test.

Because all RPLs are the same length, you can omit this parameter if you are testing the RPLLEN field. Select from among the following parameters for other conditions to test:

### ACB

The address of the access method control block that relates the RPL to the file you are processing.

### AIXFLAG=AIXPKP

Indicates whether the alternate index just processed contains prime key pointers.

### AIXPC

The number of alternate index pointers.

### AREA

The address of the work area that your program uses to process the file records. Access to this file is defined by the RPL.

### AREALEN

The length of the work area whose address is specified in the AREA field.

### ARG

If you are using search arguments to process your file, the address of the field containing that search argument.

### ECB

The address of the event control block associated with the RPL in question. It is in this ECB that the completion of the request associated with the RPL is posted.

### ERET

Specifies the address of a routine that will receive control if the condition you want to test for cannot be tested.

This routine will receive control if the TESTCB macro places a return code of 4 in register 15. On entry to this routine, register 0 contains more information describing the error.

The ERET routine probably should issue an ABEND macro, because a failure to carry out a test probably is the result of a program logic error. If the ERET routine allows the program to continue, then it must transfer control to the continuation point, though it must not return to VSAM.

### FDBK

Specifies the return code from the request associated with this RPL.

For asynchronous requests, you must enter the CHECK macro to place the return code in this field.

See “[CHECK](#)” on page 425. The significance of this return code depends upon the contents of register 15, which indicates whether the request was successful or unsuccessful because of logical or physical error.

### FTNCD

The code that describes the function which a logical or physical error occurred. It indicates whether the upgrade set may have been modified incorrectly by the request.

### IO=COMPLETE

Specifies that a test will be made to determine whether an asynchronous request is complete.

Under GCS this test will always show that the request is not complete.

**KEYLEN**

If you are using a generic key as a search argument, the length of that argument.

**NXTRPL**

The address of the next request parameter list in the chain, if one exists.

**OPTCD**

Indicates what option or combination of options will be tested for. Select from among the following:

**ADR**

Indicates addressed access to a key-sequenced or entry-sequenced file.

RBAs will be used as search arguments, and sequential access is by entry sequence.

**ARD**

Indicates that the user's argument determines the record to be located, retrieved, or stored.

**ASY**

Specifies that you want your file processed asynchronously.

This means that when the request associated with the RPL you are creating is scheduled, control will return to your program so it can continue processing. Meanwhile, your request is being carried out.

Remember that asynchronous processing is merely simulated by GCS. Disk I/O in GCS is always synchronous.

**BWD**

Indicates that processing is to proceed through the file in a backward direction for keyed, addressed, sequential, or direct access.

This parameter is valid for POINT, GET, PUT, and ERASE operations. When you specify it, the KGE and GEN parameters are ignored, while the KEQ and FKS parameters are assumed, by default.

**CNV**

Indicates access will be to the entire contents of a control interval, rather than to an individual record.

**DIR**

Indicates direct access to a key-sequenced, entry-sequenced, or relative record file.

**FKS**

Indicates that you are providing a full key as a search argument.

**FWD**

Indicates that processing is to proceed through the file in a forward direction.

**GEN**

Indicates that you are providing a generic key as a search argument.

If you select this parameter, then you must also specify the length of the generic key in the KEYLEN parameter.

**KEQ**

Indicates that the key you provide as a search argument must equal the key or relative record number of the record.

You can use this parameter only if you also select the OPTCD=(KEY,DIR) or OPTCD=(KEY,SKP) parameter. This parameter is assumed by default for an RRDS, except when you enter the POINT macro.

**KEY**

Indicates access to a key-sequenced or relative record file.

Keys will be relative record numbers used as search arguments, and sequential access will be by key or relative record number.

**KGE**

Indicates that if the key you specify as a search argument does not equal a certain record, then the request will affect the record with the next highest key.

This parameter has the same restrictions and requirements as the KEQ parameter. For relative record processing, this parameter positions to the specified relative record, whether that slot is empty or not. If the relative record number is greater than the highest existing record, then the system returns the EOD. A following PUT macro will insert the record at this position.

**LOC**

Indicates that during retrieval, the record will be put in the I/O buffer to be processed.

This parameter is not valid if you intend to start the PUT or ERASE macros, though it is valid with the GET macro. However, to update the record, you must build a new version of it in a work area. Then, modify the RPL from LOCATE MODE to MOVE MODE before you enter any PUT macro. For keyed-sequential retrieval, modifying key fields in the I/O buffer may cause erroneous results in further GET requests until the record is reread.

**LRD**

Indicates that the last record in the file will be located or retrieved.

If you choose this parameter, then you must also choose the BWD parameter.

**MVE**

Indicates that, during retrieval, the record will be moved to a work area for processing. For storage, it will be moved from the work area to the I/O buffer.

**NSP**

Indicates that GCS is to remember the current position within the file for subsequent, sequential access.

Only the ENDREQ macro will cause the position to be forgotten.

**NUP**

Indicates that any record retrieved will not be updated or deleted. Moreover, any record that is stored is a new record.

On direct access requests, GCS does not remember the record's position.

**SEQ**

Indicates sequential access to a key-sequenced, entry-sequenced, or relative record file.

**SKP**

Indicates skip-sequential access to a key-sequenced or relative record file.

This is valid only with keyed access in a forward direction.

**SYN**

Specifies that you want your file processed synchronously.

This means that control will return to your program only after the request associated with the RPL you are creating has been carried out.

**UPD**

Indicates that any record retrieved can be updated or deleted.

On direct and sequential requests, GCS will remember the record's position.

**RBA**

The relative byte address of the most recently processed record in the file.

**RECLEN**

The length of the file record, access to which is defined by the request parameter list.

**RPLLEN**

The length, in bytes, of any request parameter list.

**TRANSID**

Specifies a number from 0 to 31 when RPL= is specified.

Number	Description
0	Default value. Indicates that the request defined by this RPL is not associated with other requests.
1-31	Relates the requests defined by this RPL to the requests defined by other RPLs with the same TRANSID value.

## Usage

1. You can use the TESTCB macro to test for only one attribute at a time. After the test, analyze the CONDITION CODE field of the PSW. It will indicate one of the following conditions:

- EQUAL TO
- GREATER THAN
- LESS THAN.

You can then proceed, based upon the condition.

2. Each time you enter the TESTCB macro, you must provide the system with a 72-byte save area. Before you enter the macro, place the address of this save area in register 13.
3. See [Appendix B, “Using VSAM,” on page 517](#).

## Completion Codes, Return Codes, and ABEND Codes (RPL)

When this macro completes execution, it passes to the caller a completion code in register 15.

Completion Code	Meaning
0	Function completed successfully.
4	Function completed unsuccessfully.
8	You attempted to use the execute form of this macro to modify a keyword that is not in the parameter list.
12	The TESTCB macro was not executed because an error occurred while a VSAM module was being loaded.

When register 15 contains 4, then register 0 contains one of the following return codes.

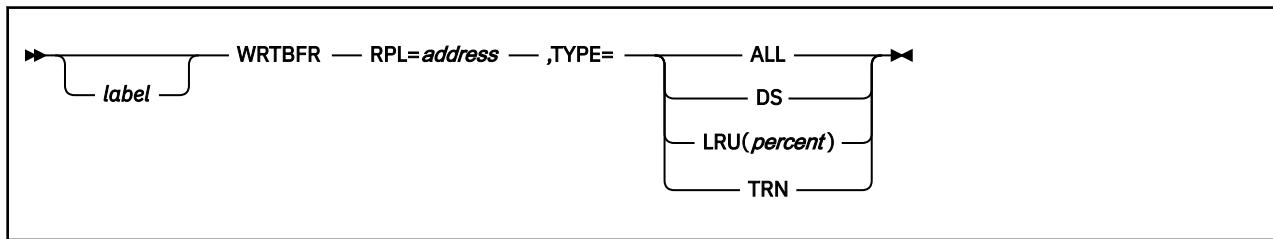
Hex Code	Decimal Code	Meaning
X'01'	1	The type of request was invalid.
X'02'	2	The block type was invalid.
X'03'	3	One of the keywords in the parameter list is invalid.
X'04'	4	The block at the address you specified was not of the type you indicated.
X'OE'	14	The MACRF or ATRB parameters contain incompatible options.
X'10'	16	You specified an invalid control block address in the RPL parameter.

ABEND Code	Meaning
03B	An invalid address was found in a VSAM control block or a VSAM parameter list. This means that your program tried to use an address to which it has no access.

# WRTBFR

---

## Format



## Purpose

Use the WRTBFR macro to manage I/O buffers in the following cases:

- Deferring writes for direct PUT requests, to reduce the number of I/O operations.
- Writing buffers that have been modified by related requests.
- Writing out buffers whose writing has been deferred.

## Parameters

### RPL

Specifies the address of the request parameter list which defines the WRTBFR request. It may be built for requests other than WRTBFR.

**Note:**

1. Only the ACB and the TRANSID operands of the RPL are significant for WRTBFR; all other RPL operands are ignored.
2. WRTBFR assumes that RPLs are not chained, unlike the other action macros (GET, PUT, and so forth).

### TYPE

Specifies what buffers are written.

### ALL

Specifies that all modified buffers in each buffer pool in the resource pool are written.

### DS

Specifies that all modified buffers are written for the file defined by the ACB to which RPL, associated with WRTBFR, is related.

### LRU(*percent*)

Specifies the percentage of the total number of buffers in each buffer pool in the resource pool that are examined for possible writing. The least recently used buffers are examined.

**Note:** To ensure that buffers are always available for GET or PUT requests, you can periodically force out the least recently used part of each buffer pool through the LRU option.

### TRN

Specifies that all buffers that were modified by requests with the same TRANSID are written. The TRANSID for the buffer must be the same as the one specified in the RPL associated with WRTBFR. TRANSDIDs are then disassociated with these buffers.

## Usage

When using this macro you must make sure that register 13 contains the address of a 72-byte save area. If you enter the macro from within one of your exit routines (LERAD or SYNAD) you must provide a second 72-byte save area because the original one is still in use by the external VSAM routine.

## Return Codes and ABEND Codes

The feedback error codes result from GET or PUT when operating in the LSR/DFR environment. To find the specific error code, you can use the SHOWCB macro to obtain the FDBK field in the RPL.

### When the return code in register 15 is 0:

Feedback Error Code	Meaning
12(C)	The last request macro detected the condition that there are no more unmodified buffers to read the contents of a control interval into. You must either use the WRTBFR macro thereby freeing up some buffers, or accept the delay required by VSAM to do an implicit buffer write before servicing your next request.

### When the return code in register 15 is 8:

Feedback Error Code	Meaning
64(40)	Your request could not be started because there are already as many requests active as the number specified in the STRNO operand for the ACB or BLDVRP macro.
104(68)	For LSR, <ul style="list-style-type: none"> <li>• The ACB address is not the same as for a previous request that used the same RPL.</li> <li>• When WRTBFR was issued the: <ul style="list-style-type: none"> <li>– TRANSID was greater than 31, or</li> <li>– Shared resource option (LSR) was not specified, or</li> <li>– LRU percentage was not between 0 and 100.</li> </ul> </li> </ul>

ABEND Code	Meaning
03B	You specified a TYPE parameter of CHK or DRBA and those parameters are not supported.

# Appendix A. Tailoring and Building the GCS Nucleus

This appendix describes how to change GCS nucleus options, change default definitions, and rebuild and save the GCS nucleus.

## Changing GCS Nucleus Options

If you want to modify your GCS nucleus options, such as relocating your GCS named saved system, changing its size, changing its name, or adding multiple GCS systems, you would alter your GCS nucleus build list.

IBM supplies a single GCS nucleus build list with a name of GCTLOAD EXEC. An IBM-shipped service file for GCTLOAD would have a file type EXCnnnnn, where *nnnnn* is the five digit PTF number used by service.

The default name for the configuration file is GCS ASSEMBLE. The name of the configuration file must be the same as the name of the saved system.

The following are possible reasons for making modifications to GCTLOAD:

- Change the name of your GCS system. The IBM-supplied GCS nucleus build list, GCTLOAD, has the file name of the configuration file coded as GCS. To use another file name, you must change the file name in the GCS nucleus build list to match the file name of the configuration file you wish to use.
- Add multiple GCS systems. For each additional GCS system, you must:
  1. Refer to “[Creating a New GCS Nucleus Build List](#)” on page 507 to create a new build list, update the build list with the new GCS system name, and create a PPF override to make any new build lists known to VMSES/E.
  2. Refer to “[Rebuilding and Saving the GCS Nucleus](#)” on page 513 to create a unique configuration file with the GROUP exec and to rebuild the GCS nucleus with the new PPF you created. Remember the file name of the configuration file must match the new GCS system name specified in the new build list created.
- Change the size or location of your GCS system. You can accomplish this by changing the Set Location Counter (SLC) values in the GCS nucleus build list and creating SLC Lnnnnnn files for these new values.

IBM recommends that you do not make modifications directly to GCTLOAD because IBM may provide service for it. Instead, you can create a new build list(s) by copying the latest IBM-supplied version of GCTLOAD. Choose any file name for your new build list or lists. In addition, create a PPF override to make any new build list(s) known to VMSES/E.

## Creating a New GCS Nucleus Build List

The following is an example of how to create a new GCS nucleus build list:

1. Log on to the MAINTvrm user ID. The default password for MAINTvrm is MAINTvrm.

```
logon maintvrm
```

2. Establish the correct minidisk access order. Use the GCS operand if you loaded GCS to minidisks, the GCSSFS operand if you moved it to SFS directories. **Make a note of the file mode assigned to the 6C4 and 51D minidisks.**

**Attention:** If you loaded GCS to minidisks, you must also build and save GCS on minidisks. You must continue to access and service GCS on minidisks. If you moved GCS to Shared File System directories, you **must** continue to access and service GCS on the Shared File System directories.

```
vmfsetup servp2p {gcs|gcssfs}
```

## Changing GCS Nucleus Options

3. Determine the latest IBM-supplied service level for GCTLOAD. Use that level to make local modifications.

```
vmfsm getlvl servp2p {gcs|gcssfs} tdata :part gctload exc  
GCTLOAD EXC00000 BASE-FILETYPE or GCTLOAD EXCnnnnn.
```

If BASE-FILETYPE is returned in the system response, then it has not been serviced and the file type will be EXEC.

4. Identify the file mode (*fm*) of the disk containing the latest IBM-supplied service level of GCTLOAD. Copy that version of GCTLOAD to the GCS LOCALMOD disk (6C4) in preparation for modification.

```
listfile gctload {exec|excnnnnn.} * (date  
copyfile gctload {exec|excnnnnn.} fm bln excl0001 fm-6c4
```

*fm* is the file mode returned from the previous LISTFILE command. *bln* is the build list name of your choice. GCTLOAD is the default. *fm-6C4* is the file mode of the GCS LOCALMOD disk.

5. Update the local Version Vector Table enabling VMSES/E to identify the local modification file.

```
vmfsm logmod 7vmpgcs40 vvtlgct fm-6c4 tdata :part bln exc :mod lcl0001
```

**7vmpgcs40 vvtlgct** is the file name and type of the GCS local Version Vector Table. *fm-6c4* is the file mode of the GCS LOCALMOD disk (6C4). **lcl0001** is the local modification ID.

6. If you wish to change the name of your GCS saved system, you must change the file name of the configuration file in the new GCS nucleus build list so the two names match.

```
xedit bln excl0001 fm-6c4
```

*bln* is the name you chose for the new GCS nucleus build list.

```
====> locate /&3 GCS/  
====> change /GCS/systemname/
```

*systemname* is the file name of the saved system and the configuration file. If you plan to use the GROUP EXEC to create the configuration file, use the same system name with which you invoke the GROUP EXEC.

```
====> file
```

**Note:** Now you are ready to substitute your GCS system name whenever you see *systemname* in the following procedures. You must also substitute the name of your new build list whenever you see *loadlistname* or *bln*.

7. If you wish to change the location where the low common storage portion of GCS is loaded into virtual storage change the SLC values in the new build list.

**Note:** SLC values must be on megabyte boundaries. Valid SLC values for low common storage are 1MB to 16MB (address location X'100000' to X'1000000').

- a. The following sample procedure shows how to increase the size of a GCS named saved system from 2M to 3M and how to move the named saved system from X'400000' to X'800000'.

```
xedit bln excl0001 fm-6c4
```

GCTLOAD EXEC is the name of the GCS build list. *fm-6c4* is the file mode of the LOCALMOD disk (6C4).

```
====> set case upper  
====> top  
====> locate /** GCTALP/  
====> up 1  
====> change /SLC L400000/SLC L800000/1 1
```

- SLC L4000000 is the CMS file containing the IBM default starting address of GCS low common storage. Note that the file type contains the address.

SLC L8000000 is a CMS file containing the new starting address of low common storage. You will create this file in substep “[7.b](#)” on page 509.

```
=====> top
=====> locate /GCTZET/
=====> up 1
=====> change /SLC L6000000/SLC LB000000/1 1
```

- SLC L6000000 is the CMS file containing the IBM default ending address of GCS low common storage. Note that the file type contains the address.

SLC LB000000 is a CMS file containing the new ending address. You will create this file in the following substep.

```
=====> file
```

- Create two new SLC files to match the new loadlist:

```
xedit SLC L8000000 fm-6c4
```

*fm-6c4* is the GCS local modification minidisk.

```
=====> input $SLC 800000
=====> set hex on
=====> change /$/X'02'/
=====> file
```

There must be two blanks between SLC and the address. X'02' is an unprintable loader control character.

```
xedit SLC LB000000 fm-6c4
```

*fm-6c4* is the GCS local modification minidisk.

```
=====> input $SLC B00000
=====> set hex on
=====> change /$/X'02'/
=====> file
```

There must be two blanks between SLC and the address. X'02' is an unprintable loader control character.

- If you wish to change the location where the high common storage portion of GCS is loaded into virtual storage change the SLC values in the new build list.

**Note:** SLC values must be on megabyte boundaries. The default SLC values for high common storage are 16MB to 18MB (address location X'1000000' to X'1200000').

- The following sample procedure shows how to move the named saved system from X'1000000' to X'1300000'.

```
xedit bln excl0001 fm-6c4
```

GCTLOAD EXEC is the name of the GCS build list. *fm-6c4* is the file mode of the LOCALMOD disk (6C4).

```
=====> set case upper
=====> top
=====> locate /GCTBHC/
=====> up 1
=====> change /SLC L1000000/SLC L1300000/1 1
```

- SLC L1000000 is the CMS file containing the IBM default starting address of GCS high common storage. Note that the file type contains the address.

## Changing GCS Nucleus Options

SLC L1300000 is a CMS file containing the new starting address of high common storage. You will create this file in substep “8.b” on page 510.

```
====> top  
====> locate /*** GCTEHC/  
====> up 1  
====> change /SLC L1200000/SLC L1500000/1 1  
====> file
```

- SLC L1200000 is the CMS file containing the IBM default ending address of GCS high common storage. Note that the file type contains the address.

SLC L1500000 is a CMS file containing the new ending address. You will create this file in the following substep.

- Create two new SLC files to match the new loadlist:

```
xedit SLC L1300000 fm-6c4
```

*fm-6c4* is the GCS local modification minidisk.

```
====> input $SLC 1300000  
====> set hex on  
====> change /$/X'02'/  
====> file
```

There must be two blanks between SLC and the address. X'02' is an unprintable loader control character.

```
xedit SLC L1500000 fm-6c4
```

*fm-6c4* is the GCS local modification minidisk.

```
====> input $SLC 1500000  
====> set hex on  
====> change /$/X'02'/  
====> file
```

There must be two blanks between SLC and the address. X'02' is an unprintable loader control character.

- If you changed the build list name or added a new build list, create a PPF override file to add your build list name to the PPF build section (:BLD.). The GCSPPF SAMPLE file is shipped on the 6B2 minidisk or VMPSFS:MAINTvrn.GCS.OBJECT. Copy GCSPPF SAMPLE from the 6B2 minidisk to the 51D minidisk, then make the appropriate changes.

- The following is the contents of GCSPPF SAMPLE:

```
*****  
*  
*   Override $PPF to use modified GCS nucleus build lists.  
*  
*****  
* Start of Product Header  
*:OVERLST. GCS GCSSFS  
*  
* End of Product Header  
*:GCS. GCS 7VMGCS40  
:BLD. UPDATE  
.INSERT GCTLOAD AFTER  
b\ln VMFBDNUC BUILD7 TXT TXS * Build modified GCS nucleus  
.END  
.END.  
:GCSSFS. GCSSFS 7VMGCS40  
:BLD. UPDATE  
.INSERT GCTLOAD AFTER  
b\ln VMFBDNUC BUILD7 TXT TXS * Build modified GCS nucleus
```

```
./END
:END.
```

*bln* is the changed name of the build list or the name of the new build list.

- Copy the sample file over to the 51D disk using the new file name you selected for your PPF override file and the file type of \$PPF. You can then add or replace the build list name.

```
copyfile gcsppf sample fm-6b2 ppfovername $ppf fm-51d
```

*ppfovername* is the name you chose for your PPF override file. *fm-51d* is the minidisk (51D) where PPF files reside.

- Edit the PPF override file you just created.

```
xedit ppfovername $ppf fm-51d
```

*ppfovername* is the name you chose for your PPF override file. *fm-51d* is the minidisk (51D) where PPF files reside.

- Change the two occurrences of *bln* shown in GCSPPF SAMPLE to the name of your new build list.
- If you do not want to use the GCTLOAD build list, add the following statement **before** both ./INSERT GCTLOAD AFTER statements shown in GCSPPF SAMPLE:

```
GCTLOAD -VMFBDNUC BUILD7 TXT TXS * Build GCS nucleus
```

- To add another build list, then for each new build list, add the following three lines **after** the two ./END statements shown in GCSPPF SAMPLE, and change the two occurrences of *bln* to the name of your new build list.

```
./INSERT GCTLOAD AFTER
bln VMFBDNUC BUILD7 TXT TXS * Build modified GCS nucleus
./END
```

- Save the information you just made on your A-disk.

```
====> file
```

#### 10. Finish creating the PPF override file.

```
vmfppf ppfovername {gcs|gcssfs}
```

Use **gcs** as the component name if you loaded GCS to minidisks and **gcssfs** if you moved GCS to SFS directories.

```
copyfile ppfovername ppf a = d2 (olddate
erase ppfovername ppf a
```

**Note:** You will use this PPF override file when you rebuild the GCS nucleus.

- If you are building this system as a restricted system, you need to update the directory file in this substep. A single user environment is **never** restricted. (If you want to look at the GROUP EXEC panel default values, refer back to the panels shown at substep “2” on page 512.)

To IPL your GCS system, you need to add an entry to the user directory for each authorized user. When a system is restricted, only those users whose directory entries contain a NAMESAVE statement specifying the GCS system name are allowed to IPL the named saved system. The IBM supplied directory contains an NAMESAVE GCS entry for the user IDs AVSVM, GCS, and MAINTvrm.

- Establish the required minidisk order.

```
vmfsetup servp2p cp
```

- Edit the USER DIRECT file.

```
xedit user direct fm-2c2
```

## Changing GCS Default Definitions

- c. Locate each user ID, *userid*, you want to authorize and add the NAMESAVE statement, NAMESAVE *systemname*.

```
====> locate /USER userid/  
====> file
```

- d. Bring the new directory online, and reestablish the correct minidisk order.

```
directxa user direct
```

If you wish to change the GCS default definitions, continue to “[Changing GCS Default Definitions](#)” on [page 512](#). Otherwise, go to “[Rebuilding and Saving the GCS Nucleus](#)” on [page 513](#) to rebuild the GCS nucleus.

## Changing GCS Default Definitions

This section describes how to change the GCS default definitions supplied by IBM. It involves running the GROUP command to redefine your GCS System through the configuration file, and then reassembling the GCS configuration file.

1. Establish the correct minidisk order. Use the GCS operand if you left GCS on minidisks, or the GCSSFS operand if you moved GCS to SFS directories. **Make a note of the file mode assigned to the LOCALMOD disk (6C4).**

**Attention:** If you left GCS on minidisks, you must also build and save GCS on minidisks. You **must** continue to access and service GCS on minidisks. If you moved GCS to Shared File System directories, you **must** continue to access and service GCS on the Shared File System.

```
vmfsetup servp2p {gcs|gcssfs}
```

- **The following console is for VMFSETUP SERVP2P GCS only. If you install on Shared File System directories, the minidisk names below will be substituted with directory addresses.**

```
VMFSET2760I VMFSETUP processing started  
VMFUTL2205I Minidisk|Directory Assignments:  
          String   Mode   Stat   Vdev   Label/Directory  
VMFUTL2205I LOCALMOD   E     R/W    6C4    MNT6C4  
VMFUTL2205I LOCALMD2   F     R/W    3C4    MNT3C4  
VMFUTL2205I LOCALSAM   G     R/W    6C2    MNT6C2  
VMFUTL2205I APPLY      H     R/W    6A6    MNT6A6  
VMFUTL2205I           I     R/W    6A4    MNT6A4  
VMFUTL2205I           J     R/W    6A2    MNT6A2  
VMFUTL2205I APPLY2     K     R/W    3A6    MNT3A6  
VMFUTL2205I           L     R/W    3A4    MNT3A4  
VMFUTL2205I           M     R/W    3A2    MNT3A2  
VMFUTL2205I DELTA      N     R/W    6D2    MNT6D2  
VMFUTL2205I DELTA2     O     R/W    3D2    MNT3D2  
VMFUTL2205I BUILD7     P     R/W    493    MNT493  
VMFUTL2205I BUILD5     Q     R/W    19D    MNT19D  
VMFUTL2205I BUILD2     R     R/W    193    MNT193  
VMFUTL2205I BASE2      T     R/W    6B2    MNT6B2  
VMFUTL2205I BASE4      U     R/W    3B2    MNT3B2  
VMFUTL2205I -----     A     R/W    191    MNT191  
VMFUTL2205I -----     B     R/W    5E5    MNT5E5  
VMFUTL2205I -----     D     R/W    51D    MNT51D  
VMFUTL2205I -----     S     R/O    190    MNT190  
VMFUTL2205I -----     Y/S   R/O    19E    MNT19E  
VMFSET2760I VMFSETUP processing completed successfully
```

2. Enter the GROUP command to display the configuration panels.

```
group systemname
```

- This command assigns *systemname* as the file name of the GCS configuration file that you are creating and invokes the Primary Option Menu. *systemname* is either the IBM-supplied system name (GCS) or the changed name you specified.

**Note:**

- a. If you are using a printer-keyboard ("line-mode") terminal instead of a full-screen display device, you cannot use the GROUP command, because you cannot display the panels. You must build the configuration file manually using the build macros.
- b. GROUP uses a GCS message repository that is available only in American mixed case English (AMENG) and uppercase English (UCENG). If your system default national language is other than AMENG, GROUP uses the UCENG message repository.
- c. The IBM default name for the GCS configuration file is GCS, and the file type is ASSEMBLE. Therefore *systemname* is GCS, and the defaults listed at the beginning of this step display on the GROUP panels (unless the GCS GROUP file is erased).
- d. If you are changing the *systemname*, you should have already completed “[Changing GCS Nucleus Options](#)” on page 507.

Refer to “[Function Keys](#)” on page 105 for guidance on PF key functions as you move through the panels.

If you specify a system name here, the Primary Option Menu appears with the system name filled in. If you do not specify a system name here, then the Primary Option Menu panel appears with the SYSTEM NAME field filled in with the IBM-supplied default name of **GCS**.

Refer to “[GROUP Panels](#)” on page 103 for a description of the GROUP panels.

3. Prepare the file for assembly and copy the file to the GCS LOCALMOD minidisk (6C4).

```
copyfile systemname group a = assemble fm-6c4 (replace olddate
copyfile systemname group a == fm-6c4 (replace olddate
erase systemname group a
```

*systemname* is either the IBM-supplied system name (GCS) or the changed name you specified.

4. Assemble the GCS configuration file.

```
vmfasm systemname servp2p {gcs|gcssfs} (outmode localmod
```

- *systemname* is either the IBM-supplied system name (GCS), or the changed name you specified. Use the **gcs** operand if you loaded GCS to minidisks, the **gcssfs** operand if you moved GCS to SFS directories.

The **outmode localmod** options place the updated text file on the GCS LOCALMOD minidisk (6C4).

**Note:** If you recreate the configuration file, you must enter the GROUP command with the same system name, and change the information brought up on those panels to the correct values. Then perform steps “3” on page 513 and “4” on page 513 to prepare the file for assembly and to reassemble the configuration file.

Continue to “[Rebuilding and Saving the GCS Nucleus](#)” on page 513.

## Rebuilding and Saving the GCS Nucleus

This section describes how to rebuild and save the GCS nucleus.

1. Spool the output of your virtual punch and printer to your own virtual reader.

```
spool punch *
spool print *
```

2. Ensure that your virtual storage is defined at least 1MB larger than the value you defined for HIGH COMMON END in the procedure defined in “[Changing GCS Nucleus Options](#)” on page 507, substep “7” on page 508. If you did not change the default, high common end storage is 18MB.

```
query virtual storage
```

## Rebuilding and Saving the GCS Nucleus

If you have less than 20MB of storage, issue the following DEFINE command:

```
define storage 20m  
ipl 190 clear
```

### 3. Build and save the GCS nucleus.

**Attention:** If you left GCS on minidisks, you must also build and save GCS on minidisks. You **must** continue to access and service GCS on minidisks. If you moved GCS to Shared File System directories, you **must** continue to access and service GCS on a Shared File System.

```
vmfbld ppf ppfname compname bln (all setup)
```

- *ppfname* is set to one of the following:

**servp2p**

To build GCS in mixed-case English (AMENG)

**uceng**

To build GCS in uppercase English (UCENG)

**ppfovername**

The PPF name created in [“Changing GCS Nucleus Options” on page 507](#).

- *compname* is set to one of the following:

**gcs**

GCS is loaded on minidisks

**gcssfs**

GCS was moved to SFS directories.

- *buillistname* is set to one of the following:

**gctload**

If you did not change your system name

**name**

If you did change your system name, the name of the new build list you created in [“Changing GCS Nucleus Options” on page 507](#).

```
VMFBLD2185R The following source product parameter files have been  
serviced:  
VMFBLD2185R 7VMGCS40 $PPF  
VMFBLD2185R When source product parameter files are serviced,  
all product parameter files built from them must be  
recompiled using VMFPPF before VMFBLD can be run  
VMFBLD2185R Enter zero (0) to have the serviced source product  
parameter files built to your A-disk and exit VMFBLD  
so you can recompile your product parameter files  
with VMFPPF  
VMFBLD2185R Enter one (1) to continue only if you have already  
recompiled your product parameter files with VMFPPF
```

```
0  
vmfppf ppfname compname  
copyfile 7vmgcs40 $ppf a = = fm-51d (olddate replace  
erase 7vmgcs40 $ppf a  
vmfsetup ppfname compname
```

Reissue the VMFBLD command to build the GCS nucleus.

```
vmfbld ppf ppfname compname bln (all setup)  
VMFBLD2185R Enter zero (0) to have the serviced soure product  
parameter files built to your A-disk and exit VMFBLD  
so you can recompile your product parameter files  
with VMFPPF  
VMFBLS2185R Enter one (1) to continue only if you have already  
recompiled your product parameter files with VMFPPF  
Enter :pk.1:epk. to continue
```

4. Verify that the GCS nucleus is in MAINTvrm's virtual reader. Make a note of the file number (*fileno*) of the GCS nucleus file (\$\$\$TLL\$\$ IPL) that you will IPL in substep “7” on page 515.

```
query rdr * all
```

ORIGINID	FILE	CLASS	RECORDS	CPY	HOLD	DATE	TIME	NAME	TYPE	DIST
MAINTvrm	<i>fileno</i>	A	PUN	nnnnnnnn	001	USER	mm/dd hh:mm:ss	\$\$\$TLL\$\$	IPL	SYSROG

5. If the GCS nucleus is not the first file in your reader, order your reader so that the GCS nucleus will be processed first.

```
order rdr fileno
```

*fileno* is the file number of the GCS nucleus.

6. Change the nucleus reader file status to ensure it remains in the reader after you IPL.

```
change rdr fileno keep
```

7. IPL MAINTvrm's virtual reader to save the GCS new nucleus.

```
ipl 00c clear
```

```
RDR FILE mfileno SENT FROM MAINTvrm PRT WAS mfileno
RECS nnnn. CPY 001 A NOHOLD NOKEEP
Storage cleared - system reset.
```

*mfileno* is the spool file number of the GCS load map.

8. IPL your System disk (190).

```
ipl 190 clear
```

Purge the GCS nucleus from your reader to save spool space.

```
purge rdr fileno
```

*fileno* is the file number of the GCS nucleus identified in substep “4” on page 515.

9. Receive the GCS load map file from the virtual reader to the alternate TOOLS disk (493).

```
access 493 e
receive mfileno fn ft e (replace
DMSRDC738I Record length is nnn bytes
fn ft E1 created
File fn ft E1 received from MAINTvrm at * sent as (none) (none) A1
```

- *mfileno* is the file identifier of the GCS load map you noted in substep “7” on page 515. You may give the GCS load map any file name and file type you like. The GCS load map on the System DDR is GCSNUC MAP. You may want to erase the old version to save space.

The GCS load map is loaded onto the alternate SYSTEM TOOLS disk (493).

10. If you wish, you can now print a copy of the GCS load map.

11. Examine the load map for unresolved symbols. Unresolved symbols may indicate an error. Make sure that you understand the reason for any unresolved symbols you find before going on.

```
xedit fn ft e
====> locate /UNRESOLVED/
====> qquit
```

12. If you wish, you can now pack the GCS load map to save minidisk space.

```
copyfile fn ft e (olddate pack
```

13. Copy the GCS load map to the production tools disk (193).

## Rebuilding and Saving the GCS Nucleus

```
access 193 f
copyfile fn ft e = = f (olddate replace
```

14. Release the TOOLS minidisks (493 and 193).

```
release e
release f
```

# Appendix B. Using VSAM

## VSAM I/O Operations under GCS

GCS applications can access VSAM disks using the VSAM interface. This interface contains the macros described in [Chapter 7, “VSAM Data Management Service Macros,” on page 417](#).

This VSAM interface, as supported by GCS, is very like that supported by CMS. Of particular significance is that VSAM disks are in VSE/VSAM format. MVS/VSAM requests are mapped to VSE/VSAM requests and executed through VSE/VSAM code.

VSAM DASD space can be saved by using the Data Compression feature of VSE/VSAM. You can compress or expand data automatically when you DEFINE a VSAM cluster as compressed. The COMPRESS and NOCOMPRESS modifiers will let VSAM know whether data is to be automatically converted by VSAM when a VSAM I/O request is processed. All existing applications can remain unchanged. For more information on VSAM Data Compression, see *VSE/VSAM Version 6 Release 1 Commands* and *VSE/VSAM Version 6 Release 1 User’s Guide and Application Programming*.

The macros described in [Chapter 7, “VSAM Data Management Service Macros,” on page 417](#) are up to the MVS Release 3.8 level and are contained in a CMS macro library named OSVSAM MACLIB. In addition, GCS includes MVS Release 3.8 levels of the OS OPEN, CLOSE, GET, and PUT macros to support access method control blocks (ACBs).

GCS must map macro requests to VSE/VSAM before invoking the VSE/VSAM code, OS mapping macros affecting access method control blocks, exit lists, and request parameter lists (RPLs) are not supported. After the first macro call, these data structures are converted from the OS format to the VSE/VSAM format. Hence, GCS provides the TESTCB and SHOWCB macros to test and examine the contents of these data structures.

Using the RPL macro, you can specify that you want your file processed asynchronously. This means that when the request associated with the RPL you are creating is scheduled, control will return to your program so it can continue processing. Meanwhile, your request is being carried out. Remember, though, that asynchronous processing is merely simulated by GCS. Disk I/O in GCS is always synchronous.

GCS does not support utility functions, such as disk initialization, catalog definition, and file definition. These AMS functions must be performed under CMS.

GCS supports an OS/MVS macro interface, VSAM operations are performed by the VSE/VSAM licensed product. And, although you can use the macros discussed in this section only for VSAM, many of them are also used for VTAM, QSAM, and BSAM—but, in other environments.

GCS supports Local Shared Resources (LSR) and Deferred Write (DFR) to enhance synchronous VM/VSAM processing. LSR allows the sharing of buffers, I/O control blocks, and channel programs among several VSAM files within a virtual machine. It also allows deferred writing. Sharing resources among files optimizes their use and reduces the working set for the virtual machine. DFR allows an application to have more control over physical writing of buffers. Macro interfaces for LSR/DFR include BLDVRP, DLVRP, SHOWCAT, and WRTBFR.

Several storage management subpools are used during the execution of a VSAM request. (See the GETMAIN macro for a description of various subpool numbers.) The storage may be obtained and released during the execution of a single module or a single task; or it may be held from the time the master catalog is opened, because the first data set being opened, until the time the master catalog is closed, because the last data set being closed. So, you are not responsible for releasing any storage obtained through a VSE/VSAM function. For example, storage is obtained from subpool 230 during the execution of a GENCB and an attempt to release the storage by a program in the wrong key may result in an abend.

## Control-Block Manipulation Macros

The GENCB, MODCB, SHOWCB, and TESTCB macros are control-block manipulation instructions. The list, list address, and execute formats of these macro instructions allow you to save virtual storage by using one parameter list for two or more macro invocations. You can also make your program reentrantable, that is, executable by more than one task at a time. However, while the generate format of these macros enables you to make programs reentrantable, it does not allow shared parameter lists.

### VSAM Macro Addresses

VSAM Data Management Services under GCS is provided only below the 16MB line. All addresses (parameter list or pointers) in all VSAM macros have to be below the line in order to work.

### List Format

The list format of the GENCB, MODCB, SHOWCB, and TESTCB macros has the same parameters as the standard form, except that you add the

MF= L

parameter. The parameter list of the macro is created in-line when you code MF=L. Therefore, your program is not reentrant if the parameter list is modified at execution time. And, because the expansion of the list format of a macro does not include executable code, you cannot use register notation or expressions that generate S-type constants.

### List Address Format

When you add the

MF=(L, address, label)

parameter, you create the parameter list in the area specified by ADDRESS. This version is reentrantable, and you must supply the area through the GETMAIN macro when your program is executed. See “[GETMAIN](#)” on page 257. You can determine the size of the parameter list by coding the third parameter LABEL. VSAM equates this parameter with the length of the parameter list.

### Execute Format

The execute format produces code that executes the function. The execute format is identical with the standard format, except that you add the

MF=(E, address)

parameter. ADDRESS points to the parameter list created by the list format of the instruction. All other parameters of the instructions are optional. Code them only if you wish to change entries in the parameter list before it is used. However, you cannot use the execute format to add or delete entries from the parameter list or to change the type of list.

### Generate Format

The generate format of these macros builds the parameter list in a remote area, the address of which you specify, and passes it to VSAM for execution. It lets you make your program reentrantable, but it does not create shared parameter lists. The generate format is the same as the standard format, except that you add the

MF=(G, address, label)

parameter. The parameter list is created in an area pointed to by ADDRESS. To make it possible for the parameter list to be reentrantable, you should code ADDRESS using register notation. You must obtain this area through the GETMAIN macro when the program is executed. You can determine the size of

the parameter list by coding the third parameter LABEL. VSAM equates LABEL with the length of the parameter list.

## Parameter Notation for GENCB, MODCB, SHOWCB, and TESTCB Macros

The addresses, names, numbers, and options required with parameters in the GENCB, MODCB, SHOWCB, and TESTCB macros can be expressed in a variety of ways:

- As an absolute numeric expression:

```
CPIES=10
```

- As a character string:

```
DDNAME=DATASET
```

- As a code or a list of codes separated by commas and enclosed in parentheses:

```
OPTCD=(KEY,DIR,IN)
```

- An expression valid for a relocatable A-type address constant:

```
AREA=MYAREA+4
```

- As a register from 2 through 12 that contains an address or numeric value:

```
SYNAD=(3)
```

Equated labels can be used to designate a register:

```
ERR EQU 3
.
.
.
....SYNAD=(ERR)
```

- As an expression of the format:

```
(S,SCON)
```

SCON is an expression valid for an S-type address constant, including the base-displacement form. The contents of the base register will be added to the displacement to obtain the value of the keyword. For example, if the value of the keyword being represented is a numeric value (that is, CPIES, LENGTH, RECLEN), then the contents of the base register will be added to the displacement to determine the numeric value. If the value of the keyword being represented is an address constant (that is, WAREA, EXLST, EODAD, ACB), then the contents of the base register will be added to the displacement to determine the value of the address constant.

- As an expression of the format

```
(*,scon)
```

SCON is an expression valid for an S-type address constant, including the base-displacement form. The address specified by SCON is indirect, that is, it is the address of an area that contains the value of the keyword. The contents of the base register will be added to the displacement to determine the address of the fullword of storage that contains the value of the keyword.

If you use an indirect S-type address constant, then the value it points to must meet the following criteria:

- If it is a numeric quantity or an address, then it must occupy a fullword of storage.
- If it is an alphanumeric character string, then it must occupy two words of storage, be left justified, and be padded on the right with blanks.

The expressions that you can use depend on the keyword you specify. Register and S-type address constants cannot be used when you code MF=L.

## Using VSAM

The tables that follow summarize the manner in which the keyword parameters of VSAM control block manipulation macros can be expressed.

### GENCB Macro

Keyword	Absolute Numeric	Character Code	Character String	Register	S-Type Address	Indirect S- Type Address	A-Type Address
AM			X				
BLK			X				
COPIES	X			X	X	X	
LENGTH	X			X	X	X	
WAREA				X	X	X	X
<b>BLK=ACB:</b>							
BUFND	X			X	X	X	
BUFNI	X			X	X	X	
BUFSP	X			X	X	X	
DDNAME			X			X	
EXLST				X	X	X	X
MACRF		X					
MAREA				X	X	X	X
MLEN	X			X	X	X	
PASSWD				X	X	X	X
STRNO	X			X	X	X	
<b>BLK=EXLST:</b>							
EODAD				X	X	X	X
JRNAD				X	X	X	X
LERAD				X	X	X	X
SYNAD				X	X	X	X
A			X				
N			X				
L			X				
<b>BLK=RPL:</b>							
ACB				X	X	X	X
AREA				X	X	X	X
AREALEN	X			X	X	X	
ARG				X	X	X	X
ECB				X	X	X	X
KEYLEN	X			X	X	X	

Keyword	Absolute Numeric	Code	Character String	Register	S-Type Address	Indirect S-Type Address	A-Type Address
NXTRPL				X	X	X	X
OPTCD			X				
RECLEN	X			X	X	X	
TRANSID	X			X	X	X	

## MODCB Macro

Keyword	Absolute Numeric	Code	Character String	Register	S-Type Address	Indirect S-Type Address	A-Type Address
ACB, EXLST, or RPL				X	X	X	X
<b>ACB:</b>							
BUFND	X			X	X	X	
BUFNI	X			X	X	X	
BUFSP	X			X	X	X	
DDNAME			X			X	
EXLST				X	X	X	X
MACRF		X					
MAREA				X	X	X	X
MLEN	X			X	X	X	
PASSWD				X	X	X	X
STRNO	X			X	X	X	
<b>EXLST:</b>							
EODAD				X	X	X	X
JRNAD				X	X	X	X
LERAD				X	X	X	X
SYNAD				X	X	X	X
A		X					
N		X					
L		X					
<b>RPL:</b>							
ACB				X	X	X	X
AREA				X	X	X	X
AREALEN	X			X	X	X	
ARG				X	X	X	X

## Using VSAM

Keyword	Absolute Numeric	Code	Character String	Register	S-Type Address	Indirect S-Type Address	A-Type Address
ECB				X	X	X	X
KEYLEN	X			X	X	X	
NXTRPL				X	X	X	X
OPTCD		X					
RECLEN	X			X	X	X	
TRANSID	X			X	X	X	

## SHOWCB Macro

Keyword	Absolute Numeric	Code	Character String	Register	S-Type Address	Indirect S-Type Address	A-Type Address
ACB, EXLST, or RPL				X	X	X	X
<b>ACB:</b>							
AREA				X	X	X	X
FIELDS		X					
LENGTH	X			X	X	X	
OBJECT		X					
<b>EXLST:</b>							
AREA				X	X	X	X
FIELDS		X					
LENGTH	X			X	X	X	
<b>RPL:</b>							
AREA				X	X	X	X
FIELDS		X					
LENGTH	X			X	X	X	
TRANSID	X			X	X	X	

## TESTCB Macro

Keyword	Absolute Numeric	Code	Character String	Register	S-Type Address	Indirect S-Type Address	A-Type Address
ACB, EXLST, or RPL				X	X	X	X
ERET				X	X	X	X
<b>ACB:</b>							
ACBLEN	X			X	X	X	

Keyword	Absolute Numeric	Character Code	Register	S-Type Address	Indirect S- Type Address	A-Type Address
ATRB		X				
AVSPAC	X		X	X	X	
BUFND	X		X	X	X	
BUFNI	X		X	X	X	
BUFNO	X		X	X	X	
BUFSP	X		X	X	X	
CINV	X		X	X	X	
DDNAME		X				
ERROR	X		X	X	X	
EXLST			X	X	X	X
FS	X		X	X	X	
KEYLEN	X		X	X	X	
LRECL	X		X	X	X	
MACRF		X				
MAREA			X	X	X	X
MLEN	X		X	X	X	
NCIS	X		X	X	X	
NDELRL	X		X	X	X	
NEXCP	X		X	X	X	
NEXT	X		X	X	X	
NINSR	X		X	X	X	
NIXL	X		X	X	X	
NLOGR	X		X	X	X	
NRETR	X		X	X	X	
NSSS	X		X	X	X	
NUPDR	X		X	X	X	
OBJECT		X				
OFLAGS		X				
OPENOBJ		X				
PASSWD			X	X	X	X
RKP	X		X	X	X	
STMST						X
STRNO	X		X	X	X	
<b>EXLST:</b>						

## Using VSAM

Keyword	Absolute Numeric	Character Code	Register String	S-Type Address	Indirect S-Type Address	A-Type Address
EODAD			X	X	X	X
EXLLEN	X		X	X	X	
JRNAD			X	X	X	X
LERAD			X	X	X	X
SYNAD			X	X	X	X
A		X				
N		X				
L		X				
<b>RPL:</b>						
ACB			X	X	X	X
AIXFLAG		X				
AIXPC	X		X	X	X	
AREA			X	X	X	X
AREALEN	X		X	X	X	
ARG			X	X	X	X
ECB			X	X	X	X
FDBK	X		X	X	X	
FTNCD	X		X	X	X	
IO		X				
KEYLEN	X		X	X	X	
NXTRPL			X	X	X	X
OPTCD		X				
RBA	X		X	X	X	
RECLEN	X		X	X	X	
RPLLEN	X		X	X	X	
TRANSID	X		X	X	X	

## Feedback Field Codes

VSAM request macros include ENDREQ, ERASE, GET, POINT, and PUT. After you enter one of these macros, or the CHECK macro, register 15 contains a return code that shows the manner which your request was completed. This is described even further by the error code placed in the FDBK field of the request parameter list (RPL) associated with your request. These error codes are as follows.

## When the Return Code in Register 15 is 0

Error Code	Meaning
0	Request completed successfully.
4	VSAM detected an END-OF-VOLUME condition.
8	VSAM detected a non-unique key in the alternate index.
16	A control area split occurred because there was not enough space to make an index entry in a sequence set record. Some data intervals could not be used in the control area that was split.
28	The record retrieved by a GET macro, without UPDATE, may be a duplicate of a record in another control interval. Eliminate duplicate records by processing the data using keyed access with UPDATE. For sequential processing, this error code is set only for the first record in the control interval.

## When the Return Code in Register 15 is 8

Error Code	Meaning
4	Either VSAM met an END-OF-FILE condition during sequential retrieval, or the search argument is greater than the highest existing key (or relative record number) in the file.
8	One of three things happened: <ul style="list-style-type: none"> <li>• An attempt was made to store a record with a duplicate key.</li> <li>• A duplicate record was found for an alternate index with the UNIQUEKEY option.</li> <li>• A record already exists at the accessed record location.</li> </ul>
12	VSAM detected a record out of sequence in a key-sequenced or relative-record file. There may be a duplicate key or record number.
16	No record was found. If a relative-record file was being accessed, VSAM may have detected a deleted or invalid empty slot at the accessed record location.  This code can be issued for a file being accessed through a path if the pointer to the record is missing from the alternate index. Although the record is in the base cluster, VSAM could not find it because the pointer to it was missing. This situation should only result from a system failure during UPGRADE processing.
20	The requested record is contained in a control interval that is already held in exclusive control by another request.
24	The requested record is on a volume or extent that cannot be accessed because no extent blocks are available.
28	All extents of the file are full. VSAM cannot suballocate any additional extents to the file for one of the following reasons: <ul style="list-style-type: none"> <li>• No secondary allocation was specified. Furthermore, no space of the required class was available for primary space suballocation on an additional volume (if one was specified).</li> <li>• The maximum number of extensions for the file has been exceeded.</li> <li>• No space of the required class is available for additional secondary allocations.</li> </ul>
32	An invalid RBA was specified.

Error Code	Meaning
36	The key of the record to be inserted does not fall into an existing key range in the file.
40	VSAM could not obtain a sufficiently large contiguous area of virtual storage.
44	The work area you have supplied through the RPL macro's AREA parameter is not large enough for the requested record.
48	Update of ESDS record is not permitted.
64	As many requests are active as the number specified in the STRNO parameter in the ACB macro. Therefore, another request cannot be started.
68	<p>The type of accessing for the request does not match the type of accessing in the access method control block. For example,</p> <ul style="list-style-type: none"> <li>• ADR or CNV was specified, but keyed access is requested.</li> <li>• INPUT was specified, explicitly or by default, but an UPDATE request was made.</li> <li>• GET UPD ADR was requested. However, ADR was not specified on the ACB macro when the SHAREOPTIONS(4) KSDS was opened.</li> </ul>
72	You requested keyed access for an entry-sequenced file.
76	You requested addressed or control interval insertion for a key-sequenced or relative record file.
80	You issued an ERASE macro either for an entry-sequenced file (directly or through a path), or for a file for which control-interval processing has been specified.
84	You specified LOCATE mode for either a PUT request or for processing in a user buffer.
88	<p>A positioning error occurred. The problem program did one of the following things:</p> <ul style="list-style-type: none"> <li>• Issued a sequential GET macro without having VSAM positioned first.</li> <li>• Changed from addressed to keyed access without having VSAM positioned for keyed-sequential retrieval.</li> <li>• Issued a sequential PUT macro for a relative-record file without having VSAM positioned first.</li> <li>• Attempted to improperly switch between forward and backward processing.</li> </ul>
92	You issued a PUT for UPDATE or an ERASE macro without a preceding GET for UPDATE macro.
96	<p>An attempt was made to either change the prime key of a record that is being updated, or to change an alternate key that has the UNIQUEKEY attribute.</p> <p>This produced a sequence error during sequential updating. For example, during REPRO REPLACE, two separate updates to the same record were attempted.</p>
100	An attempt was made to either change record length during update with addressed access, or to change record length for a relative-record file.

Error Code	Meaning
104	<p>You specified invalid or conflicting RPL options or parameters, as follows:</p> <ul style="list-style-type: none"> <li>• SKP together with BWD</li> <li>• LRD without BWD</li> <li>• CNV together with BWD</li> <li>• ARG parameter was not specified when required.</li> </ul>
108	<p>The RECLEN value specified for the RPL was one of the following:</p> <ul style="list-style-type: none"> <li>• Larger than the allowed maximum</li> <li>• Equal to zero</li> <li>• Smaller than key length plus relative key position</li> <li>• Not equal to record (slot) size specified for a relative-record file.</li> </ul> <p>For alternate index upgrade processing, the alternate index contains too many duplicate keys. Increase the maximum record length to accommodate more keys.</p>
112	<p>The length of the generic key specified for the RPL is too large or is equal to zero.</p>
116	<p>Either a request to insert records was issued during initial loading of the file, or a request other than PUT insert was issued during initial loading of a relative-record file. Possibly an attempt was made to read an empty file.</p>
132	<p>An attempt was made to retrieve a spanned record in LOCATE mode.</p>
136	<p>An attempt was made to retrieve a spanned record of a keyed-sequenced file with addressed access.</p>
140	<p>VSAM met an inconsistent spanned record, that is, one or more segments were incompletely updated or destroyed.</p> <p>If the request was GET, then the record (or as much of it as possible) was moved to the user's work area. The record may contain segments at different update levels. The RECLEN field of the RPL shows the length actually moved to the work area.</p> <p>If the request was sequential or skip-sequential (but not direct), then the file remains positioned for update or subsequent sequential retrieval. An update of the record will update the status of all segments to a consistent level.</p> <p>If the error was in the AIX during path access (RPL FTNCD=X'02'), then the base cluster is not accessed and no record is moved to the work area. During sequential or skip-sequential access, a subsequent request will access records with a higher alternate key than the one in error.</p>
144	<p>VSAM met a pointer in an alternate index without an associated base record.</p>
148	<p>The maximum number of pointers in the alternate index has been exceeded.</p>
156	<p>One or more records in this control interval may contain duplicate data after an addressed GET UPDATE. Any duplicates can be eliminated by processing the file using keyed access.</p>
192	<p>VSAM met an invalid relative-record number.</p>
196	<p>An addressed request was issued for a relative-record file.</p>
200	<p>An addressed or control-interval access was attempted through a path.</p>
204	<p>The program issued a PUT to insert a record while in backward mode.</p>

<b>Error Code</b>	<b>Meaning</b>
229	Record length change detected on expansion.
245	Compression Management Services error on compression.
246	Compression Management Services error on expansion.

## **When the Return Code in Register 15 is 12**

<b>Error Code</b>	<b>Meaning</b>
4	A READ error occurred for a file.
8	A READ error occurred for an index set.
12	A READ error occurred for a sequence set.
16	A WRITE error occurred for a file.
20	A WRITE error occurred for an index set.
24	A WRITE error occurred for a sequence set.

---

## Appendix C. Appendix for QUERY ADDRESS and QUERY MODDATE

The following names may be specified on the QUERY ADDRESS command. Those indicated with an asterisk (\*) may be specified on the QUERY MODDATE command.

ABBREV  
ADTSECT  
AFTSTART  
ALPHATB  
CONAUTHU  
CONBAM  
CONBAMN  
CONBAM2  
CONDMPU  
CONMAXU  
CONRECU  
CONSEGS  
CONFIG  
CONSYS  
CONSYSID  
CONTABSZ  
CONTRPRI  
CONUSED  
CONUSRS  
CONVSAM  
CONVSAMN  
CVT  
DEVADSK  
DEVCONS  
DEVTAB  
DIOSECT  
ECVT  
EXCPW  
EXTWA  
FVS  
FVSLADSV  
GCSUSRAB  
GCTABD \*  
GCTABDAB  
GCTABDAT  
GCTABDHX  
GCTABNW1  
GCTABNW2  
GCTACC \*  
GCTACF \*  
GCTACM \*  
GCTAES \*  
GCTAESAP

## QUERY ADDRESS/QUERY MODDATE

GCTAESAS  
GCTALP  
GCTALU \*  
GCTALUSO  
GCTANM \*  
GCTANT \*  
GCTANTM  
GCTANTT  
GCTARE \*  
GCTATT \*  
GCTATTDE  
GCTATTET  
GCTATTFT  
GCTATT44  
GCTATT62  
GCTATU \*  
GCTATUIT  
GCTAUD \*  
GCTAUDUP  
GCTBEO \*  
GCTBHC  
GCTBMR \*  
GCTBMRAR  
GCTBMRFL  
GCTBMRIN  
GCTBOP \*  
GCTBUF  
GCTBUFND  
GCTCAT \*  
GCTCATDK  
GCTCATMK  
GCTCATNB  
GCTCATR  
GCTCFG \*  
GCTCIO \*  
GCTCIOEX  
GCTCIOPU  
GCTCIORD  
GCTCLS \*  
GCTCMD \*  
GCTCMH \*  
GCTCMHE  
GCTCMSSH  
GCTCMSSS  
GCTCOI \*  
GCTCOM  
GCTCON \*  
GCTCPF \*  
GCTCTB  
GCTCTS  
GCTCUP \*

GCTCUPWA  
GCTCWR \*  
GCTDAS \*  
GCTDIE \*  
GCTDIO \*  
GCTDIOLR  
GCTDIOLW  
GCTDIORD  
GCTDIOWR  
GCTDIP \*  
GCTDIPBI  
GCTDIPDI  
GCTDIPOS  
GCTDIPTI  
GCTDLB \*  
GCTDLBCN  
GCTDMMP \*  
GCTDOS \*  
GCTDSI \*  
GCTDSP \*  
GCTDUM \*  
GCTDUMSD  
GCTDUMSI  
GCTDUMTD  
GCTDUQ \*  
GCTDUQBU  
GCTDUQDG  
GCTDUQFM  
GCTDUQGM  
GCTDUR \*  
GCTDURMT  
GCTDURSR  
GCTDUX \*  
GCTEHC  
GCTENQ \*  
GCTENQDB  
GCTENQDS  
GCTEIO \*  
GCTEIOAB  
GCTERS \*  
GCTERSTR  
GCTERSX  
GCTERW \*  
GCTERWRD  
GCTERWWR  
GCTEST \*  
GCTESTBR  
GCTESTXC  
GCTESTET  
GCTESTEX  
GCTETR \*

## **QUERY ADDRESS/QUERY MODDATE**

GCTFLD \*  
GCTFLDCN  
GCTFLE \*  
GCTFNC \*  
GCTFNS \*  
GCTFNSDI  
GCTFNST  
GCTFSV \*  
GCTFSVBA  
GCTFSVBR  
GCTFSVXC  
GCTFSV0A  
GCTFSV05  
GCTFSV78  
GCTGFC \*  
GCTGFCFF  
GCTGFCFM  
GCTGFD \*  
GCTGFDGP  
GCTGFDGM  
GCTGFDRM  
GCTGFE \*  
GCTGFEPS  
GCTGFERC  
GCTGFEUA  
GCTGFF \*  
GCTGFFAG  
GCTGFFLG  
GCTGFFRG  
GCTGFI \*  
GCTGFIBD  
GCTGFIPL  
GCTGFJ \*  
GCTGFJCS  
GCTGFT \*  
GCTGFTIN  
GCTGFTND  
GCTGIA \*  
GCTGII \*  
GCTGIM \*  
GCTGIMSB  
GCTGIMXC  
GCTGIMT  
GCTGIN \*  
GCTGINSX  
GCTGIP \*  
GCTGIPT  
GCTGIPU  
GCTGIT \*  
GCTGIU \*  
GCTGIUXA

GCTGIX \*  
GCTGLB \*  
GCTGMF \*  
GCTGSU \*  
GCTGSV \*  
GCTGSVBA  
GCTGSVBX  
GCTGSVBR  
GCTGSVXC  
GCTGSV0A  
GCTGSV04  
GCTGSV78  
GCTGSW \*  
GCTGSX \*  
GCTGVE \*  
GCTGX1 \*  
GCTGXN \*  
GCTHTB \*  
GCTIIS \*  
GCTINA \*  
GCTINALT  
GCTINA1S  
GCTINL \*  
GCTIOSAV  
GCTITE \*  
GCTITEWA  
GCTITM \*  
GCTITP \*  
GCTITPWA  
GCTITS \*  
GCTIUE \*  
GCTIUEFL  
GCTIUI \*  
GCTIUIFI  
GCTIUSSI  
GCTIUM \*  
GCTIUS \*  
GCTIUSBR  
GCTIUSXC  
GCTIUSSV  
GCTIUSTR  
GCTIUX \*  
GCTIXT \*  
GCTLAB \*  
GCTLAC \*  
GCTLAD \*  
GCTLADAD  
GCTLADN  
GCTLADNW  
GCTLADW  
GCTLAF \*

## **QUERY ADDRESS/QUERY MODDATE**

GCTLAFFE  
GCTLAFFT  
GCTLAFNX  
GCTLCK \*  
GCTLDC \*  
GCTLDF \*  
GCTLFS \*  
GCTLFSHM  
GCTLFSTY  
GCTLFSW  
GCTLK \*  
GCTLKM  
GCTLKT  
GCTLKWA  
GCTLK2  
GCTLOS \*  
GCTLPAB  
GCTMCKSA  
GCTMES  
GCTMOD \*  
GCTMSG \*  
GCTMSGWA  
GCTMTA \*  
GCTMTB \*  
GCTMTBEX  
GCTNUC \*  
GCTNUCIN  
GCTNUCIU  
GCTNUCL1  
GCTNUCL2  
GCTNUCL3  
GCTNUCMS  
GCTNUCOS  
GCTNUCPA  
GCTNUCPD  
GCTNUCPL  
GCTNUCQB  
GCTNUCSN  
GCTNUCSR  
GCTNUCTI  
GCTNUCT1  
GCTNUCT2  
GCTNUCT3  
GCTNXT \*  
GCTNXTAB  
GCTOME  
GCTOMX  
GCTOSR \*  
GCTPGF \*  
GCTPGFWA  
GCTPIO \*

GCTPIOB  
GCTPIOEX  
GCTPIOPR  
GCTPMA \*  
GCTPMB \*  
GCTPMC \*  
GCTPMC12  
GCTPMC6  
GCTPMC7  
GCTPMD \*  
GCTPMDB  
GCTPMDHX  
GCTPMD9  
GCTPMI \*  
GCTPML \*  
GCTPMLAD  
GCTPMM \*  
GCTPMN \*  
GCTPMR \*  
GCTPMS \*  
GCTPOS \*  
GCTPOSBE  
GCTPOSBR  
GCTPOSXC  
GCTPOST  
GCTQRD \*  
GCTQRL \*  
GCTQRQ \*  
GCTQRR \*  
GCTQRS \*  
GCTQRT \*  
GCTQRU \*  
GCTQRW \*  
GCTQRX \*  
GCTQRY \*  
GCTQRZ  
GCTRET \*  
GCTRETB  
GCTRET76  
GCTREX \*  
GCTREXB  
GCTREXBR  
GCTREXGC  
GCTREXIT  
GCTREXMS  
GCTREXSC  
GCTREXVE  
GCTREXV2  
GCTRFF \*  
GCTRFFCP  
GCTRFFF

## **QUERY ADDRESS/QUERY MODDATE**

GCTRFFFS  
GCTRFFGM  
GCTROS \*  
GCTR PY \*  
GCTR SS \*  
GCTR SSSU  
GCTR WB \*  
GCTR WBRD  
GCTR WBWR  
GCTR XL \*  
GCTS AI  
GCTS AP  
GCTS AR  
GCTS BS \*  
GCTS CL \*  
GCTS CL20  
GCTS CL23  
GCTS CN \*  
GCTS CNN  
GCTS CNO  
GCTS CNT  
GCTS CNUP  
GCTS CON  
GCTS CT \*  
GCTS CTCE  
GCTS CTCK  
GCTS CTNP  
GCTS DT \*  
GCTS DTA  
GCTS DTM  
GCTS DTWA  
GCTS DT2  
GCTS DX \*  
GCTS DXBR  
GCTS DXXC  
GCTS DXWA  
GCTS EB \*  
GCTS ER \*  
GCTS ERAB  
GCTS ERSY  
GCTS ET \*  
GCTS ETWA  
GCTS GIOP  
GCTS IDW1  
GCTS IG  
GCTS MAB  
GCTS OP \*  
GCTS OP19  
GCTS PI \*  
GCTS PIEP  
GCTS PIIR

GCTSQS \*  
GCTSQSGT  
GCTSQSPT  
GCTSTT \*  
GCTSTTCL  
GCTSTTNW  
GCTSTTR  
GCTSTTWX  
GCTSTTX  
GCTSUS \*  
GCTSUP \*  
GCTSVL \*  
GCTSVQ \*  
GCTSVQFM  
GCTSVQGM  
GCTSVQNT  
GCTSVQXT  
GCTSVT \*  
GCTSVT17  
GCTSVT24  
GCTSVT40  
GCTSVT64  
GCTSVT96  
GCTSYM \*  
GCTTAB  
GCTTCBCM  
GCTTIH \*  
GCTTIM \*  
GCTTIMFA  
GCTTIMST  
GCTTIMTE  
GCTTIMTI  
GCTTIMTT  
GCTTKN \*  
GCTTKS \*  
GCTTRK \*  
GCTTRKDE  
GCTTRKMA  
GCTTSABD  
GCTTZI \*  
GCTVAL \*  
GCTVALB  
GCTVALL  
GCTVALWA  
GCTVIB \*  
GCTVIP \*  
GCTVIP2  
GCTVIR \*  
GCTVIS \*  
GCTVSI \*  
GCTVSR \*

## QUERY ADDRESS/QUERY MODDATE

GCTVSS \*  
GCTVST \*  
GCTVSTBG  
GCTVSTSY  
GCTVSTVA  
GCTVTI \*  
GCTWAI \*  
GCTWAIBR  
GCTWAIXC  
GCTWAIT  
GCTWTE \*  
GCTWTR \*  
GCTXCP \*  
GCTYTE \*  
GCTYTG \*  
GCTYTI \*  
GCTYTI01  
GCTYTI02  
GCTYTK \*  
GCTYTO \*  
GCTYTW \*  
GCTZET  
GCTZIT  
GCTZITEP  
GCTZNR  
GIMSB  
GST  
INSWA  
INVTBL  
IUCBK  
IUSBR  
LOWERADD  
MTWA  
NUCABW  
NUCON  
OSSVCTAB  
PGMW  
POSBR  
REXXCODE  
SCANADD  
SCVT  
SDXBR  
SIE  
SVCWA  
SV202TAB  
SV203TAB  
SYID  
TABEND  
UPPERADD

# Appendix D. Data Compression Services

This appendix discusses how to use Data Compression Services with GCS as follows:

- Compression Processing
- Expansion Processing
- Using Compression and Expansion Dictionaries
- Compressing and Expanding GCS Data.

## Compression and Expansion Services

You can save data in a compressed format to conserve storage media and network transmission line costs. The CSRCMPSC macro provides a pair of services that compress and expand data. These services are available when the CVTCMPSC bit is on in the communication vector table (CVT).

Compression takes an input string of data and, using a data area called a *dictionary*, produces an output string of compression symbols. Each symbol represents a string of one or more characters from the input.

Expansion takes an input string of compression symbols and, using a *dictionary*, produces an output string of the characters represented by those compression symbols.

The CSRCMPSC is a general user interface to system Data Compression Services. When using the CSRCMPSC macro on GCS the user must supply a save area that is 144 bytes in length. In this respect, the GCS API differs from the API on MVS.

The CSRCMPSC macro has two defined functions:

- Compressing data
- Expanding previously compressed data

This interface uses the S/390 hardware compression instruction CMPSC. If your hardware does not support the CMPSC compression instruction, the system software simulation of the instruction will be used to perform the service. For more information on the CSRCMPSC macro, see [z/VM: CMS Macros and Functions Reference](#).

Using the Data Compression Services on GCS and CMS are similar. Except, GCS does not provide CSL or REXX support. For additional information on how to use Data Compression Services, see [z/VM: CMS Application Development Guide](#).



## 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*

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 book primarily documents intended Programming Interfaces that allow the customer to write programs to obtain services of z/VM.

This book also documents information that is NOT intended to be used as Programming Interfaces of z/VM. This information is identified where it occurs, either by an introductory statement to a chapter or section or by the following marking:

**NOT-PI**

<...NOT Programming Interface information...>

**NOT-PI end**

## 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>).

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) (<https://www.ibm.com/privacy>)
- [Cookies and Similar Technologies](https://www.ibm.com/privacy#Cookies_and_Similar_Technologies) ([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

- [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/Support Facility on the Hardware Management Console](#) ([https://www.ibm.com/docs/en/SSLTBW\\_2.3.0/pdf/SC14-7580-02.pdf](https://www.ibm.com/docs/en/SSLTBW_2.3.0/pdf/SC14-7580-02.pdf)), SC14-7580
- [Open Systems Adapter-Express ICC 3215 Support](#) (<https://www.ibm.com/docs/en/zos/2.3.0?topic=osa-icc-3215-support>), SA23-2247
- [Open Systems Adapter Integrated Console Controller User's Guide](#) ([https://www.ibm.com/docs/en/SSLTBW\\_2.3.0/pdf/SC27-9003-02.pdf](https://www.ibm.com/docs/en/SSLTBW_2.3.0/pdf/SC27-9003-02.pdf)), SC27-9003
- [Open Systems Adapter-Express Customer's Guide and Reference](#) ([https://www.ibm.com/docs/en/SSLTBW\\_2.3.0/pdf/ioa2z1f0.pdf](https://www.ibm.com/docs/en/SSLTBW_2.3.0/pdf/ioa2z1f0.pdf)), SA22-7935

## **Performance Toolkit for z/VM**

- [z/VM: Performance Toolkit Guide](#), SC24-6302
- [z/VM: Performance Toolkit Reference](#), SC24-6303

The following publications contain sections that provide information about z/VM Performance Data Pump, which is licensed with Performance Toolkit for z/VM.

- [z/VM: Performance](#), SC24-6301. See [z/VM Performance Data Pump](#).
- [z/VM: Other Components Messages and Codes](#), GC24-6300. See [Data Pump Messages](#).

## **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](https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ickug00_v2r5.pdf)), GC35-0033

## 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

### z/OS

IBM Documentation - z/OS (<https://www.ibm.com/docs/en/zos>)

# Index

## Special Characters

? operand  
    DBL command [64](#)  
\* operand  
    DBL command [64](#)  
    QUERY DISK command [122](#)

## Numerics

0 parameter  
    BLD macro [181](#)  
24-bit address  
    mode [351](#)  
    setting [343](#)  
31-bit address  
    mode [351](#)  
    setting [343](#)

## A

A parameter  
    EXLST macro [435](#)  
    FREEMAIN macro [234](#)  
    GENCB macro [444](#)  
    GETMAIN macro [259](#)  
    MODCB macro [459](#)  
ABEND macro  
    description [162](#)  
    format [162](#)  
abnormal end (abend) [22](#)  
ACB macro  
    description [418](#)  
    format [418](#)  
ACB parameter  
    GENCB macro [446](#)  
    MODCB macro [454](#), [461](#)  
    RPL macro [472](#)  
    SHOWCAT macro [476](#)  
    SHOWCB macro [481](#), [487](#)  
    TESTCB macro [491](#), [501](#)  
ACBLEN parameter  
    SHOWCB macro [481](#)  
    TESTCB macro [494](#)  
ACCEPT parameter  
    IUCVCOM macro [275](#)  
ACCESS command  
    description [59](#)  
    format [59](#)  
Access Control Block (ACB)  
    definition [55](#)  
ACQUIRE parameter  
    LOCKWD macro [302](#)  
ACSMETH parameter  
    SYNADAF (BSAM/QSAM) macro  
        [409](#)  
add  
add (*continued*)  
    tasks  
        using ATTACH macro [18](#)  
ADDR parameter  
    LOAD macro [299](#)  
    VALIDATE macro [362](#)  
ADDRESS GCS [42](#)  
ADDRESS operand  
    QUERY command [119](#)  
ADSR macro  
    description [164](#)  
    format [164](#)  
Advanced Program-to-Program Communications/VM  
    (APPC/VM)  
    definition [7](#)  
AIXFLAG parameter  
    TESTCB macro [501](#)  
AIXPC parameter  
    SHOWCB macro [487](#)  
    TESTCB macro [501](#)  
ALL operand  
    ETRACE command [73](#)  
    GDUMP command [99](#)  
    ITRACE command [109](#)  
AMODE instruction  
    determining [182](#)  
AMODE parameter  
    CONTENTS macro [198](#)  
    SYNCH macro [351](#)  
APPC/VM VTAM Support (AVS)  
    definition [3](#)  
application  
    supported in GCS [2](#)  
    unauthorized in GCS [44](#)  
applications  
    interfacing with Control Program [1](#)  
AREA parameter  
    GENCB macro [446](#)  
    MODCB macro [461](#)  
    RPL macro [472](#)  
    SHOWCAT macro [477](#)  
    SHOWCB macro [481](#), [484](#), [487](#)  
    TESTCB macro [501](#)  
AREALEN parameter  
    GENCB macro [446](#)  
    MODCB macro [461](#)  
    RPL macro [472](#)  
    SHOWCB macro [487](#)  
    TESTCB macro [501](#)  
ARG parameter  
    GENCB macro [446](#)  
    MODCB macro [461](#)  
    RPL macro [473](#)  
    SHOWCB macro [487](#)  
    TESTCB macro [501](#)  
ASYNCH parameter  
    ESTAE macro [224](#)

ATRB parameter  
   TESTCB macro [492](#)

attach  
   tasks [18](#)

ATTACH macro  
   description [165](#)  
   execute format [171](#)  
   list format [170](#)  
   standard format [165](#)

AUTHCALL macro  
   description [172](#)  
   format [172](#)

AUTHNAME macro  
   authorizing entry points [9](#)  
   description [174](#)  
   execute format [177](#)  
   list address format [177](#)  
   list format [176](#)  
   standard format [174](#)

authorize  
   accessing GCS [29](#)  
   commands [9, 29](#)  
   CP command use [9](#)  
   entry point [9](#)  
   provided by GCS [7](#)  
   real I/O [29, 49](#)  
   storage  
     accessing [50](#)  
     key switching [44](#)  
     protection [44](#)  
   supervisor state [9](#)  
   user IDs [9, 28](#)

AUTHUSER ADD operand  
   CONFIG command [62](#)

AUTHUSER DELETE operand  
   CONFIG command [62](#)

AUTHUSER macro  
   description [179](#)  
   format [179](#)

AUTHUSER operand  
   QUERY command [120](#)

AUTOLOG [29](#)

automatic  
   IPL [30](#)

AVOID operand  
   EXECIO command [80](#)

AVSPAC parameter  
   SHOWCB macro [482](#)  
   TESTCB macro [494](#)

**B**

BAMSEG parameter  
   CONFIG macro [195](#)

base register [35](#)

BIN parameter  
   TIME macro [359](#)

BINTVL parameter  
   STIMER macro [345](#)

BLDL macro  
   description [181](#)  
   format [181](#)

BLDVRP macro  
   description [423](#)

BLDVRP macro (*continued*)  
   format [423](#)

BLK parameter  
   GENCB macro [438, 446](#)

BLKSIZE operand  
   FILEDEF command [95](#)

BLKSIZE parameter  
   DCB (BSAM/QSAM) macro [386](#)

BLOCK operand  
   FILEDEF command [95](#)

BNDRY parameter  
   GETMAIN macro [259](#)

BRANCH parameter  
   ESTAE macro [224](#)  
   FREEMAIN macro [233](#)  
   GENIO macro [249](#)  
   GETMAIN macro [258](#)  
   IUCVCOM macro [275](#)  
   SCHEDEX macro [327](#)

BSAM  
   data management rules [379](#)

BUFFER operand  
   EXECIO command [80](#)

BUFFERS parameter  
   BLDVRP macro [423](#)

BUFND parameter  
   ACB macro [420](#)  
   GENCB macro [439, 440](#)  
   MODCB macro [455](#)  
   SHOWCB macro [481](#)  
   TESTCB macro [494](#)

BUFNI parameter  
   MODCB macro [455](#)  
   SHOWCB macro [481](#)  
   TESTCB macro [494](#)

BUFNO parameter  
   SHOWCB macro [482](#)  
   TESTCB macro [494](#)

BUFSP operand  
   DBL command [64](#)

BUFSP parameter  
   ACB macro [420](#)  
   GENCB macro [440](#)  
   MODCB macro [455](#)  
   SHOWCB macro [481](#)  
   TESTCB macro [494](#)

build  
   shared segments [27](#)  
   virtual machine groups [4](#)

BYTECOUNT operand  
   GDUMP command [98](#)

**C**

CALL macro  
   description [184](#)  
   execute format [186](#)  
   list format [185](#)  
   standard format [184](#)

cancel  
   programs [32](#)  
   timer [47, 361](#)

CANCEL parameter

CANCEL parameter (*continued*)  
     TTIME macro [361](#)

CARD operand  
     EXECIO command [78](#)

CASE operand  
     EXECIO command [80](#)

CAT operand  
     DLBL command [64](#)

CC operand  
     EXECIO command [79](#)

CCW parameter  
     GENIO macro [249](#)

chain save area [36](#)

CHANGE operand  
     DLBL command [64](#)  
     FILEDEF command [95](#)

change storage key [44](#)

changing GCS nucleus options [507](#)

channel control program  
     building [50](#)

CHAP macro  
     assigning task priority [19](#)  
     description [187](#)  
     format [187](#)

CHAR parameter  
     GENIO macro [248](#)

CHECK (BSAM) macro  
     description [380](#)  
     format [380](#)

CHECK macro  
     description [425](#)  
     format [425](#)

CI parameter  
     SHOWCAT macro [476](#)

CINV parameter  
     SHOWCB macro [482](#)  
     TESTCB macro [494](#)

CLEAR command  
     description [61](#)  
     format [61](#)

CLEAR operand  
     DLBL command [64](#)  
     FILEDEF command [95](#)

CLOSE (BSAM/QSAM)  
     macro  
     description [382](#)  
     execute format [383](#)  
     list format [383](#)  
     standard format [382](#)

CLOSE macro  
     description [427](#)  
     format [427](#)

CLOSE parameter  
     GENIO macro [248](#)

CLR parameter  
     AUTHNAME macro [175](#)  
     IUCVINI macro [287](#)  
     MACHEXIT macro [306](#)  
     TASKEXIT macro [354](#)

CMDSI macro  
     description [189](#)  
     execute format [192](#)  
     list address format [191](#)  
     list format [191](#)

CMDSI macro (*continued*)  
     standard format [189](#)

CODE parameter  
     IUCVCOM macro [275](#)

coding conventions, macro [158](#)

command  
     class assignments [9](#)  
     common with CMS [3](#)  
     entering [42](#)  
     file [42](#)  
     halting [107](#)  
     processing [43](#)  
     stopping [107](#)  
     supported in GCS [43](#)

commands  
     GROUP [102](#)

common lock [50](#)

COMMON operand  
     QUERY command [121](#)

common storage [26, 43](#)

communication support  
     between execs [42](#)  
     EXECCOMM macro [42](#)  
     through IUCV [47](#)  
     through the console [41](#)

compare  
     storage keys [50](#)

COMPcod parameter  
     SETRP macro [341](#)

CONFIG command  
     description [62](#)  
     format [62](#)

CONFIG macro  
     description [193](#)  
     format [193](#)

configuration file [4](#)

CONNECT parameter  
     IUCVCOM macro [275](#)

console  
     CMDSI macro [42](#)  
     issuing commands from [42](#)  
     support [41](#)  
     writing messages to and from [42](#)

CONTENTS macro  
     description [197](#)

control  
     conventions for  
         passing [37](#)  
         receiving [37](#)

GCS  
     access to GCS supervisor [8](#)  
     CP commands [9](#)  
     supervisor state [9](#)

Control Program (CP)  
     AUTOLOG [30](#)  
     command restrictions [9](#)  
     intercepting instructions, diagram of [15](#)  
     interfacing with licensed applications [1](#)  
     SAVESYS command [28](#)  
     signal system service [7](#)

convention for  
     formatting macros [159](#)  
     passing control [37](#)  
     receiving control [37](#)

conventions, macro coding [158](#)  
 Conversational Monitor System (CMS)  
     defining [94](#)  
     files GCS processes [51](#)  
     LOAD command [28](#)  
     load libraries, defining [101](#)  
     relating to GCS [3](#), [42](#)  
     with SNA [14](#)  
 coordinate  
     resources [22](#)  
     tasks [20](#)  
 COPIES parameter  
     GENCB macro [441](#), [444](#), [448](#)  
 CP operand  
     EXECIO command [78](#)  
 CREATE parameter  
     GCSTOKEN macro [242](#)  
 CT parameter  
     ESTAE macro [223](#)  
 CVT macro  
     description [200](#)  
     format [200](#)

**D**

data compression  
     benefits of [53](#), [539](#)  
     definition of [53](#), [539](#)  
     with GCS [539](#)  
 Data Control Block (DCB)  
     multiple openings [52](#)  
 data management [51](#)  
 DATA operand  
     EXECIO command [79](#)  
 DATA parameter  
     GENIO macro [248](#)  
     GTRACE macro [265](#)  
     SHOWCB macro [481](#)  
 data path  
     through VTAM machine [17](#)  
 DCB (BSAM/QSAM) macro  
     description [385](#)  
     format [385](#)  
 DCBD (BSAM/QSAM)  
     macro  
     description [391](#)  
     format [391](#)  
 ddname operand  
     FILEDEF command [94](#)  
 DDNAME operand  
     DLBL command [64](#)  
 DDNAME parameter  
     ACB macro [421](#)  
     DCB (BSAM/QSAM) macro  
     386  
     GENCB macro [440](#)  
     MODCB macro [456](#)  
     SHOWCB macro [481](#)  
     TESTCB macro [494](#)  
 DE parameter  
     ATTACH macro [166](#)  
     DELETE macro [202](#)  
     LINK macro [294](#)  
     LOAD macro [299](#)

DE parameter (*continued*)  
     XCTL macro [374](#)  
 debug  
     commands [9](#), [29](#)  
 DEC parameter  
     TIME macro [359](#)  
 define  
     CMS files [94](#)  
     CMS load libraries [101](#)  
     commands [112](#)  
     spool files [94](#)  
     VSAM files [64](#)  
 DELETE macro  
     description [202](#)  
     format [202](#)  
 DELETE parameter  
     GCSTOKEN macro [242](#)  
 DEQ macro  
     description [204](#)  
     execute format [207](#)  
     list format [206](#)  
     standard format [204](#)  
     synchronizing tasks [22](#)  
 DET operand  
     RELEASE command [146](#)  
 DETACH macro  
     description [208](#)  
     format [208](#)  
 DEV parameter  
     GENIO macro [250](#)  
 DEVTAB parameter  
     DEVTYPE macro [210](#)  
 DEVTYPE macro  
     description [210](#)  
     format [210](#)  
 DIAG98 [25](#)  
 DINTVL parameter  
     STIMER macro [346](#)  
 directory user control statement  
     entries  
         DIRECTORY [49](#)  
         OPTION [49](#)  
 disable  
     external tracing [73](#)  
     tracing of GTRACE events [108](#)  
 discard  
     tasks [18](#)  
 disk  
     accessing  
         automatically at IPL [31](#)  
         releasing [146](#)  
 DISK operand  
     FILEDEF command [95](#)  
     QUERY command [122](#)  
 DISKR operand  
     EXECIO command [78](#)  
 DISKW operand  
     EXECIO command [79](#)  
 DISP MOD operand  
     FILEDEF command [95](#)  
 dispatch  
     tasks [19](#)  
 display field  
     ACB [480](#)

display field (*continued*)  
     exit list 484  
     RPL 486

DLBL command  
     description 64  
     format 64

DLBL operand  
     QUERY command 124

DLVRP macro  
     description 429  
     format 429

DPMOD parameter  
     ATTACH macro 167

DSECT parameter  
     IHASDWA macro 273

DSN operand  
     DLBL command 64

DSORG parameter  
     DCB (BSAM/QSAM) macro 386  
     DCBD (BSAM/QSAM) macro 391

DSORG PS operand  
     FILEDEF command 95

DSP operand  
     ETRACE command 74  
     ITRACE command 109

DSS operand  
     GDUMP command 99

DUMMY operand  
     FILEDEF command 95

dump  
     virtual machine storage 98

DUMP operand  
     QUERY command 126  
     SET command 150

DUMP parameter  
     ABEND macro 162  
     SETRP macro 341

DUMLOCK operand  
     QUERY command 127  
     SET command 151

DUMPVM operand  
     CONFIG command 62  
     QUERY command 128

DUMPVM parameter  
     CONFIG macro 194

**E**

E parameter  
     ENQ macro 214  
     FREEMAIN macro 234

EC parameter  
     GETMAIN macro 258

ECB parameter  
     ATTACH macro 166  
     GENCB macro 447  
     MODCB macro 462  
     RPL macro 473  
     SHOWCB macro 487  
     TESTCB macro 501  
     WAIT macro 365  
     WTOR macro 370

ECBLIST parameter

ECBLIST parameter (*continued*)  
     WAIT macro 365

ECVT macro  
     description 212  
     format 212

EMSG operand  
     EXECIO command 79, 88

end  
     tasks 22

END operand  
     ETRACE command 73  
     GDUMP command 98  
     ITRACE command 109

END parameter  
     AUTHUSER macro 179  
     CONTENTS macro 198  
     RESSTOR macro 320  
     SEGMENT macro 338

ENDREQ macro  
     description 430  
     format 430

ENQ macro  
     description 213  
     execute format 217  
     list format 217  
     standard format 213  
     synchronizing tasks 22

ENTRY parameter  
     IDENTIFY macro 270

entry point  
     authorizing with AUTHNAME 9  
     shared storage 47

EODAD parameter  
     DCB (BSAM/QSAM) macro 386  
     EXLST macro 434  
     GENCB macro 443  
     MODCB macro 459  
     SHOWCB macro 485  
     TESTCB macro 497

EP parameter  
     ATTACH macro 165  
     AUTHCALL macro 172  
     AUTHNAME macro 174  
     CONTENTS macro 197  
     DELETE macro 202  
     IDENTIFY macro 270  
     LINK macro 293  
     LOAD macro 298  
     MACHEXIT macro 306  
     TASKEXIT macro 354  
     XCTL macro 373

EPLOC parameter  
     ATTACH macro 166  
     AUTHCALL macro 172  
     DELETE macro 202  
     IDENTIFY macro 270  
     LINK macro 293  
     LOAD macro 298  
     XCTL macro 374

ERASE command  
     description 70  
     format 70

ERASE macro

**ERASE** macro (*continued*)
   
 description 432
   
 format 432

**ERET** parameter
   
 TESTCB macro 492, 497, 501

**ERRET** parameter
   
 LOAD macro 299

**ERROR** parameter
   
 AUTHNAME macro 175
   
 CMDSI macro 190
   
 GENIO macro 250
   
 IUCVCOM macro 275
   
 IUCVINI macro 289
   
 MACHEXIT macro 306
   
 SHOWCB macro 481
   
 TASKEXIT macro 355
   
 TESTCB macro 494

**ESPIE** macro
   
 description 219
   
 format 219

**establish**
  
 base register 35

**ESTAE** macro
   
 abnormal termination 23
   
 description 223
   
 execute format 228
   
 list format 227
   
 standard format 223

**ESTATE/ESTATEW** command
   
 description 71
   
 format 71

**ETRACE** command
   
 description 73
   
 format 73

**ETRACE** operand
   
 QUERY command 129

**ETXR** parameter
   
 ATTACH macro 167

**EU** parameter
   
 FREEMAIN macro 234
   
 GETMAIN macro 258

**Event Control Block (ECB)**
  
 definition 21

**exec**
  
 using 42

**EXECCOMM** macro
   
 description 229
   
 format 229

**EXECIO** command
   
 description 77
   
 format 76

**execute**
  
 channel programs 49
   
 format
   
 GCS macro description 158
   
 VSAM macro description 518
   
 real channel I/O 49

**exit**
  
 GCS
   
 establishing 51
   
 machine 51
   
 resource cleanup 23
   
 scheduling 51
   
 task 22

**EXIT** parameter
   
 GENIO macro 247
   
 IUCVCOM macro 275
   
 IUCVINI macro 287
   
 SCHEDEX macro 326

**EXITBR** parameter
   
 GENIO macro 248

**EXLLEN** parameter
   
 SHOWCB macro 485
   
 TESTCB macro 498

**EXLST** macro
   
 description 434
   
 format 434

**EXLST** parameter
   
 ACB macro 421
   
 DCB (BSAM/QSAM) macro 387
   
 GENCB macro 440, 443
   
 MODCB macro 456, 459
   
 SHOWCB macro 482, 484
   
 TESTCB macro 494, 497

**EXT** operand
   
 ACCESS command 59
   
 ETRACE command 74
   
 ITRACE command 109

**extended file system** 51

**external tracing** 73

## F

**FDBK** parameter
   
 SHOWCB macro 487
   
 TESTCB macro 501

**FID** parameter
   
 GTRACE macro 265

**FIELDS** parameter
   
 SHOWCB macro 481, 485, 487

**FIFO** operand
   
 EXECIO command 80

**file**
  
 defining
   
 CMS 94
   
 VSAM 64
   
 sharing 52

**FILEBLK** parameter
   
 CMDSI macro 190

**FILEDEF** command
   
 description 94
   
 format 94

**FILEDEF** operand
   
 QUERY command 130

**FIND** operand
   
 EXECIO command 80

**FINIS** operand
   
 EXECIO command 80

**FLS** macro
   
 description 231
   
 format 231

**FM** operand
   
 ACCESS command 59
   
 ERASE command 70
   
 ESTATE/ESTATEW command 71

**FN** operand
   
 ACCESS command 59

FN operand (*continued*)  
     ERASE command 70  
     ESTATE/ESTATEW command 71  
 format  
     conventions  
         macros 159  
 FORMAT parameter  
     GENIO macro 249  
 FORMAT TYPE operand  
     GDUMP command 99  
 formatting routines, coding 267  
 FRE operand  
     ETRACE command 74  
     ITRACE command 109  
 FREE parameter  
     GCSSAVE macro 240  
     GCSSAVI macro 241  
 FREEMAIN macro  
     description 233  
     execute format 237  
     list format 236  
     standard format 233  
 FS parameter  
     SHOWCB macro 482  
     TESTCB macro 494  
 FT operand  
     ACCESS command 59  
     ERASE command 70  
     ESTATE/ESTATEW command 71  
 FTNCD parameter  
     SHOWCB macro 487  
     TESTCB macro 501

**G**

GCSBAM operand  
     SET SYSNAME command 154  
 GCSLEVEL macro  
     description 238  
     format 238  
 GCSLEVEL operand  
     QUERY command 131  
 GCSSAVE macro  
     description 240  
     format 240  
 GCSSAVI macro  
     description 241  
     format 241  
 GCSTOKEN macro  
     description 242  
     execute format 245  
     GCSTOKEN parameter 243  
     list address format 245  
     list format 244  
     standard format 242  
 GCSVSAM operand  
     SET SYSNAME command 154  
 GDUMP command  
     description 98  
     format 98  
 GENCB macro  
     access control block format 437  
     description 438  
     exit list format 442

GENCB macro (*continued*)  
     request parameter list format 445  
 GENCB parameter notation 519  
 General I/O 49  
 generate  
     ACB 438  
     exit list 443  
     macro format for VSAM 518  
     RPL 445  
 GENIO macro  
     description 247  
     execute format 256  
     list address format 255  
     list format 254  
     performing I/O 49  
     standard format 247  
 GET (QSAM) macro  
     description 394  
     format 394  
 GET macro  
     description 451  
     format 451  
 GET operand  
     ETRACE command 74  
     ITRACE command 109  
 GET parameter  
     GCSSAVE macro 240  
     GCSSAVI macro 241  
 GETMAIN macro  
     description 257  
     execute format 263  
     list format 262  
     standard format 257  
 GLOBAL command  
     description 101  
     format 101  
 GROUP command 102  
 Group Control System (GCS)  
     accessing authority 8  
     administration  
         authorizing access to GCS 29  
         authorizing commands 29  
         authorizing for real I/O 29  
         setting up a PROFILE GCS 29  
         using AUTOLOG functions 29  
     applications 2  
     basic function 1  
     channel control program 29  
     commands  
         ACCESS 59  
         CLEAR 61  
         CONFIG 62  
         DLBL 64  
         ERASE 70  
         ESTATE/ESTATEW 71  
         ETRACE 73  
         EXECIO 77  
         FILEDEF 94  
         GDUMP 98  
         GLOBAL 101  
         HX 107  
         ITRACE 108  
         LOADCMD 112  
         OSRUN 116

Group Control System (GCS) (*continued*)  
commands (*continued*)  
    QUERY [117](#)  
    QUERY ADDRESS [119](#)  
    QUERY AUTHUSER [120](#)  
    QUERY COMMON [121](#)  
    QUERY DISK [122](#)  
    QUERY DLBL [124](#)  
    QUERY DUMP [126](#)  
    QUERY DUMLOCK [127](#)  
    QUERY DUMPVM [128](#)  
    QUERY ETRACE [129](#)  
    QUERY FILEDEF [130](#)  
    QUERY GCSLEVEL [131](#)  
    QUERY GROUP [132](#)  
    QUERY IPOLL [133](#)  
    QUERY ITRACE [134](#)  
    QUERY LOADALL [135](#)  
    QUERY LOADCMD [136](#)  
    QUERY LOADLIB [137](#)  
    QUERY LOCK [138](#)  
    QUERY MODDATE [139](#)  
    QUERY REPLY [140](#)  
    QUERY REXXSTOR [141](#)  
    QUERY SEARCH [142](#)  
    QUERY SYSNAMES [143](#)  
    QUERY TRACETAB [144](#)  
    QUERY TSLICE [145](#)  
    RELEASE [146](#)  
    REPLY [147](#)  
    SET [149](#)  
    SET DUMP [150](#)  
    SET DUMLOCK [151](#)  
    SET IPOLL [152](#)  
    SET REXXSTOR [153](#)  
    SET SYSNAME [154](#)  
configuration  
    using the GROUP command [512](#)  
configuration file [102](#)  
console and command support [41](#)  
data management services [51](#)  
dumps  
    abend with DUMP option [22](#)  
immediate commands [58](#)  
macros  
    ABEND [162](#)  
    ADSR [164](#)  
    ATTACH [165](#)  
    AUTHCALL [172](#)  
    AUTHNAME [174](#)  
    AUTHUSER [179](#)  
    BLDL [181](#)  
    CALL [184](#)  
    CHAP [187](#)  
    checking storage key [50](#)  
    CMDSI [189](#)  
    CONFIG [193](#)  
    CONTENTS [197](#)  
    controlling locks [50](#)  
    CVT [200](#)  
    DELETE [202](#)  
    DEQ [204](#)  
    DETACH [208](#)  
    DEVTYPE [210](#)

Group Control System (GCS) (*continued*)  
macros (*continued*)  
    ECVT [212](#)  
    ENQ [213](#)  
    ESPIE [219](#)  
    ESTAE [223](#)  
    EXECCOMM [229](#)  
    FLS [231](#)  
    FREEMAIN [233](#)  
    GCS supported macro overview [3](#)  
    GCSLEVEL [238](#)  
    GCSSAVE [240](#)  
    GCSSAVI [241](#)  
    GCSTOKEN [242](#)  
    GENIO [247](#)  
    GETMAIN [257](#)  
    GTRACE [265](#)  
    IDENTIFY [270](#)  
    IHADVA [272](#)  
    IHASDWA [273](#)  
    IUCV communication [47](#)  
    IUCVCOM [275](#)  
    LINK [293](#)  
    LOAD [298](#)  
    loading modules [45](#)  
    LOCKWD [302](#)  
    MACHEXIT [305](#)  
    managing data [51](#)  
    managing timers [46](#)  
    performing I/O [49](#)  
    PGLOCK [310](#)  
    PGULOCK [312](#)  
    POST [314](#)  
    processing VSAM files [54](#)  
    RDJFCB [317](#)  
    resource sharing [22](#)  
    RESSTOR [320](#)  
    RETURN [322](#)  
    SAVE [324](#)  
    SCHEDEX [326](#)  
    SDUMP [329](#)  
    SDUMPX [333](#)  
    securing storage [50](#)  
    SEGMENT [338](#)  
    SETRP [340](#)  
    SPLEVEL [343](#)  
    STIMER [345](#)  
    SYMREC [348](#)  
    SYNCH [350](#)  
    TASKEXIT [354](#)  
    TIME [359](#)  
    TTIME [361](#)  
    VALIDATE [362](#)  
    WAIT [365](#)  
    WTO [368](#)  
    WTOR [370](#)  
    XCTL [373](#)  
multitasking services [17](#)  
native services [47](#)  
nucleus buildlist [507](#)  
nucleus options, changing [507](#)  
OS services [43](#)  
overview [13](#)  
program management [44](#)

## Group Control System (GCS) (*continued*)

rebuilding 513  
relating to CMS 3, 28  
storage  
    common storage 43  
    dump 4, 10  
    fetch-protected 44  
    key 44  
    layout in a group 25  
    managing 43  
    obtaining 44  
    private storage 43  
    protecting 44  
    requirements 25  
    securing pages of 50  
    using FREEMAIN macro 44  
    using GETMAIN macro 44  
supervisor 4  
tailoring 507  
timer management 46  
user ID directory entries 30  
virtual storage layout 25

## group exec 28

GROUP operand  
    ETRACE command 74  
    ITRACE command 109  
    QUERY command 132

GTRACE macro  
    description 265  
    execute format 267  
    list format 266  
    standard format 265

GTRACE operand  
    ITRACE command 109

## H

HALT parameter  
    GENIO macro 249

halting commands and programs 107

HDR parameter  
    SDUMP macro 329  
    SDUMPX macro 333

HDRAD parameter  
    SDUMP macro 329  
    SDUMPX macro 333

HEXLOC operand  
    GDUMP command 98

HX (Halt Execution) immediate command 32

HX command  
    description 107  
    format 107

## I

I/O operand  
    ETRACE command 74  
    ITRACE command 109

ID operand  
    REPLY command 147

ID parameter  
    CALL macro 185  
    GTRACE macro 265

## ID parameter (*continued*)

LINK macro 294  
SCHEDEX macro 326

IDENTIFY macro  
    description 270  
    format 270

IHADVA macro  
    description 272  
    format 272

IHASDWA macro  
    description 273  
    format 273

immediate  
    commands 58

INDEX parameter  
    SHOWCB macro 481

initialize  
    CMS  
        from a SNA terminal 14

GCS 30

INPUT parameter  
    OPEN (QSAM/BSAM) macro 398  
    RDJFCB macro 317

## install

    considerations  
        defining authorized user IDs 9

Inter-User Communications Vehicle (IUCV)

    defined 16  
    functions provided with GCS 47, 48  
    use in group communications 7

## interface with

    GCS (Group Control System) 1  
    licensed Applications and Control Program 1  
    shared VTAM 16

internal trace table 26, 27

## IO parameter

    TESTCB macro 501

IPL command 5, 30

IPOLL operand  
    QUERY command 133  
    SET command 152

ITRACE command  
    description 108  
    format 108

ITRACE operand  
    QUERY command 134

IUCVCOM macro  
    description 275  
    execute format 284  
    list address format 283  
    list format 283  
    standard format 275

IUCVINI macro  
    description 286  
    execute format 291  
    list address format 291  
    list format 290  
    standard format 286

## J

join  
    groups 5, 30

## JRNAD parameter (*continued*)

- EXLST macro 434
- GENCB macro 443
- MODCB macro 459
- SHOWCB macro 485
- TESTCB macro 497

## JSTCB parameter

- ATTACH macro 168

## K

### key

- changing 44
- protection 44

### KEY parameter

- ESTAE macro 224
- GETMAIN macro 258
- VALIDATE macro 362

### KEYLEN parameter

- BLDVRP macro 424
- GENCB macro 447
- MODCB macro 462
- RPL macro 473
- SHOWCB macro 482, 487
- TESTCB macro 494, 502

## L

### L parameter

- EXLST macro 435
- GENCB macro 444
- MODCB macro 459

### LA parameter

- GETMAIN macro 259

### last operand

- QUERY MODDATE 139

### layout virtual storage

- 25

### LENGTH parameter

- GENCB macro 441, 444, 448
- SHOWCB macro 481, 485, 487
- VALIDATE macro 362

### LERAD parameter

- EXLST macro 434
- GENCB macro 443
- MODCB macro 459
- SHOWCB macro 485
- TESTCB macro 497

### LIBNAME operand

- GLOBAL command 101

### LIFO operand

- EXECIO command 80

### LINK macro

- description 293
- execute format 296
- list format 296
- standard format 293

### linkage register

- 35

### list address format

- 158, 518

### list format

- 158, 518

### LIST parameter

- SDUMP macro 329

- SDUMPX macro 334

### LNG parameter

## LNG parameter (*continued*)

- GTRACE macro 265

### load

- GCS modules 44, 112
- GCS supervisor, the 30
- library 31, 44

### LOAD macro

- description 298
- format 298

### LOADALL operand

- QUERY command 135

### LOADCMD command

- description 112
- format 112

### LOADCMD operand

- QUERY command 136

### loading functions

- 45, 46

### LOADLIB operand

- GLOBAL command 101

### LOADLIB parameter

- QUERY command 137

### LOC parameter

- GETMAIN macro 259

### LOCATE operand

- EXECIO command 80

### lock

- functions 50

- local 50

### LOCK operand

- QUERY command 138

### LOCK parameter

- LOCKWD macro 302

### LOCKWD macro

- description 302
- format 302

### LRECL operand

- FILEDEF command 95

### LRECL parameter

- DCB (BSAM/QSAM) macro 387
- SHOWCB macro 482
- TESTCB macro 494

### LV parameter

- FREEMAIN macro 234

- GETMAIN macro 258

## M

### MACHEXIT macro

- description 305

- execute format 309

- list address format 308

- list format 308

- standard format 305

### machine exits

- 51

### MACRF parameter

- ACB macro 419

- DCB (BSAM/QSAM) macro 387

- GENCB macro 438

- MODCB macro 454

- TESTCB macro 492

### macro

- formats

**macro** (*continued*)
   
 formats (*continued*)
   
 execute format 158
   
 in GCS 158
   
 list address format 158
   
 list format 158
   
 standard format 158
   
 macro coding conventions 158
   
 macro return code placement 159
   
 manage timer service 46
   
**MAREA** parameter
   
 ACB macro 421
   
 GENCB macro 441
   
 MODCB macro 456
   
 SHOWCB macro 482
   
 TESTCB macro 494
   
**MARGINS** operand
   
 EXECIO command 80
   
**MAX** operand
   
 QUERY DISK command 122
   
**MAXVM** parameter
   
 CONFIG macro 194
   
**MEMBER** operand
   
 LOADCMD command 112
   
 OSRUN command 116
   
**message**
  
 replying to 32, 147
   
 sending to 42
   
 WTO macro 42
   
 WTOR macro 42
   
**message examples, notation used in** xv
   
**MESSAGE** parameter
   
 WTO macro 370
   
**MF** parameter
   
 ATTACH macro 171
   
 AUTHNAME macro 177, 178
   
 CALL macro 186
   
 CLOSE (BSAM/BSAM) macro 383, 384
   
 CMDSI macro 191, 192
   
 DEQ macro 207
   
 ENQ macro 217, 218
   
 ESTAE macro 227, 228
   
 FREEMAIN macro 237
   
 GENIO macro 255, 256
   
 GETMAIN macro 263, 264
   
 GTRACE macro 267
   
 IUCVCOM macro 283–285
   
 IUCVINI macro 290–292
   
 LINK macro 297
   
 MACHEXIT macro 308, 309
   
 OPEN (QSAM/BSAM) macro 400
   
 READ (BSAM) macro 408
   
 SDUMP macro 331, 332
   
 SDUMPX macro 336, 337
   
 SYMREC macro 348, 349
   
 SYNCH macro 353
   
 TASKEXIT macro 357, 358
   
 WRITE (BSAM) macro 415
   
 WTO macro 369
   
 WTOR macro 371, 372
   
 XCTL macro 376
   
**MLEN** parameter
   
 ACB macro 421
   
 GENCB macro 441
   
**MLEN** parameter (*continued*)
   
 MODCB macro 456
   
 SHOWCB macro 482
   
 TESTCB macro 494
   
**MNOTE** 159
   
**MODCB** macro
   
 access control block format 453
   
 description 453
   
 exit list format 458
   
 request parameter list format 460
   
**MODCB** parameter notation 519
   
**MODDATE** operand
   
 QUERY MODDATE 139
   
**MODE** operand
   
 ACCESS command 59
   
 DBL command 64
   
 QUERY DISK command 122
   
 RELEASE command 146
   
**modify**
  
 ACB 453
   
 exit list 458
   
 RPL 460
   
**MODIFY** parameter
   
 GENIO macro 249
   
**MULT** operand
   
 DBL command 64
   
**multiple**
  
 DCBs 52
   
 virtual machine groups 5
   
**multitask**
  
 assigning priority 19
   
 coordinating tasks 20
   
 defining 17
   
 exit routines, defining 22
   
 task family tree diagram 18
   
 terminating tasks 22
   
**MVS™ functions simulated** 43

## N

**N** parameter
   
 EXLST macro 435
   
 GENCB macro 444
   
 MODCB macro 459
   
**NAME** operand
   
 LOADCMD command 112
   
**NAME** parameter
   
 AUTHNAME macro 175
   
 AUTHUSER macro 179
   
 CONTENTS macro 197
   
 GCSTOKEN macro 242
   
 IUCVCOM macro 275
   
 IUCVINI macro 287
   
 MACHEXIT macro 306
   
 QUERY command 119
   
 QUERY MODDATE 139
   
 RESSTOR macro 320
   
 SEGMENT macro 338
   
 SHOWCAT macro 476
   
 TASKEXIT macro 355
   
**native**
  
 GCS services 47
   
**NCIS** parameter
   
 SHOWCB macro 482

NCIS parameter (*continued*)

  TESTCB macro [494](#)

NDELR parameter

  SHOWCB macro [482](#)

  TESTCB macro [494](#)

NetView [3](#)

NEXCP parameter

  TESTCB macro [495](#)

NEXT parameter

  SHOWCB macro [482](#)

  TESTCB macro [495](#)

NINSR parameter

  SHOWCB macro [482](#)

  TESTCB macro [495](#)

NIXL parameter

  SHOWCB macro [482](#)

  TESTCB macro [495](#)

NLOGR parameter

  SHOWCB macro [482](#)

  TESTCB macro [495](#)

NOCHANGE operand

  DLBL command [64](#)

  FILEDEF command [95](#)

nonreenterable program save area [36](#)

notation used in message and response examples [xv](#)

NOTE (BSAM) macro

  description [396](#)

  format [396](#)

NOTYPE operand

  EXECIO command [80](#)

NRETR parameter

  SHOWCB macro [483](#)

  TESTCB macro [495](#)

NSSS parameter

  SHOWCB macro [483](#)

  TESTCB macro [495](#)

NUPDR parameter

  SHOWCB macro [483](#)

  TESTCB macro [495](#)

NXTRPL parameter

  GENCB macro [447](#)

  MODCB macro [462](#)

  RPL macro [473](#)

  SHOWCB macro [488](#)

  TESTCB macro [502](#)

## O

OBJECT parameter

  SHOWCB macro [481](#)

  TESTCB macro [492](#)

obtain

  storage [44](#)

OFF operand

  ETRACE command [74](#)

  ITRACE command [109](#)

OFLAGS parameter

  TESTCB macro [495](#)

OL parameter

  CONTENTS macro [197](#)

OPEN (BSAM/QSAM)

  macro

  description [398](#)

  execute format [400](#)

OPEN (BSAM/QSAM) macro (*continued*)

  list format [399](#)

  standard format [398](#)

OPEN macro

  description [465](#)

  format [465](#)

OPEN parameter

  GENIO macro [247](#)

OPENOBJ parameter

  TESTCB macro [495](#)

operation

  initializing GCS [30](#)

OPTCD operand

  FILEDEF command [95](#)

OPTCD parameter

  DCB (BSAM/QSAM) macro

[389](#)

  GENCB macro [447](#)

  MODCB macro [462](#)

  RPL macro [473](#)

  TESTCB macro [502](#)

OPTION

  directory control statement [29, 49](#)

OS services [43](#)

OSRUN command

  description [116](#)

  format [116](#)

OUTPUT parameter

  OPEN (QSAM/BSAM) macro [398](#)

  RDJFCB macro [317](#)

OV parameter

  ESTAE macro [223](#)

## P

PARAM parameter

  ATTACH macro [166](#)

  ESTAE macro [224](#)

  LINK macro [294](#)

  XCTL macro [376](#)

parameter

  list

    examples of [113](#)

  notation

    GENCB [519](#)

    MODCB [519](#)

    SHOWCB [519](#)

    TESTCB [519](#)

PARM operand

  OSRUN command [116](#)

PARM parameter

  ESPIE macro [220](#)

PARM1 parameter

  SYNADAF (BSAM/QSAM) macro

[409](#)

PARM2 parameter

  SYNADAF (BSAM/QSAM) macro

[410](#)

pass control [37](#)

PASSWD parameter

  ACB macro [422](#)

  GENCB macro [441](#)

  MODCB macro [457](#)

  SHOWCB macro [482](#)

PASSWD parameter (*continued*)  
     TESTCB macro 495  
 path between SNA console and virtual machine 14  
 PATH parameter  
     IUCVCOM macro 275  
 perform GCS  
     real I/O 29, 49  
     virtual I/O 49  
 PERM operand  
     DLBL command 64  
     FILEDEF command 96  
 PGLOCK macro  
     description 310  
     format 310  
 PGULOCK macro  
     description 312  
     format 312  
 POINT (BSAM) macro  
     description 402  
     format 402  
 POINT macro  
     description 468  
     format 468  
 POST macro  
     coordinating dependant tasks 21  
     description 314  
     format 314  
 PRG operand  
     ETRACE command 74  
     ITRACE command 109  
 PRINT operand  
     EXECIO command 79  
 PRINTER operand  
     FILEDEF command 95  
 priority  
     assigning with CHAP macro 19  
     ID number 19  
 PRIV parameter  
     IUCVINI macro 289  
 private storage 25, 43  
 privilege classes  
     changing 29  
     redefining 9, 29  
 PRMLIST parameter  
     IUCVCOM macro 275  
 problem state 9, 47  
 process  
     GCS commands 43  
     spool files 54  
 PROFILE GCS 29, 31, 42  
 program  
     loading functions  
         BLDL macro 46  
         CALL macro 46  
         DELETE macro 46  
         IDENTIFY macro 46  
         LINK macro 45  
         LOAD macro 45  
         RETURN macro 46  
         SAVE macro 46  
         SYNCH macro 46  
         XCTL macro 46  
     managing 44  
     stacking 42  
 program (*continued*)  
     starting 31, 116  
     stopping 32, 107  
 protect  
     storage 44  
 PSW Key 14 44  
 PUNCH operand  
     EXECIO command 79  
     FILEDEF command 95  
 PURGE parameter  
     IUCVCOM macro 275  
 purpose of  
     GCS in z/VM 1  
 PUT (QSAM) macro  
     description 404  
     format 404  
 PUT macro  
     description 470  
     format 470

## Q

QSAM  
     data management rules 379  
 QSAM and BSAM  
     macro  
         CHECK (BSAM) 380  
         CLOSE (BSAM/BSAM) 382  
         DCB (BSAM/QSAM) 385  
         DCBD (BSAM/QSAM) 391  
         GET (QSAM) 394  
         NOTE (BSAM) 396  
         OPEN (BSAM/QSAM) 398  
         POINT (BSAM) 402  
         PUT (QSAM) 404  
         READ (BSAM) 406  
         SYNADAF (BSAM/QSAM) 409  
         SYNADRLS (BSAM/QSAM)  
             411  
         WRITE (BSAM) 413

QUAL operand  
     DLBL command 64  
 query  
     virtual machine 32  
 QUERY ADDRESS command  
     description 119  
     format 119  
 QUERY AUTHUSER command  
     description 120  
     format 120  
 QUERY command  
     description 117  
 QUERY COMMON command  
     description 121  
     format 121  
 QUERY DISK command  
     description 122  
     format 122  
 QUERY DLBL command  
     description 124  
     format 124  
 QUERY DUMP command  
     description 126  
     format 126

**R**  
 QUERY DUMPLOCK command  
   description [127](#)  
   format [127](#)  
 QUERY DUMPVM command  
   description [128](#)  
   format [128](#)  
 QUERY ETRACE command  
   description [129](#)  
   format [129](#)  
 QUERY FILEDEF command  
   description [130](#)  
   format [130](#)  
 QUERY GCSLEVEL command  
   description [131](#)  
   format [131](#)  
 QUERY GROUP command  
   description [132](#)  
   format [132](#)  
 QUERY IPOLL command  
   description [133](#)  
   format [133](#)  
 QUERY ITRACE command  
   description [134](#)  
   format [134](#)  
 QUERY LOADALL command  
   description [135](#)  
   format [135](#)  
 QUERY LOADCMD command  
   description [136](#)  
   format [136](#)  
 QUERY LOADLIB command  
   description [137](#)  
   format [137](#)  
 QUERY LOCK command  
   description [138](#)  
   format [138](#)  
 QUERY MODDATE command  
   description [139](#)  
   format [139](#)  
 QUERY parameter  
   IUCVCOM macro [275](#)  
 QUERY REPLY command  
   description [140](#)  
   format [140](#)  
 QUERY REXXSTOR command  
   description [141](#)  
   format [141](#)  
 QUERY SEARCH command  
   description [142](#)  
   format [142](#)  
 QUERY SYSNAMES command  
   description [143](#)  
   format [143](#)  
 QUERY TRACETAB command  
   description [144](#)  
   format [144](#)  
 QUERY TSLICE command  
   description [145](#)  
   format [145](#)  
 QUIESCE parameter  
   IUCVCOM macro [275](#)  
 RESUME parameter  
   IUCVCOM macro [275](#)  
 R parameter  
   GETMAIN macro [258](#)  
 R/W operand  
   QUERY DISK command [122](#)  
 RBA parameter  
   SHOWCB macro [488](#)  
   TESTCB macro [503](#)  
 RC parameter  
   FREEMAIN macro [233](#)  
   GETMAIN macro [257](#)  
   RETURN macro [322](#)  
   SETRP macro [342](#)  
 RDJFCB macro  
   description [317](#)  
   format [317](#)  
 READ (BSAM) macro  
   description [406](#)  
   execute format [408](#)  
   list format [407](#)  
   standard format [406](#)  
 READER operand  
   FILEDEF command [95](#)  
 real I/O  
   functions [49](#)  
 REAL parameter  
   STIMER macro [345](#)  
 REASON parameter  
   SETRP macro [341](#)  
 rebuilding  
   GCS nucleus [513](#)  
 receive  
   control [37](#)  
 RECEIVE parameter  
   IUCVCOM macro [275](#)  
 RECFM operand  
   FILEDEF command [96](#)  
 RECFM parameter  
   DCB (BSAM/QSAM) macro  
   388  
 RECLEN parameter  
   GENCB macro [448](#)  
   MODCB macro [463](#)  
   RPL macro [475](#)  
   SHOWCB macro [488](#)  
   TESTCB macro [503](#)  
 recovery machine [4](#), [25](#), [30](#), [51](#)  
 RECV parameter  
   CONFIG macro [194](#)  
 reenterable program save area [36](#)  
 register  
   (15)  
     return code 0 [524](#)  
     return code 12 [528](#)  
     return code 8 [525](#)  
     base [35](#)  
     linkage [35](#)  
 regular  
   segments [27](#)  
 REJECT parameter  
   IUCVCOM macro [275](#)  
 RELATED parameter  
   DELETE macro [202](#)

**RELATED** parameter (*continued*)
   
 DEQ macro 205
   
 ENQ macro 214
   
 LOAD macro 299
   
 POST macro 314
   
 WAIT macro 365

**release**
  
 disks 146

**RELEASE** command
   
 description 146
   
 format 146

**RELEASE** parameter
   
 LOCKWD macro 302

**Remote Spooling Communications Subsystem (RSCS)**
  
 operating environment 4

**REP** parameter
   
 IUCVCOM macro 275
   
 IUCVINI macro 287

**REPLY** command
   
 description 147
   
 format 147

**REPLY** operand
   
 QUERY command 140

**REPLY** parameter
   
 IUCVCOM macro 275

**reply to**
  
 messages 32

**replying to messages** 147

**REQLIST** parameter
   
 EXECCOMM macro 229

**request**
  
 information 117

**requirement**
  
 calculating storage 25

**resource coordination** 22

**response examples, notation used in** xv

**RESSTOR** macro
   
 description 320
   
 format 320

**RESTORE** parameter
   
 SYNCH macro 350

**RESTRCT** parameter
   
 CONFIG macro 194

**restrict**
  
 access to
   
 GCS supervisor 8
   
 real I/O 49
   
 storage 44
   
 supervisor state 9

**REstructured eXtended eXecutor/Virtual Machine**
  
 (REXX/VM)

creating GCS files 4

using 42

**RET** parameter
   
 DEQ macro 205
   
 ENQ macro 214

**retrieve**
  
 information from the VSAM catalog 476

**RETRIEVE** parameter
   
 GCSTOKEN macro 242

**return code placement** 159

**RETURN** macro
   
 description 322
   
 format 322

**REXX/VM** interpreter 30

**REXXSTOR** operand
   
 QUERY command 141
   
 SET command 153

**RKP** parameter
   
 SHOWCB macro 483
   
 TESTCB macro 495

**RMODE**
  
 determining 182

**RPL** macro
   
 description 472
   
 format 472

**RPL** parameter
   
 CHECK macro 425
   
 ENDREQ macro 430
   
 ERASE macro 432
   
 GET macro 451
   
 MODCB macro 461
   
 POINT macro 468
   
 PUT macro 470
   
 SHOWCB macro 487
   
 TESTCB macro 501
   
 WRTBFR macro 505

**RPLLEN** parameter
   
 SHOWCB macro 488
   
 TESTCB macro 503

**RPS** parameter
   
 DEVTYPE macro 210

**RU** parameter
   
 FREEMAIN macro 233
   
 GETMAIN macro 258

**rules of**
  
 task dispatching 19

## S

**S** parameter
   
 CHAP macro 188
   
 ENQ macro 214
   
 READ (BSAM) macro 407
   
 WRITE (BSAM) macro 414

**save area**
  
 content of 35

**nonreentrantable program** 36

**providing** 35

**reentrantable program** 36

**SAVE** macro
   
 description 324
   
 format 324

**saving**
  
 GCS nucleus 515

**scenario**
  
 of GCS in z/VM 14, 17

**SCHEDEX** macro
   
 description 326
   
 format 326

**schedule exits** 51

**SDUMP** macro
   
 description 329
   
 execute format 331
   
 list format 331
   
 standard format 329

**SDUMPX** macro
   
 description 333

**SDUMPPX** macro (*continued*)
   
 execute format [336](#)  
 list format [336](#)  
 standard format [333](#)

**SEARCH** operand
   
**QUERY** command [142](#)

**security of storage**
  
**LOCKWD** macro [50](#)  
**PGLOCK** macro [50](#)  
**PGULOCK** macro [50](#)  
**VALIDATE** macro [50](#)

**SEGMENT** macro
   
 description [338](#)  
 format [338](#)

**segment space**
  
 VSAM support [27](#)

**SEND** parameter
   
**IUCVCOM** macro [275](#)

**set**
  
 timers [46](#)

**SET** command
   
 description [149](#)

**SET DUMP** command
   
 description [150](#)  
 format [150](#)

**SET DUMPLOCK** command
   
 description [151](#)  
 format [151](#)

**SET IPOLL** command
   
 description [152](#)  
 format [152](#)

**SET** parameter
   
**AUTHNAME** macro [174](#)  
**ESPIE** macro [219](#)  
**IUCVINI** macro [286](#)  
**MACHEXIT** macro [306](#)  
**SPLEVEL** macro [343](#)  
**TASKEXIT** macro [354](#)

**SET REXXSTOR** command
   
 description [153](#)  
 format [153](#)

**SET SYSNAME** command
   
 description [154](#)  
 format [154](#)

**SET TSLICE** command
   
 description [155](#)  
 format [155](#)

**SETRP** macro
   
 description [340](#)  
 format [340](#)

**SEVER** parameter
   
**IUCVCOM** macro [275](#)

**SF** parameter
   
**ATTACH** macro [171](#)  
**LINK** macro [296, 297](#)  
**READ (BSAM)** macro [406](#)  
**WRITE (BSAM)** macro [413](#)  
**XCTL** macro [376](#)

**SGROUP** parameter
   
**CONFIG** macro [194](#)

**SHAPV** parameter
   
**ATTACH** macro [167](#)

**share in GCS**
  
 disks and files [52](#)

**Shared File System (SFS)**
  
 data management services [51](#)  
 file pool [4](#)

**shared segment**
  
 build [27](#)

**shared VTAM** [16](#)

**SHOWCAT** macro
   
 description [476](#)  
 format [476](#)

**SHOWCB** macro
   
 access control block format [480](#)  
 description [480](#)  
 exit list format [484](#)  
 request parameter list format [486](#)

**SHOWCB** parameter notation [519](#)

**SHSPL** parameter
   
**ATTACH** macro [168](#)

**Signal System Service (SSS)**
  
 signal ID [7](#)

**single user group**
  
 communicating between machines [7](#)  
 defining [4](#)  
 dump receiving machine [6](#)  
 environment [7](#)  
 recovery machine [6](#)

**SIO** operand
   
**ETRACE** command [74](#)  
**ITRACE** command [109](#)

**SKIP** operand
   
**EXECIO** command [80](#)

**SM** parameter
   
**ATTACH** macro [168](#)

**SP** operand
   
**ITRACE** command [109](#)

**SP** parameter
   
**FREEMAIN** macro [234](#)  
**GETMAIN** macro [259](#)

**SPLEVEL** macro
   
 description [343](#)  
 format [343](#)

**spool files**
  
 defining [94](#)  
 processing [54](#)

**SR** parameter
   
**SYMREC** macro [348](#)

**SSS** operand
   
**ETRACE** command [74](#)  
**ITRACE** command [109](#)

**standard format** [158](#)

**start**
  
 programs [31, 116](#)

**START** parameter
   
**AUTHUSER** macro [179](#)  
**CONTENTS** macro [197](#)  
**GENIO** macro [249](#)  
**RESSTOR** macro [320](#)  
**SEGMENT** macro [338](#)

**STARTR (start real)** [49](#)

**STARTR** parameter
   
**GENIO** macro [249](#)

**STATE** parameter
   
**SYNCH** macro [351](#)

**STEM** operand
   
**EXECIO** command [80](#)

STEP parameter  
   ABEND macro [162](#)  
 STIMER macro  
   description [345](#)  
   format [345](#)  
 STMST parameter  
   SHOWCB macro [483](#)  
   TESTCB macro [495](#)  
 stop  
   commands [107](#)  
   programs [31, 32, 107](#)  
 storage  
   anchor block [10](#)  
   calculating requirements [25](#)  
   common [5](#)  
   layout [11](#)  
 STORAGE parameter  
   SDUMP macro [329](#)  
 STRING operand  
   EXECIO command [81](#)  
 STRIP operand  
   EXECIO command [81](#)  
 STRMAX parameter  
   SHOWCB macro [483](#)  
 STRNO parameter  
   ACB macro [422](#)  
   BLDVRP macro [424](#)  
   GENCB macro [441](#)  
   MODCB macro [457](#)  
   SHOWCB macro [482](#)  
   TESTCB macro [495](#)  
 structure  
   save area [35](#)  
 subpool  
   defining [44](#)  
 subtask  
   adding [18](#)  
   discarding [18](#)  
 SUP operand  
   ITRACE command [109](#)  
 supervisor state [9, 47](#)  
 SVC operand  
   ETRACE command [74](#)  
   ITRACE command [109](#)  
 SVEAREA parameter  
   ESTAE macro [224](#)  
 SYMREC macro  
   description [348](#)  
   execute format [348](#)  
   list format [348](#)  
   standard format [348](#)  
 SYN operand  
   ETRACE command [74](#)  
   ITRACE command [109](#)  
 SYNAD parameter  
   DCB (BSAM/QSAM) macro  
     [389](#)  
   EXLST macro [434](#)  
   GENCB macro [444](#)  
   MODCB macro [459](#)  
   SHOWCB macro [485](#)  
   TESTCB macro [497](#)  
 SYNADAF (BSAM/QSAM)  
   macro

SYNADAF (BSAM/QSAM) macro (*continued*)  
   description [409](#)  
   format [409](#)  
 SYNADRLS (BSAM/QSAM)  
   macro  
   description [411](#)  
   format [411](#)  
 SYNCH macro  
   description [350](#)  
   execute format [353](#)  
   list format [352](#)  
   standard format [350](#)  
 synchronize  
   machines [50](#)  
   tasks [20](#)  
 syntax diagrams, how to read [xiii](#)  
 SYSID parameter  
   CONFIG macro [195](#)  
 SYSNAME operand  
   SET SYSNAME command [154](#)  
 SYSNAME parameter  
   CONFIG macro [193](#)  
 SYSNAMES operand  
   QUERY command [143](#)  
 SYSTEM parameter  
   ABEND macro [162](#)  
   SETRP macro [341](#)  
 Systems Network Architecture (SNA)  
   logging on at a SNA terminal [14](#)  
   network control unit [17](#)  
   path between terminal and virtual machine [14](#)  
   with GCS [1](#)  
 Systems Support Program (SSS) [3](#)  
 SZERO parameter  
   ATTACH macro [167](#)

## T

T parameter  
   RETURN macro [322](#)  
   SAVE macro [324](#)  
 TABSIZE parameter  
   CONFIG macro [194](#)  
 tailoring  
   GCS nucleus [507](#)  
 task  
   accessing authority [8](#)  
   ends  
     abnormally [22](#)  
     normally [22](#)  
   GCS  
     adding and discarding subtasks [18](#)  
     coordinating [20](#)  
     defining [17](#)  
     dispatching [19](#)  
     exit routines [22](#)  
     priority [19](#)  
     program stack for each [42](#)  
     sharing resource [22](#)  
     terminating [22, 51](#)  
 TASKEXIT macro  
   description [354](#)  
   execute format [357](#)  
   list address format [357](#)

**TASKEXIT** macro (*continued*)  
 list format 356  
 standard format 354  
 termination routines 22  
**terminal** macro  
 appealing abends 22  
**test field**  
 ACB 491  
 exit list 497  
 RPL 500  
**TEST** parameter  
 LOCKWD macro 302  
 SPLEVEL macro 343  
**TESTCB** macro  
 description 491  
 format 490, 496, 499  
**TESTCB** parameter notation 519  
**TEXT** operand  
 REPLY command 147  
**TIME** macro  
 description 359  
 format 359  
 time zone support 201  
 time-of-day (TOD) clock  
 description 47  
 timer management  
 STIMER macro 46  
 TIME macro 47  
 TTIMER macro 47  
**TO \*** operand  
 GDUMP command 99  
**TO USERID** operand  
 GDUMP command 99  
**TOD** parameter  
 STIMER macro 346  
**TOKEN** parameter  
 TOKEN parameter 243  
**trace**  
 external tracing 73  
 GTRACE events 108  
 trace table size 26, 27  
**TRACETAB COMMON** operand  
 CONFIG command 62  
**TRACETAB** operand  
 QUERY command 144  
**TRACETAB PRIVATE** operand  
 CONFIG command 62  
**TRACPRI** parameter  
 CONFIG macro 195  
**transfer**  
 authorized program control  
 AUTHCALL macro 47  
 AUTHNAME macro 47  
 data to VSCS 16  
**TRANSID** parameter  
 GENCB macro 449  
 MODCB macro 463  
 RPL macro 475  
 SHOWCB macro 488  
 TESTCB macro 503  
**TSLICE** operand  
 QUERY command 145  
**TTIME** macro  
 description 361

**TTIME** macro (*continued*)  
 format 361  
**TYPE** parameter  
 BLDVRP macro 424  
 CLOSE macro 427  
 DLVRP macro 429  
 WRTBFR macro 505  
**U**  
**unauthorize**  
 applications 44  
**UPDAT** parameter  
 OPEN (QSAM/BSAM) macro 398  
 RDJFCB macro 317  
**USER** parameter  
 SETRP macro 341  
**user-supplied routines** 265, 267  
**UWORD** parameter  
 AUTHCALL macro 172  
 AUTHNAME macro 175  
 GENIO macro 248  
 IUCVCOM macro 275  
 IUCVINI macro 288  
 MACHEXIT macro 306  
 SCHEDEX macro 326  
 TASKEXIT macro 354

**V**  
**V** parameter  
 FREEMAIN macro 234  
**VALIDATE** macro  
 description 362  
 format 362  
 validate requests for storage access 50  
**VAR** operand  
 EXECIO command 81  
**VC** parameter  
 GETMAIN macro 258  
**VDEV** operand  
 ACCESS command 59  
 RELEASE command 146  
**virtual I/O**  
 perform 49  
**virtual machine**  
 groups  
 building 4  
 communication within 7  
 configuration file 4  
 defining 4  
 joining 5, 30  
 single user group 4  
 storage layout 25  
 storage  
 dumping 98  
 VMSIZE 25  
**Virtual Storage Access Method (VSAM)**  
 catalog  
 diagram of 64  
 retrieving information 476  
 data management services 51  
 data sets 28

Virtual Storage Access Method (VSAM) (*continued*)  
generating macro format 518  
GROUP user IDs requiring storage for VSAM 103  
macro  
    ACB 418  
    BLDVRP 423  
    CHECK 425  
    CLOSE 427  
    DLVRP 429  
    ENDREQ 430  
    ERASE 432  
    EXLST 434  
    GENCB 438  
    GET 451  
    MODCB 453  
    OPEN 465  
    POINT 468  
    PUT 470  
    RPL 472  
    SHOWCAT 476  
    SHOWCB 480  
    TESTCB 491  
    WRTBFR 505  
macro addresses 518  
macro library 54  
operating under GCS 517  
processing 54  
supporting  
    saving segments 27  
Virtual Telecommunications Access Method (VTAM)  
description 2  
operating environment 4  
running in a group machine 16  
running on GCS 16  
shared VTAM 16  
VL parameter  
    ATTACH macro 166  
    CALL macro 184  
    LINK macro 294  
    XCTL macro 377  
VSAM operand  
    DLBL command 64  
VSAM parameter  
    GENCB macro 438, 443, 446  
VSAMSEG parameter  
    CONFIG macro 195  
VTAM SNA Console Support (VSCS)  
running  
    VTAM 16  
sending information to VTAM 16  
VU parameter  
    FREEMAIN macro 234  
    GETMAIN macro 258

## W

WAIT macro  
    coordinating dependant tasks 21  
    description 365  
    format 365  
WAIT parameter  
    STIMER macro 345  
WAREA parameter  
    GENCB macro 441, 444, 449

WKAREA parameter  
    SETRP macro 341  
WRITE (BSAM) macro  
    description 413  
    execute format 415  
    list format 414  
    standard format 413  
WRTBFR macro  
    description 505  
    format 505  
WTO macro  
    description 368  
    execute format 369  
    list format 368  
    standard format 368  
WTOR macro  
    description 370  
    execute format 371  
    list format 371  
    standard format 370

## X

XA mode  
    running 2  
XCTL macro  
    description 373  
    execute format 376  
    list format 375  
    standard format 373  
XCTL parameter  
    ESTAE macro 224

## Z

ZONE operand  
    EXECIO command 81







Product Number: 5741-A09

Printed in USA

SC24-6289-74

