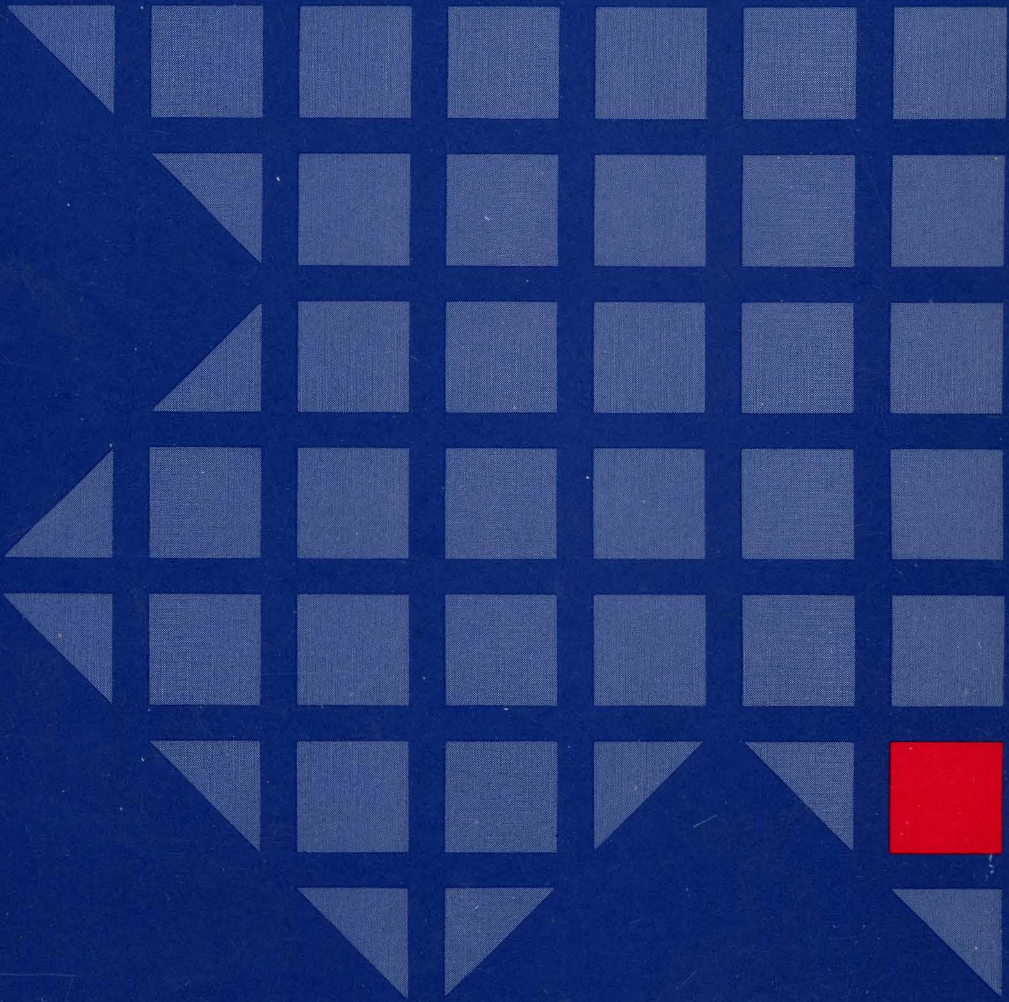
IBM

Virtual Machine/Extended Architecture™
System Product

SC23-0433-0

# Group Control System Command and Macro Reference

VM/XA™ SP Release 2



IBM

Virtual Machine/Extended Architecture™
System Product

SC23-0433-0

# Group Control System Command and Macro Reference

VM/XA™ SP Release 2

**First Edition (November 1988)**

This edition applies to Release 2 of the IBM Virtual Machine/Extended Architecture System Product (VM/XA SP) Licensed Program 5664-308. Changes are made periodically to the information herein; before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370, 30xx, 4300, and 9370 Processors Bibliography,* GC20-0001, for the editions that are applicable and current.

# Preface

## Purpose

This publication is a reference manual; it contains detailed information on the purpose and use of all Group Control System (GCS) commands and macro instructions.

This publication presents the format, syntax rules, and parameter descriptions, plus factual and feedback information on all GCS commands and macro instructions. In special cases, this book also gives an example of exactly how a programmer might issue a command or macro instruction.

## Audience

This publication is for those who write programs to run under GCS; this includes system and application programmers in both customer and IBM development environments.

## Related Publications

See the Bibliography at the back of this publication.

# Contents

# Chapter 1. Introduction

## A Quick Definition

The Group Control System (GCS) is:

- A component of the Virtual Machine/Extended Architecture™ System Product (VM/XA™ SP)
- A virtual machine supervisor
- An interface between program products, like Virtual Telecommunications Access Methods (VTAM) and Remote Spooling Communications Subsystem (RSCS), and the system's control program (CP) (Figure 1).

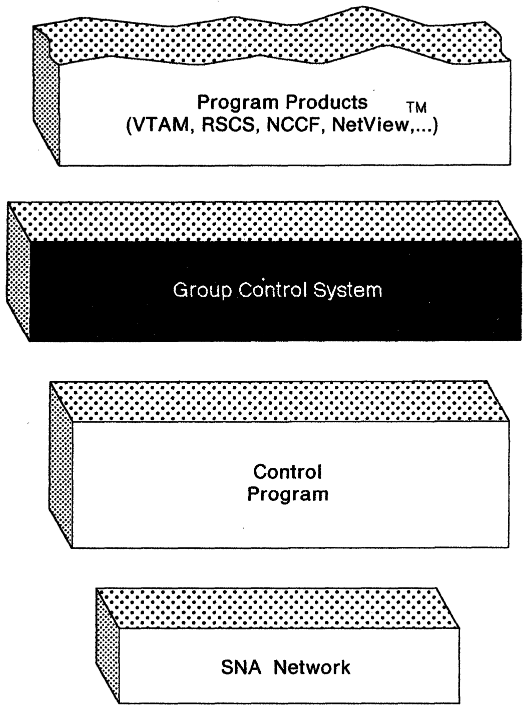


Figure 1. GCS, an Interface between Program Products and CP

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

Virtual Machine/Extended Architecture, VM/XA, and NetView are trademarks of the International Business Machines Corporation.

## The Whole Picture At A Glance

Figure 2 on page 4 shows a *conceptual* view of how the GCS can fit into your VM/XA SP environment. Familiar elements in the picture include:

- CP, at the bottom, a base for the rest of the system to build on
- Virtual machines, at the top, running various applications
- Conversational monitor system (CMS), at the left, an interactive VM/XA SP component that runs on CP
- A route to the SNA network, lower right, 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 VM/XA SP component on CP.

GCS may appear to offer some of the services offered by IBM's Multiple Virtual System (MVS). Granted, there are similarities between the two, but there are also some very significant differences in function and use. If an application is to run successfully under GCS, it must conform to GCS as discussed in this book.

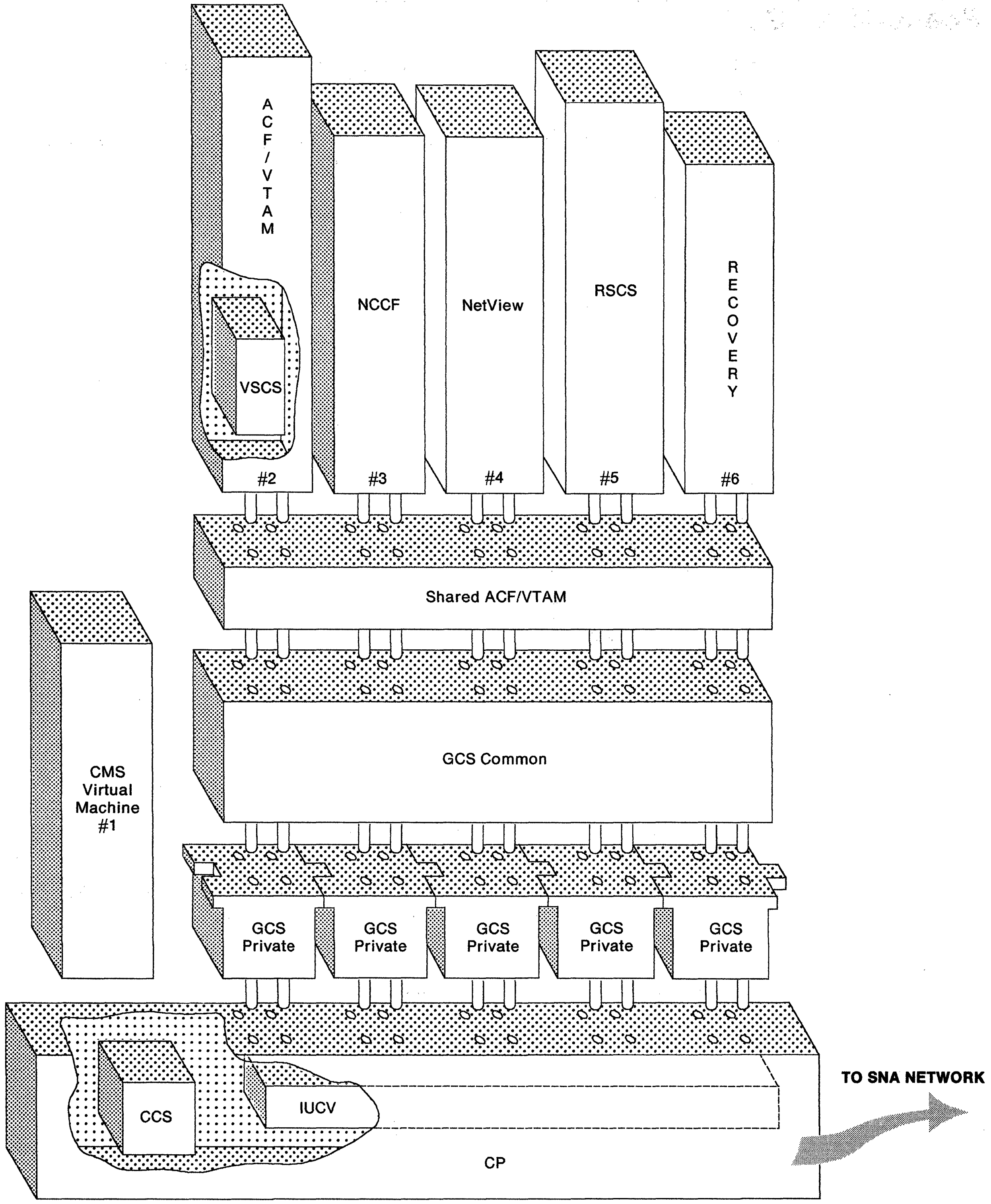


Figure 2. GCS in VM/XA SP

This diagram illustrates 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.

## A Scenario for GCS

The following scenario shows how GCS helps support standalone SNA/CCS communications.

First, you log on from a SNA/CCS terminal and IPL CMS (Figure 3). Neither you, as a user, nor CMS needs to know that it is a SNA/CCS terminal.

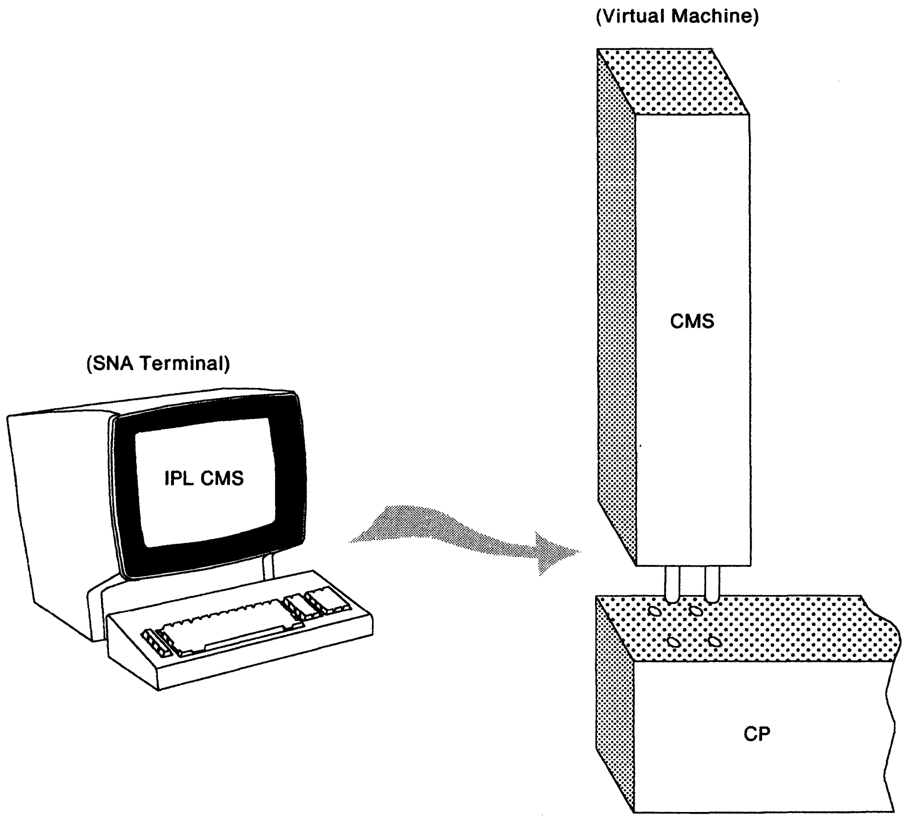


Figure 3. Initializing CMS from a SNA/CCS Terminal. On the right, you IPL CMS from your virtual console, a SNA/CCS terminal. On the left, CMS begins running in your virtual machine.

CMS responds to your commands. Being an interactive system, it communicates back and forth with you via this terminal. The information exchange seems to happen easily enough. But because you have a SNA/CCS terminal, the path from your console to CMS is a complex one, involving GCS, ACF/VTAM, and SNA/CCS somewhere in between.

## The Path between System and Console

Let's say CMS begins communicating with your console by issuing:

```
Start I/O (SIO)
```

Or, a CMS application like XEDIT issues:

```
DIAGNOSE code X'58'
```

The instruction leaves your virtual machine, and CP intercepts it (Figure 4).

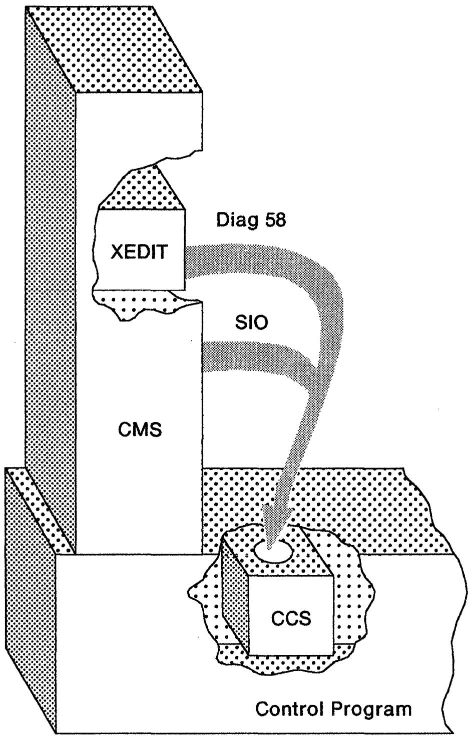


Figure 4. CP Intercepts Instructions from the Virtual Machine

After decoding and extracting the instruction's pertinent information, CP prepares to send data out on the network. A component in CP called CCS does the actual sending.

From CCS, the data passes to a virtual machine running VSCS. (In the Figure 5 example, the VTAM machine runs VSCS.[1]) The transfer from CCS to VSCS takes place through another CP facility, Inter-User Communications Vehicle (IUCV).

[1] VSCS also may run in its own virtual machine.

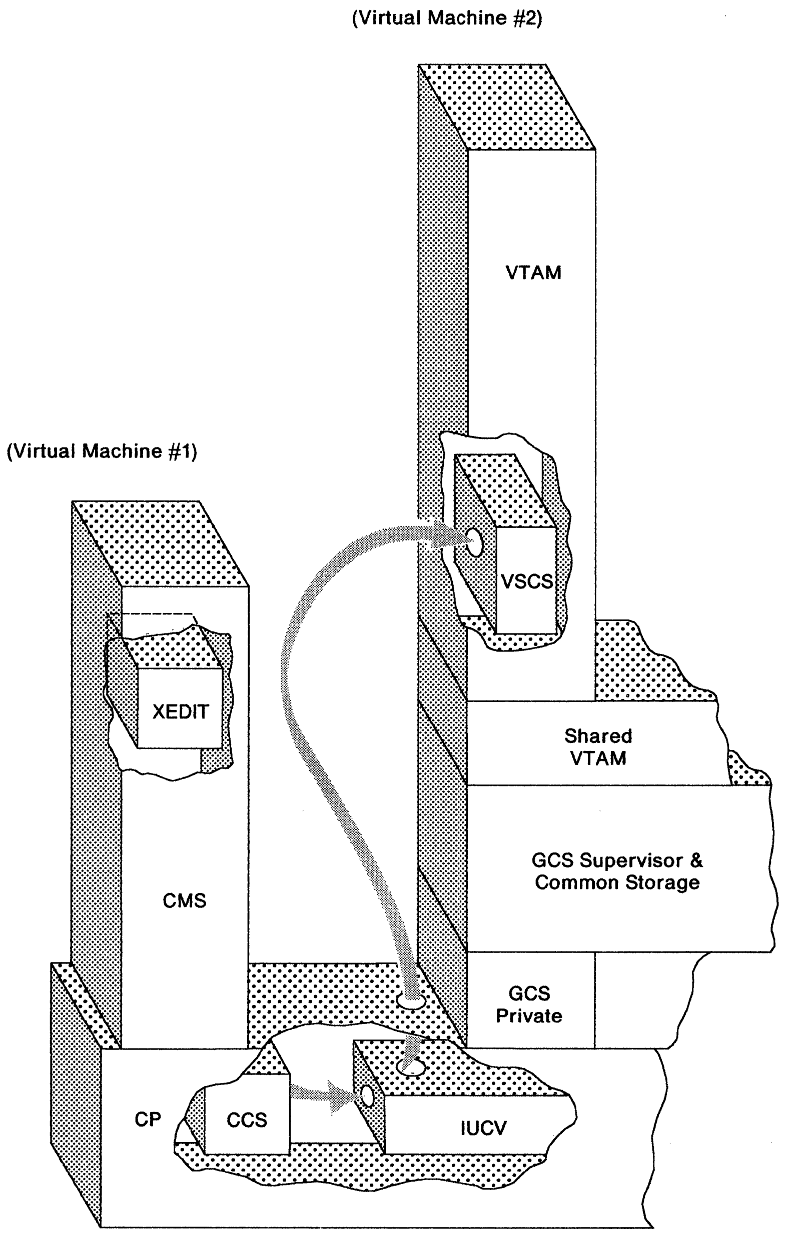


Figure 5. Transferring Data to the Machine Running VSCS. Using IUCV, CCS sends information to the virtual machine where VSCS is running. In this case, it is the VTAM machine.

Figure 5 shows the VTAM virtual machine running on GCS. In a VM/XA SP system with SNA/CCS 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 6 shows what happens after CCS 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.

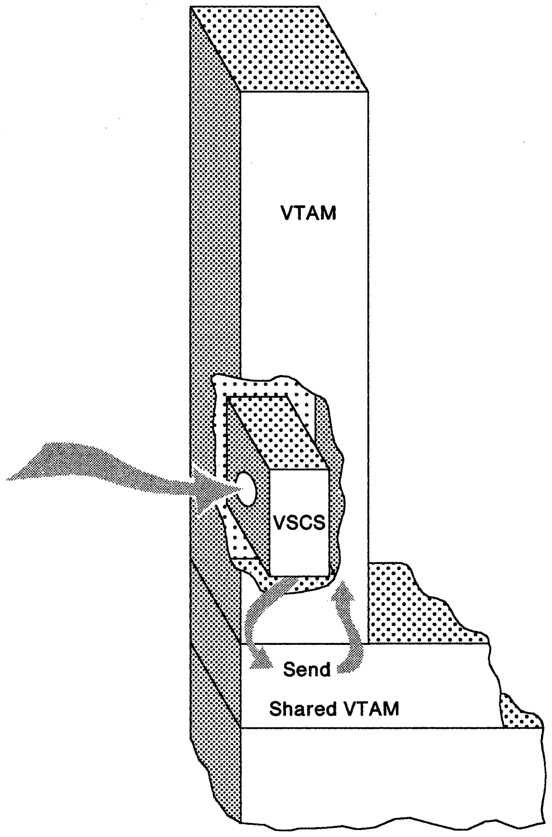


Figure 6. Path of Data Moving through the VTAM Machine. The VSCS component receives data from CCS, processes it, and sends it into VTAM's control.

From VTAM, the information travels toward your terminal (Figure 7 on page 9). 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 has charge of sending the data through the SNA network to your virtual console.

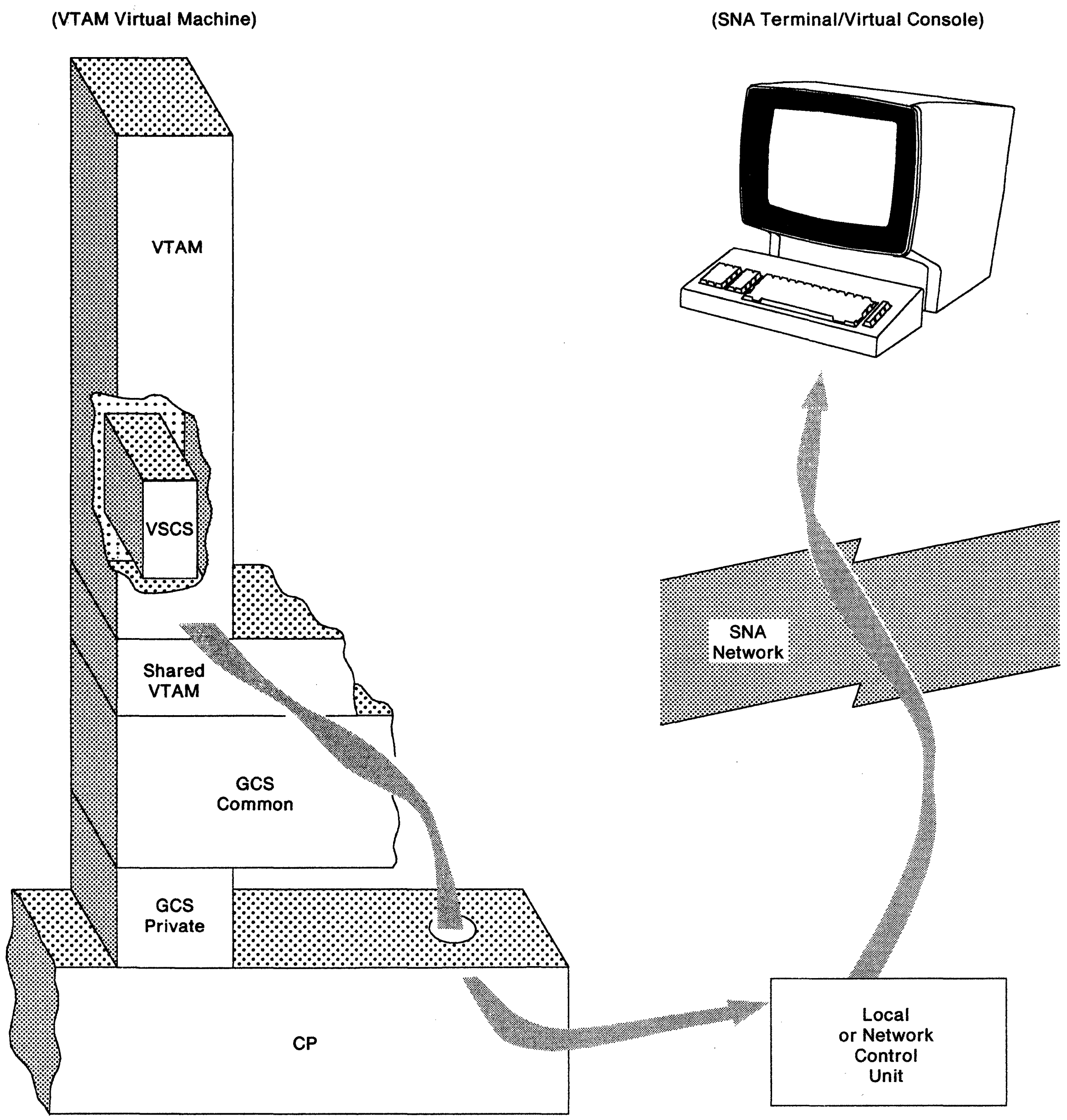


Figure 7. Data Traveling from VTAM to the Virtual Console. VTAM, at the top, sends data to GCS. GCS relays it out to the SNA network and finally to the console.

## 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 as follows:

| Register | Conventional Purpose |
|---|---|
| 0 and 1 | Used to pass parameters to the supervisor or to a called program. Some system macro instructions 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. Once 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 macro instructions 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 VM/XA SP, 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 macro instructions 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, since 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 macro instructions 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 in which the contents of its registers are saved by the program it calls. A register save area is eighteen 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). |
| 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 store multiple (STM) assembler instruction or the SAVE macro instruction. 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. Refer to "SAVE" on page 147 for more detailed information on the SAVE macro.

## An 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 via 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.

## An 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 instruction 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 instruction, 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 of its own storage via the GETMAIN macro instruction. 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 via the GETMAIN macro instruction, then it must release that storage via the FREEMAIN instruction.

## GCS Macro Instruction Formats

Generally, there are four possible formats in which macro instructions are available, the:[2]

- Standard Format
- List Format
- List Address Format
- Execute Format.

**Note:** Not every GCS macro instruction 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 macro instructions listed in this book differ from this somewhat. Before you use these instructions, be certain to review the entry titled "Using VSAM under GCS" on page 280.

The entry in this book devoted to each macro instruction tells you exactly which of these formats applies and provides more detailed information. In general, the significance of each format is as follows:

**The Standard Format**
Generates an in-line parameter list to the macro. It also generates non-reentrant code that executes the function as part of the macro expansion.

**The List Format**
Generates an in-line parameter list to the macro but generates no code that executes the function.

**The 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.

**The 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.

## GCS Macro Formatting Conventions

You will notice that each macro instruction entry is accompanied by a 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. Generally speaking, there are only two places where a blank space can appear in a macro instruction. These are between the label and the instruction, and between the instruction and its first parameter. Moreover, you probably notice that the parameters themselves are not delimited by blanks, but by commas. In these respects, macro instructions resemble assembler language instructions rather closely.

Let us illustrate this by looking at a fictitious macro instruction called DUCK. The DUCK macro instruction takes three parameters: A, B, and C. And, like most other instructions, an optional label can be applied.

Its format box looks like this:

| | | |
|---|---|---|
| [label] | DUCK | A,B,C=some number |

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 macro instructions 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 commas delimit the parameters.

Few macro instructions 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 instruction, GOOSE, has two parameters, one of which is optional.

Its format box looks like this:

| | | |
|---|---|---|
| [label] | GOOSE | [A=some number,]B=some other number |

You could code GOOSE like this:

```
GOOSE B=77
```

This is perfectly valid since the brackets ( [ ] ) around the A parameter indicate that you can omit it if you choose. Note that you did not supply a comma before the B parameter, since 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 macro instructions stack optional parameters in a list.

The fictitious HORSE macro format box looks like this:

| | | |
|---|---|---|
| [label] | HORSE | $\mathtt{A,B} \begin{bmatrix} \mathtt{,C} \\ \mathtt{,D} \\ \mathtt{,E} \end{bmatrix}$ |

Notice the large set of brackets around the C, D, and E parameters. These brackets mean two things. First, all three of the parameters are optional. You can ignore the bracketed 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 bracketed 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, since you chose to omit all of the optional parameters.

Some macro instructions force you to make a choice from among a stacked list of options.

The MOOSE macro format box looks like this:

| | | |
|---|---|---|
| [label] | MOOSE | H,P,M, $\begin{Bmatrix}\text{X}\\\text{Y}\\\text{Z}\end{Bmatrix}$ |

Notice the large braces ( { } ) around the X, Y, and Z parameters. The braces mean that you have one choice among the three parameters. But, this is not an optional choice, it is a choice that you must make. So,

```
MOOSE H,P,M
```

is incorrect, since you did not select from among the list enclosed by braces. Likewise,

```
MOOSE H,P,M,X,Z
```

is incorrect because you selected more than one parameter from the list. But,

```
MOOSE H,P,M,Z
```

is correct because you made your choice and it was only one parameter.

Sometimes brackets and braces are used together. Usually, though not always, they involve parameters that take effect by default if something is not specified.

The MACKEREL macro format box looks like this:

| | | |
|---|---|---|
| [label] | MACKEREL | J,L,Q $\left[\text{,S=}\begin{Bmatrix}\underline{\text{YES}}\\\text{NO}\end{Bmatrix}\right]$ |

Notice that a set of braces surrounds the parameters YES and NO. Then, a set of brackets embraces these, as well as the S parameter. To further complicate matters, the YES parameter is underlined.

It is not that difficult to figure this out if you just remember that brackets mean you have an option to choose or not choose and that braces mean you must choose. The brackets here simply mean that you can choose the S parameter or ignore it. However, if you do choose the S parameter, then the braces mean that you must choose either the YES parameter or the NO parameter. And the line under the YES parameter means that if you ignore the S parameter, then S=YES will be in effect by default. So,

```
MACKEREL J,L,Q,S
```

is incorrect, because you chose the S parameter but did not choose either YES or NO.

```
MACKEREL J,L,Q
```

is correct, since you omitted the S parameter altogether, allowing S=YES to take effect by default. Likewise, both

```
MACKEREL J,L,Q,S=YES
MACKEREL J,L,Q,S=NO
```

are correct, since 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 instruction. 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 alphameric 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 unused high-order bits in the register should be re-set to zero. You can express the register number symbolically or via 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.

# Chapter 2. Group Control System (GCS) Commands

## GCS Commands

### Command Formats

**Braces { }**
Indicate that you must choose one of the items inside.

**Brackets [ ]**
Indicate that you may opt to choose any one or none of the items inside.

**Capital letters**
Represent letters that you must type.

**Lowercase letters**
Either finish spelling a keyword or else represent variable values that are explained in the accompanying text.

**Underlined values**
Represent defaults. If you enter nothing in their places, they automatically become the effective values.

## Console and Command Support

### Communicating through the Console

Any terminal supported by VM/XA SP can be a GCS console (ASCII and 3270 devices included). GCS virtual machine operators use their consoles to communicate with:

- The GCS supervisor (using the GCS commands)
- Applications running in the machine (using application commands defined with the LOADCMD command). Refer to "LOADCMD" on page 46 for information.

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 that the GCS virtual machine operator can respond. Unlike CMS, GCS lets programs keep running even though you might owe them many replies. |

See the macros "WTO" on page 166 and "WTOR" on page 168 for descriptions.

### Issuing Commands to GCS

A user can issue commands in the following 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. A user invokes it with a console command or with a GCS CMDSI macro or from another command file. PROFILE GCS (if the user has one) is a particular type of command file that executes automatically when the user IPLs GCS.

Besides GCS commands, a GCS command file can contain REXX statements and functions. The VM/XA SP System Product Interpreter processes these REXX statements and, in fact, the entire GCS command file. Therefore, most REXX capabilities in CMS also apply with GCS. The differences with GCS are:

- REXX programs (EXECS) have a filetype 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 the use issues PULL, and a task's program stack is empty, the user receives a message at the console (using the WTOR macro) that asks for the necessary input.
- ADDRESS GCS replaces ADDRESS CMS. (The default for REXX is ADDRESS GCS.) It acts the same as ADDRESS CMS, providing full command resolution, which includes 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.
- A user can cancel command files using HX. The commands TS, TE, and HI, which worked with REXX in CMS, have no support on GCS.
- A user can invoke REXX programs from assembler language programs with the CMDSI macro. FILEBLK, a parameter for CMDSI, contains the address of the file block. FILEBLK is useful for executing in-storage command files, executing command files with filetypes other than GCS, and establishing an initial subcommand environment.
- Non-REXX programs can share variables with REXX programs through the EXECCOMM macro. The GCS EXECCOMM macro has the same capabilities as the CMS 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. The commands are: ETRACE, ITRACE, HX, QUERY, REPLY, and GDUMP. Immediate commands always execute first; therefore, an EXEC of the same name would never execute.

Refer to *VM/XA SP System Product Interpreter Reference* for more information about REXX.

## Processing GCS Commands

GCS processes commands much the same way as CMS does. Commands can be processed as follows:

- **Executed immediately**

  An immediate command is one that gets executed as soon as you issue it. If a user enters commands with the CMDSI macro or any one of the following six immediate GCS commands:

  ETRACE (see page 31)
  GDUMP (see page 38)
  HX (see page 42)
  ITRACE (see page 43)
  QUERY (see page 51)
  REPLY (see page 58).

  they are not stacked; GCS processes them right away, even if the user enters them while another command is being executed.

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

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

  your system will process any immediate commands first. In this case, you would receive results from `immed cmd` before the results from `cmd1`. If an EXEC or routine has named the same name as an immediate command, the immediate command is executed. This differs from the way CMS processes commands.

- **Stacked and wait their turn**

  This is the regular procedure. Nonimmediate commands entered are processed serially. Refer to "LOADCMD" on page 46 for the format of the LOADCMD command. When the current command finishes, 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 a user define, start, terminate, and control an application. Some commands are unique to GCS; others are existing or modified CMS commands:

| Unique GCS Commands | GCS/CMS Commands |
|---|---|
| ETRACE | ACCESS |
| ITRACE | DLBL |
| LOADCMD | FILEDEF |
| REPLY | GLOBAL |
| GDUMP | HX |
| | OSRUN |
| | QUERY |
| | RELEASE |

In addition, a user can define "application" commands with the LOADCMD command.

# ACCESS

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

Unlike the CMS ACCESS command, you cannot specify options.

The format of the ACCESS command is:

```
ACcess    ┌                                   ┐
          │ vdev mode[/ext [fn [ft [fm]]]]    │
          │ 191   A                           │
          └                                   ┘
```

**vdev**
Makes available the disk at the specified virtual address. The default value is 191. Valid addresses are X'0001' through X'FFFF'.

**mode**
Assigns a 1-character filemode 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'll be able to read. Entering an asterisk (*) in any one of these fields indicates that you want all filenames or filetypes or filemode numbers (except 0) to be in your user file directory. You can specify filename, filetype, and filemode fields only for CMS-formatted disks that you've accessed as read-only extensions. For example, to specify a filemode, use a letter and a number:

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

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

## Messages

```
CSIACC005S Virtual storage capacity exceeded  RC=104
CSIACC006E Invalid parameter 'parameter'  RC=24
CSIACC012E No options allowed  RC=24
CSIACC021E Invalid mode 'mode'  RC=24
CSIACC414E Disk vdev not properly formatted for ACCESS  RC=16
CSIACC415E Invalid device address 'vdev'  RC=24
CSIACC422E vdev already accessed as Read/Write 'mode' disk
           RC=36
CSIACC423I mode (vdev) {R/O | R/W}
CSIACC424I vdev mode released
CSIACC425I vdev replaces mode (vdev)
CSIACC426I vdev also = mode disk
CSIACC427S mode (vdev) device error  RC=100
CSIACC428S mode (vdev) not attached  RC=100
CSIACC429E File fn [ft [fm]] not found.  Disk mode (vdev)
           will not be accessed  RC=28
CSIACC430W OS disk - Fileid specified is ignored  RC=4
CSIROS005S Virtual storage capacity exceeded
CSIROS423I mode (vdev) {R/O | R/W} {-OS | -DOS}
CSIROS426I vdev also = mode {-OS | -DOS} disk
```

For more information on messages, see the *VM/XA SP System Messages and Codes Reference*.

## DLBL

### Define VSAM Files Used for Program Input/Output

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

**Note:** For non-VSAM file definitions, you use the "FILEDEF" on page 34. VSAM itself does not always require file definition statements. To learn when file definitions are necessary, see the *VSE/VSAM Programmer's Reference*.

The format of the DLBL command is:

```
DLBL  ┌                                                                     ┐
      │ddname mode ┌DSN qual1[[.]qual2....qualn]┐ [(optionB optionC [)]]  │
      │            │DSN ?                       │                          │
      │            └                            ┘                          │
      │ddname CLEAR                                                         │
      │  *                                                                  │
      └                                                                     ┘
                                         optionB:       optionC:
                                         [PERM]         [VSAM]
                                         ┌        ┐     [MULT]
                                         │CHANGE  │
                                         │        │     [CAT catdd]
                                         │NOCHANGE│
                                         └        ┘     [BUFSP nnnnnn]
```

**ddname**
A one- to seven-character program ddname. 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.

If you have ddnames over seven characters long, be aware that only the first seven characters get processed. Should you have two different files with the same first seven letters and try to execute them both, you'll receive an error message when GCS opens the second file.

**mode**
A letter representing the filemode of a VSAM disk and, optionally, a filemode number. You must specify a letter, and it must refer to a disk that's already accessed. The filemode number, however, is optional. If you don't 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.

**CLEAR**
Removes any existing 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 '________' not accessed
```

**DSN**
Specifies that this is a VSAM file.

**? (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.

**qual1.qual2....qualn**
A unique name associated with the file on the volume. It can be from one to 44 characters of alphameric 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 to 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 alphameric (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.

## Option B:

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

**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.

## Option C:

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

**MULT**
Indicates that you want to enter volume specifications that refer to an existing multivolume VSAM data set. Often, VSE/VSAM requires no MULT information; see the *VSE/VSAM Programmer's Reference* to find out when it's required.

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 58) and the following rules apply:

- All the disks you refer to must be mounted and accessed when you issue 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 don't need to specify them in alphabetical order.

**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 29.) The following figure shows which catalog to use:

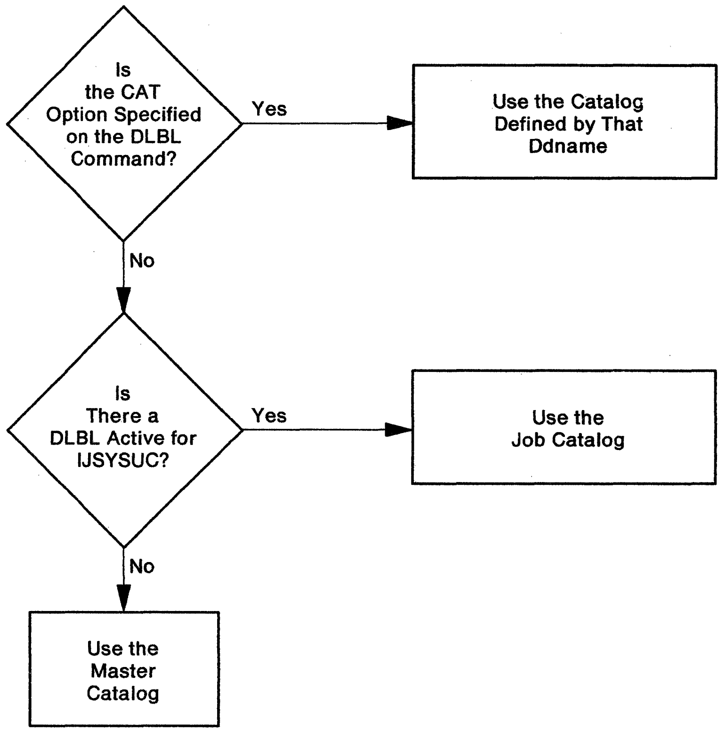

Is the CAT Option Specified on the DLBL Command?
Yes
Use the Catalog Defined by That Ddname
No
Is There a DLBL Active for IJSYSUC?
Yes
Use the Job Catalog
No
Use the Master Catalog

Figure 8. Determining Which VSAM Catalog to Use

**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 *VM/XA SP CMS Command Reference.*

## Messages

```
CSIDLB001E Invalid option 'option'  RC=24
CSIDLB002E Invalid parameter 'parameter'
           in the option 'option' field  RC=24
CSIDLB003E 'option' option specified twice  RC=24
CSIDLB004E 'option1' and 'option2' are conflicting
           options  RC=24
CSIDLB005S Virtual storage capacity exceeded
CSIDLB006E Invalid parameter 'parameter'  RC=24
CSIDLB017E Disk {mode/vdev/volumeid} not accessed  RC=37
CSIDLB021E Invalid mode 'mode'  RC=24
CSIDLB302E Parameter missing after DDNAME  RC=24
CSIDLB303I No user defined DLBL's in effect
CSIDLB305I DDNAME 'ddname' not found.  CLEAR not executed
CSIDLB310R Enter data set name:
CSIDLB311E Invalid data set name  RC=24
CSIDLB312R Enter volume specifications:
CSIDLB313E Invalid DDNAME 'ddname'  RC=24
CSIDLB314I Maximum number of disk entries recorded
```

```
CSIDLB315E Catalog DDNAME 'ddname' not found
CSIDLB316E mode disk is in CMS format; invalid for VSAM
           data set
CSIDLB317I Job catalog DLBL cleared
CSIDLB318I Master catalog DLBL cleared
CSIDLB345I No option specified  RC=24
```

For more information on messages, see the *VM/XA SP System Messages and Codes Reference*.

## Examples

1. To display all file definitions in effect for your disks, enter the following command:

   ```
   ====> dlbl
   ```

   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. To identify a file named "infile" on your C mode disk and be prompted for additional disk mode letters, issue:

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

   When processing is completed, you will receive the following message:

   ```
   nn CSIDLB312R  Enter volume specifications:
   ```

   where nn is a reply ID number. You enter the requested disk mode letters using the REPLY command ("REPLY" on page 58) and this reply ID. For example, if you want to refer to disks accessed at modes D, E, F, and G, you enter:

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

   To terminate the command, you must enter the following command:

   ```
   reply nn
   ```

   If you don't enter this command, 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.

   To identify a VSAM master catalog, named MASTCAT, for the terminal session, enter:

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

   To identify a VSAM job catalog, named MYCAT, for the terminal session, enter:

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

   To identify a VSAM file INTEST1 that is cataloged in the job catalog MYCAT as TEST.CASE, enter:

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

To identify an additional catalog TESTCAT which has an entry in the master catalog, enter:

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

Since you specified a job catalog (MYCAT) earlier, you must use the CAT option to make sure that the master catalog IJSYSCT gets used instead.

To identify an input file INFILE cataloged in your catalog TESTCAT, which was identified with a ddname of CAT3 on the previous DLBL command, enter:

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

## ETRACE

### Enable or Disable the Recording of Events in a System Data File for a Virtual Machine or Virtual Machine Group

GCS supports external tracing — the recording of events in a system data file. 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. Likewise, you can specify a certain list of options for one virtual machine and a totally different set of options for all other virtual machines in the group.

Before any external tracing actually takes place, though, a class C user must issue the CP TRSOURCE command for the virtual machine(s) to be traced.

The format of the ETRACE command is:

```
ETrace   ┌┌ ┌        ┐ ┌     ┐┐┐
         ││ │ DSP    │ │     │││
         ││ │ EXT    │ │     │││
         ││ │ FRE    │ │     │││
         ││ │ GET    │ │     │││
         ││ │ I/O    │ │     │││
         ││ │ PRG    │ │     │││
         {│ │ SIO    │ │ OFF ││}   [GRoup]
         ││ │ SSS    │ │     │││
         ││ │ SVC    │ │     │││
         ││ │ GTrace │ │     │││
         ││ └        ┘ │     │││
         ││ [ ALL    ] │     │││
         │└            └     ┘┘│
         └[ END ]               ┘
```

**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 invoked through SVC and Branch Entry calls.

**GET**
: Enables or disables external tracing of GETMAIN events invoked through SVC and Branch Entry calls.

**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.

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

**ALL**
Enables or disables external tracing of all events described above.

**OFF**
Disables external tracing of events for the specified type(s).

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

**END**
Disables external tracing of all events.

**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 even though ETRACE with the GROUP operand was specified by another authorized virtual machine.

## Messages

```
CSIYTE001E Invalid option 'option'  RC=4
CSIYTE009E Operand missing or invalid  RC=4
CSIYTE509I ETRACE set ON for event-type(s)
CSIYTE510I ETRACE set ON for event-type(s) for GROUP
CSIYTE511I ETRACE set OFF for event-type(s)
CSIYTE512I ETRACE set OFF for event-type(s) for GROUP
CSIYTE513E ETRACE GROUP option is in effect for event-type(s)
           RC=8
```

For more information on messages, see the *VM/XA SP System Messages and Codes Reference*.

The meanings of return codes for these messages are:

| Return Code | Meaning |
|---|---|
| 00 | The specified ETRACE events have been successfully enabled or disabled. |
| 04 | An invalid operand was specified, or an unauthorized user specified the GROUP operand. four request was ignored. |
| 08 | 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. |

## Examples

In the following example, the first ETRACE command requests that all types of events for the issuer's virtual machine be recorded in a system data file. The second ETRACE command was issued to disable external tracing of I/O and program interrupts for the issuer's virtual machine.

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

The following example requests that events, such as task dispatches, I/O interrupts, and GCS supervisor I/O requests, be recorded externally for the virtual machine group. The individual who issues this command must be an authorized user, since the request is for the group.

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

The example below requests that external tracing of events in a system data 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.

```
etrace end
```

# FILEDEF

## Define CMS Format Files and Spool Data Files

Application programs usually require some "setting up" before you try to start and run them. The FILEDEF command is one of the preliminary commands normally issued to prepare a program for execution. You issue it to define CMS format files and spool data files used by the program.

The format of the FILEDEF command is:

```
FIledef  ┌           ┌                                                ┐ ┐
         │           │ PRinter [(optionA OPTCD j[)]]                  │ │
         │           │                                                │ │
         │           │ PUnch [(optionA[)]]                            │ │
         │           │                                                │ │
         │           │ Reader [(optionA[)]]                           │ │
         │           │      ┌            ┌  ┐┐                        │ │
         │ {ddname}  │ DISK │fn   ft     │fm││ [(optionA optionB[)]]  │ │
         │ {*     }  │      │FILE ddname │A1││                        │ │
         │           │      └            └  ┘┘                        │ │
         │           │ DUMMY [(optionA[)]]                            │ │
         │           │ CLEAR                                          │ │
         └           └                                                ┘ ┘
                                   option A:        option B:
                                   [PERM]           [DISP MOD]
                                   ┌         ┐
                                   │CHANGE   │      [DSORG PS]
                                   │         │
                                   │NOCHANGE │
                                   └         ┘
                                   [RECFM a ]
                                   [LRECL nnnnn]
                                   ┌             ┐
                                   │BLOCK nnnnn  │
                                   │BLKSIZE nnnnn│
                                   └             ┘
```

(CHANGE, FILE, ddname (after FILE) and A1 are underlined as defaults.)

**ddname**
*****

The name of the file, as referred to in your program. ddname can contain from one to eight alphameric characters. However, the first one must be alphabetic or national.

If you specify an asterisk (*) in place of ddname and follow it with the CLEAR operand, all file definitions that you did not enter with the PERM option will be cleared.

**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.")

**DISK fn ft [fm]**
Specifies that the virtual I/O device is a disk. `fn` and `ft` are CMS fields. If `fm` is the filemode of an OS disk, `fn` and `ft` represent the only two qualifiers of an OS data set name. If you specify `fm` as an asterisk (*), all disks get searched. You cannot use this form unless the OS data set name or VSE field conforms to the OS naming convention (1- to 8-byte qualifiers, separated by periods, up to a maximum of 44 characters, including periods). Moreover, the data set name can have only two qualifiers. If you omit `DISK fn ft`, the default is `FILE ddname A1`.

**DUMMY**
Indicates that no real I/O takes place for the data set.

**CLEAR**
Removes any existing definition for the specified ddname. You should clear ddnames before defining them to make sure the ddname doesn't already exist. Doing that nullifies any operations previously defined with the ddname.

## Option A

**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 don't specify PERM, the definition is cleared when you issue FILEDEF * CLEAR.

**<u>CHANGE</u>**
Combines definitions for an existing ddname with new ones when you issue 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.

**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.

**RECFM a**
Represents the record format of the file, where a can be one of the following:

| When a 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. |

**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.

**BLOCK nnnnn**
**BLKSIZE nnnnn**
Specifies the maximum block length in bytes. For fixed-length records (unblocked), this is the record length. For variable-length records, this gives the maximum logical record length (up to 32756 bytes, plus four 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 instruction.

## Option B

**DISP MOD**
Positions the read/write pointer after the last record in a disk file. Use this option only when you're 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 couldn't write to it. For standard label tapes, you can use this to add records to the end of the tape.

**DSORG PS**
Specifies that the data set has a physical, sequential (PS) organization.

See *VM/XA SP CMS Command Reference* for more information on using this command and its operands and options.

## Messages

```
CSIFLD001E Invalid option 'option'  RC=24
CSIFLD002E Invalid parameter 'parameter' in the option
           'option' field  RC=24
CSIFLD003E 'option' option specified twice  RC=24
CSIFLD004E 'option1' and 'option2' are conflicting
           options  RC=24
CSIFLD005S Virtual storage capacity exceeded
CSIFLD006E Invalid parameter 'parameter'  RC=24
CSIFLD011E Invalid character in fileid 'fn ft'  RC=20
CSIFLD017I Disk mode not accessed
CSIFLD021E Invalid mode 'mode'  RC=24
```

```
CSIFLD023E No filetype specified  RC=24
CSIFLD301E Invalid device 'device name'  RC=24
CSIFLD302E Parameter missing after DDNAME  RC=24
CSIFLD303I No user defined FILEDEFS in effect
CSIFLD304I Invalid CLEAR request
CSIFLD320E Error during FILEDEF CLEAR  processing, DCB(s)
           not closed
```

For more information on messages, see *VM/XA SP System Messages and Codes Reference*.

## GDUMP

### Produce a Copy of the Contents of Your Virtual Machine's Storage

Use the GDUMP command to produce a copy of your virtual machine's storage. Refer to the *VM/XA SP Group Control System Diagnosis Reference* to understand how to diagnose a GCS dump. Refer to *VM/XA SP Dump Viewing Facility Operation Guide and Reference* to learn how to use the dump viewing facility (DVF) to view a GCS dump.

The format of the GDUMP command is:

```
GDUMP  ┌         ┐ ┌ ┌   ┐ ┌         ┐┐ ┌           ┐        ┌           ┐
       │ hexloc1 │ │ │ - │ │ hexloc2 ││ │ TO *      │ [DSS]  │FORMAT type│
       │    0    │ │ │ : │ │ END     ││ │ TO userid │        │        GCS│
       └         ┘ │ └   ┘ └         ┘│ └           ┘        └           ┘
                   │.bytecount        │
                   └                  ┘
```

**hexloc1**
The hexadecimal address in virtual storage where the dump is to start. If no starting address is specified, 0 is assumed.

**- (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.

**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 4-kilobyte pages. These pages correspond to the 4-kilobyte pages into which storage is segmented. If you request that a certain portion of storage be dumped, the entire 4-kilobyte 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.

**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.

**.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
```

**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 userids (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 non-fetch-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 userid receiving the dump is not on the list of authorized userids (specified with the GROUP EXEC), fetch-protected data is omitted from the dump. However, all requested non-fetch-protected data and Key 14 storage is included.

Unauthorized userids 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 userids are prevented from using the CP TRANSFER command to transfer the dump-containing spool data file to their own machines.

If you don't 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 discontiguous shared segments, in your machine (the one where you're 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).

If you omit this operand, a format type of GCS is assumed.

## Messages

```
CSIDUM009E Operand is missing or invalid  RC=0C
CSIDUM010E Command Complete
CSIDUM525E Userid is missing or invalid  RC=14
CSIDUM526E Userid 'userid' not in CP directory  RC=10
CSIDUM527E Invalid range  RC=18
CSIDUM529E Partial dump taken  RC=4
CSIDUM531E Dump failed: spooling error  RC=08
CSIDUM532E Dump failed: I/O error  RC=1C
CSIDUR528I Dump complete
CSIDUR529E Partial dump taken
CSIDUR530E Dump failed
```

For more information on messages, see the *VM/XA SP System Messages and Codes Reference*.

The meanings of return codes for these messages are:

| Return Code | Meaning |
|---|---|
| 00 | The dump finished successfully. All requested areas were recorded in the dump. |
| 04 | Not all the requested areas were recorded in the dump. |
| 08 | A spooling error in CP prevents the dump from being recorded. |
| 0C | An operand is missing or invalid. |
| 10 | The recipient's userid is not in the CP directory. |
| 14 | The TO operand was specified but the userid was missing or invalid. |
| 18 | An invalid address range was specified. |
| 1C | CP experienced an I/O error when paging in the parameter list or dump list. No dump was recorded. |
| 20 | The input parameter list address was invalid. |

## Examples

In the following example, the first GDUMP command 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 discontiguous shared 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 0:CB8F7 TO * DSS
```

The second GDUMP command requests a dump of the issuer's virtual storage contents (excluding any discontiguous shared 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.

```
gdump
```

## GLOBAL

### 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 invoke 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 invoke a program.

The format of the GLOBAL command is:

| | |
|---|---|
| GLobal | LOADLIB [libname1... libname63] |

**LOADLIB**
An operand indicating that you are referring to CMS load libraries.

**libname1...**
The filenames 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.

### Example

To find out what load libraries are currently identified to be searched, type:

```
query loadlib
```

### Messages

```
CSIGLB005S Virtual storage capacity exceeded
CSIGLB013E No function specified  RC=24
CSIGLB014E Invalid function 'function'  RC=24
CSIGLB024E File 'fn ft fm' not found  RC=28
CSIGLB220E Unable to open file 'filename'  RC=28
CSIGLB221S More than nnn libraries specified  RC=88
CSIGLB222E File 'fn ft fm' contains invalid record formats
           RC=32
CSIGLB223S Error 'xx' reading file 'fn [ft fm]' from disk
           RC=100
```

For more information on messages, see the *VM/XA SP System Messages and Codes Reference*.

## HX

### 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've 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.

The format of the HX command is:

| | |
|---|---|
| HX | |

### Messages

```
CSIABD225I Hx complete
```

For more information on messages, see the *VM/XA SP System Messages and Codes Reference*.

## ITRACE

### Enable or Disable Recording of Internal Trace Events for a Virtual Machine or an Entire Group

GCS maintains an internal trace table in common storage. This table contains two types of information:

- records of supervisor events and
- records of 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 via 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, you have to enable the tracing yourself. You can issue the ITRACE command to turn off tracing of supervisor events and later to turn it back on. Likewise, 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.

The format of the ITRACE command is:

```
ITrace   { [ [ GTrace ]            ] }
         { [ [ SUP    ]   [     ]  ] }
         { [              [ OFF ]  ] }      [ GRoup ]
         { [ [ ALL ]      [     ]  ] }
         { [END ]                    }
```

**GTrace**
: Indicates that you want to affect only the internal tracing of data passed via the GTRACE macro.

**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.

**ALL**
: Indicates that you want to apply this command to the internal tracing of both GTRACE and supervisor 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.

**GRoup**
Indicates that you want this command to apply to the whole 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 userid. Commands you issue with the GROUP option "on" normally will take precedence over commands issued without. However, an authorized virtual machine can disable tracing for itself even though another authorized virtual machine started internal tracing for the whole group.

## Messages

```
CSIYTG001E Invalid option 'option'  RC=4
CSIYTG009E Operand missing or invalid
CSIYTG517I ITRACE set ON for event-type(s)
CSIYTG518I ITRACE set ON for event-type(s) for GROUP
CSIYTG519I ITRACE set OFF for event-type(s)
CSIYTG520I ITRACE set OFF for event-type(s) for GROUP
CSIYTG521E ITRACE GROUP option is in effect for event-type(s)
           RC=8
```

For more information on messages, see the *VM/XA SP System Messages and Codes Reference*.

The meanings of return codes for these messages are:

| Return Code | Meaning |
|---|---|
| 00 | The tracing of events has been successfully enabled or disabled. |
| 04 | An invalid operand was specified, or an unauthorized user specified the GROUP operand. Your request was ignored. |
| 08 | 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. |

## Examples

To enable tracing of GTRACE events (program or application data) in the virtual machine issuing the ITRACE command, enter:

```
ITRACE GTRACE
```

To enable tracing of GTRACE events for the virtual machine group, enter:

```
ITRACE GTRACE GROUP
```

The virtual machine issuing this command must be authorized.

To disable tracing of GTRACE events for the virtual machine group, enter:

```
ITRACE GTRACE OFF GROUP
```

The virtual machine issuing this command must be authorized. To disable internal tracing of supervisor events for the virtual machine issuing this command, enter:

```
ITRACE SUP OFF
```

To disable internal tracing of all events for the virtual machine, issue this command:

```
ITRACE END
```

If tracing had been enabled for the whole group, you would need an authorized virtual machine to issue this for yourself. If the tracing was enabled just for your virtual machine, you don't have to be authorized to issue this for yourself.

## LOADCMD

### Define a Program Module to be Executed as a Command

LOADCMD is a feature that lets you define your own commands. More precisely, 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've defined with a GLOBAL command.) When you issue the command name, this module gets control and executes. It remains in storage, waiting to be run again when you issue 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've defined and issued this "VTAM" command name, you can enter any of the following ACF/VTAM commands:

- START
- HALT
- VARY
- MODIFY
- DISPLAY.

The format for the LOADCMD command is:

| | |
|---|---|
| `LOADCmd` | `name member` |

**name**
The name of the command you are defining.

**member**
The member of a CMS load library associated with the command you've defined. This member is the module that executes the command you've defined; it is loaded into private, free storage.

When you enter the `name`, GCS calls the `member` to execute the command. Here's what your registers will contain:

| Register | Contents |
|---|---|
| 0 | Address of an extended parameter list (plist). |
| 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's 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. |

| Register | Contents |
|---|---|
| 14 | Return address of the SVC handling routines. The program returns control to this address after it finishes executing. |
| 15 | 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'll see it in the ready message (if you entered the command at the console): `Ready(nnnnn);` If the program you execute 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). |

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 the System Product Interpreter when ADDRESS COMMAND is specified. X'05' is used by the System Product Interpreter 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.

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  DC   *
ARGEND  EQU  *
```

With no arguments specified, ARGBEG is set equal to ARGEND.

For more information on parameter lists, see the *VM/XA SP CMS User's Guide.*

## Messages

```
CSILDC212E  Member cannot be loaded.  Command is not defined
            RC=xx
CSILDC240I  No entry points were loaded by the LOADCMD command
```

For more information on messages, see the *VM/XA SP System Messages and Codes Reference.*

The meanings of return codes for this message are:

| Return Code | Meaning |
|---|---|
| 01 | The command has already been defined. |
| 04 | 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. |
| 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 restructured so that it does not require overlays. |
| 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. |

| Return Code | Meaning |
|---|---|
| 14 | The command name specified is a GCS immediate command or an abbreviation for one. |
| 24 | Too many operands were specified. |
| 28 | The specified member cannot be found. |
| 32 | No member name was specified. |
| 36 | A permanent I/O error was found when the system attempted to search the CMS LOADLIB directory. |
| 40 | Insufficient virtual storage was available to read the directory entry for this module. |
| 41 | Insufficient free storage was available to build the nucleus extension control blocks representing this command. |

## Example

To define the command named MYCMD to GCS, enter the following command:

```
LOADCMD MYCMD MYMOD
```

The module containing the code for this command can be found in a CMS load library under the member name of MYMOD.

The MYMOD module can be invoked by entering:

```
MYCMD
```

# OSRUN

## Start a GCS Application Program

Use the OSRUN command to execute a GCS application. The application program must either be a member of a CMS load library (defined with the GLOBAL command) or else reside in a saved segment. The OSRUN command maintains control until the program ends; therefore, you cannot execute other commands while the program is running.

The format of the OSRUN command is:

| | |
|---|---|
| OSRUN | member [PARM=parameters] |

**member**
The member of the CMS load library you want to execute.

**PARM = parameters**
The OS parameters that you want to pass to the module. If these parameters contain blanks or special characters, you must enclose them in quotes. To include a quote mark in a parameter, enter two quotes marks side-by-side (' '). Parameters must be no longer than 100 characters. They get 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

```
CSILOS220E Unable to open file 'filename'
CSILOS223S Error 'nn' reading file 'fn [ft fm]' from disk
CSILOS224E Member 'membername' not found in library
CSIOSR006E Invalid parameter 'parameter'  RC=24
CSIOSR022E No filename specified  RC=24
CSIOSR219E Parm field contains more than 100 characters  RC=24
CSIOSR236E Ending apostrophe is missing  RC=24
CSIABD237E Command ended without detaching subtasks
```

For more information on messages, see the *VM/XA SP System Messages and Codes Reference.*

## QUERY

### Request Information About Your GCS Virtual Machine

Use the QUERY command to gather information about your GCS virtual machine.

The format of the QUERY command is:

```
Query     { DISK [mode    ]  }
          {      [*       ]  }
          {      [R/W     ]  }
          {      [MAX     ]  }
          { FILEDEF          }
          { LOADLIB          }
          { SEARCH           }
          { SYSNAMES         }
          { DLBL   [mult]    }
          { ETRACE           }
          { ITRACE           }
          { GROUP            }
          { LOCK             }
          { REPLY            }
          { LOADCMD          }
          { LOADALL          }
```

The DISK, LOADLIB, FILEDEF, and SEARCH operands work the same as for the CMS QUERY command with the exception that no options are allowed.

**DISK**
Displays the following disk information:

```
LABEL VDEV M STAT CYL TYPE BLKSIZE FILES BLKS USED-(%) BLKS LEFT BLKTOTAL
label vdev m stat cyl type blksize nnnn      nnnn-nn       nnnn    nnnnn
```

**label**
The label assigned to the disk when it was formatted. If it's an OS or DOS disk, this is the volume label.

**vdev**
The virtual device address.

**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 FBA 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.

**nnnn 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.

**nn %**
The percentage of blocks in use.

**nnnn BLKS LEFT**
The number of disk blocks left. (The actual number of disk blocks remaining is lower because this number also counts control blocks.)

**nnnnn BLK TOTAL**
The total number of blocks

If you specify `DISK mode`, you see information about the single disk associated with that mode. `DISK *` gives you one line of information for each disk that's accessed. `DISK R/W` gives information for each accessed disk in read/write mode, and `DISK MAX` gives information for the R/W disk having the most available space.

**FILEDEF**
Displays all file definitions in effect.

*Response:*

```
ddname device [fn [ft]]
  .      .      .  .
  .      .      .  .
  .      .      .  .
```

If you have no file definitions in effect, you'll receive the following message:

```
No user defined FILEDEFS in effect
```

**LOADLIB**
Displays the names of all files (of filetype 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:*

```
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
```

**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 it's an OS or DOS disk, this is the volume label.

**vdev**
The virtual device address.

**m**
The filemode letter assigned to the disk when it was accessed.

**R/O**
**R/W**
Indicates whether the disk is read/only or read/write.

**OS**
**DOS**
Indicates an OS or DOS disk.

**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 via the SET command.

**DLBL**
Querying DLBL yields the following information:

```
DDNAME  MODE TYPE CATALOG VOL BUFSPC PERM DISK  DATASET.NAME
xxxxxx   nn  xxxx xxxxxxx            xxx  xxx   xxxxxxx
  .       .    .     .     .    .     .    .       .
  .       .    .     .     .    .     .    .       .
  .       .    .     .     .    .     .    .       .
```

**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 filename and filetype 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'll get the following message:

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

**ETRACE**
Displays a list of the events that are enabled for external tracing (recording in a spool data file).

***Response:***

```
All external trace events are disabled
External trace is enabled for event-type(s)
External trace is enabled for event-type(s) for GROUP
```

**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
```

**GROUP**
Displays the userids of the virtual machines in the GCS group of the issuer.

***Response:***

```
GROUPID=groupname, USERS: CURRENT=ccccc, MAXIMUM=mmmmm
  VMUSERID(s)
```

**LOCK**
Displays the status of the common lock. If the lock is held, the userid holding the lock is displayed.

***Response:***

```
The common lock is free
The common lock is held by 'userid'
```

**REPLY**
Displays the text and the identification number of all messages waiting for a reply.

***Response:***

```
No replies outstanding
The following replies are outstanding:
  xx  yyyyyyyy
  xx  yyyyyyyy
```

**LOADCMD**
Locates the entry point addresses for all entry points that are loaded by the LOADCMD command.

**LOADALL**
Displays the entry point names and addresses for the entry points that have been loaded and currently reside in the virtual machine storage.

For more information on using the QUERY command, see the *VM/XA SP CMS Command Reference*.

| Return Code | Meaning |
|---|---|
| 00 | Successful completion. The requested information is displayed. |
| 04 | Invalid option specified. No action taken. |

## Messages

All QUERY messages except CSIQRL032S are issued without message numbers.

```
CSIQRD239I "No entry points are currently loaded in this
           virtual machine"
CSIQRD240I "No entry points were loaded by the LOADCMD command"
CSIQRL032S Supervisor error 5.  Re-IPL sysname
CSIQRL217E The common lock is free
CSIQRL218I The common lock is held by userid
CSIQRQ514I All external trace events are disabled
CSIQRQ515I External trace is enabled for event-type(s)
CSIQRQ516I External trace is enabled for event-type(s)
           for GROUP
CSIQRQ522I Internal trace is enabled for event-type(s)
CSIQRQ523I Internal trace is enabled for event-type(s)
           for GROUP
CSIQRQ524I All internal trace events are disabled
CSIQRR005S Virtual storage capacity exceeded  RC=8
CSIQRR214I No replies outstanding
```

```
CSIQRR215I The following replies are outstanding:
CSIQRR216I GROUPID=systemname, USERS: CURRENT=nnnnn,
           MAXIMUM=mmmmm
CSIQRS015E 'parameter' is invalid for 'function' function
           RC=24
CSIQRS017E Disk {mode|vdev|volumeid} not accessed
CSIQRS019E No Read/Write mode disk accessed  RC=1
CSIQRS020E No Read/Write disk with space available accessed
           RC=2
CSIQRU303I No user defined FILEDEFs in effect
CSIQRX005S Virtual storage capacity exceeded  RC=8
CSIQRX006E Invalid parameter 'parameter'  RC=24
CSIQRX303I No user defined DLBLs in effect
CSIQRY005S Virtual storage capacity exceeded  RC=8
CSIQRY006E Invalid parameter 'parameter'  RC=24
CSIQRY013E No function specified  RC=24
```

For more information on messages, see the *VM/XA SP System Messages and Codes Reference*.

## RELEASE

### Release a Disk

After an application no longer needs files on a particular disk, you should issue the RELEASE command for that disk.

The format of the RELEASE command is:

| | | |
|---|---|---|
| RELease | vdev<br>mode | [(DET[)]] |

**vdev**
The virtual device address of the disk to be released.

**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 *VM/XA SP CMS Command Reference.*

### Messages

```
CSIARE006E Invalid parameter 'parameter'
CSIARE017E DISK {mode|vdev|volumeid} not accessed
CSIARE021E Invalid mode 'mode'
CSIARE415E Invalid device address 'vdev'
CSIARE416E No device specified
```

For more information on messages, see the *VM/XA SP System Messages and Codes Reference.*

# REPLY

## 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 via the WTOR macro.

The format of the REPLY command is:

| | |
|---|---|
| Reply | id [text] |

**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:** The WTOR macro instruction 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.

A list of all messages awaiting reply, along with their identification numbers, can be obtained by issuing:

```
query reply
```

| Return Code | Meaning |
|---|---|
| 0 | Your reply is accepted. |
| 4 | No message requiring a reply is associated with the identification number you specified. |
| 8 | Your reply was not accepted. Its format was invalid. |
| 10 | The reply buffer address or ECB address was not accessible. |

## Example

In the following example, the operator informs the issuer of a WTOR, whose identification number is 16, that a disk has been mounted at address 250.

```
reply 16 disk is mounted at address 250
```

## Messages

```
CSIRPY206E Reply not accepted, ID not specified
CSIRPY207E Reply not accepted, ID number not 00 to 99  RC=8
CSIRPY208I Reply xx not outstanding  RC=4
CSIRPY209E Reply xx not accepted, reply too long for requestor
           RC=8
CSIRPY210E Reply not accepted, invalid ECB address  RC=10
CSIRPY211E Reply not accepted, invalid reply buffer address
           RC=10
```

For more information on messages, see the *VM/XA SP System Messages and Codes Reference.*

## SET

### Replace a Saved System Name Entry in the SYSNAMES Table for VSAM

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. Once you initialize VSAM, these saved system names cannot be changed.

The format of the SET command is:

| | |
|---|---|
| SET | SYSNAME $\left\{\begin{array}{l}\text{GCSVSAM}\\\text{GCSBAM}\end{array}\right\}$ entry name |

**SYSNAME**
Specifies that a saved system name in the SYSNAMES table is to be replaced.

**GCSVSAM**
Indicates that the entry name you're 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 the *VM/XA SP Installation and Service.*

**GCSBAM**
Indicates that the entry name you're 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 the *VM/XA SP Installation and Service*

**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.

To display the saved system names currently available to your virtual machine, enter:

```
query sysnames
```

## Messages

```
CSISET006E Invalid parameter 'parameter'  RC=24
CSISET013E No function specified  RC=24
CSISET321E Saved system name 'name' is invalid.  Only GCSVSAM
           or GCSBAM  allowed  RC=24
CSISET322E New system name missing after 'name'  RC=24
CSISET323E Parameter missing after SYSNAME  RC=24
CSISET351E System name not changed.  VSAM already initialized.
           RC=24
```

For more information on messages, see the *VM/XA SP System Messages and Codes Reference.*

# Chapter 3. GCS Macro Overview

This section provides an overview of how GCS works.

## Multitasking

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, and, for the most part, 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, even though the CPU 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.

### 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 9 on page 65. All those tasks belong to one GCS application program. They vie with each other for an opportunity to execute in that application' 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. |

Refer to the ATTACH and DETACH commands for more detailed information.

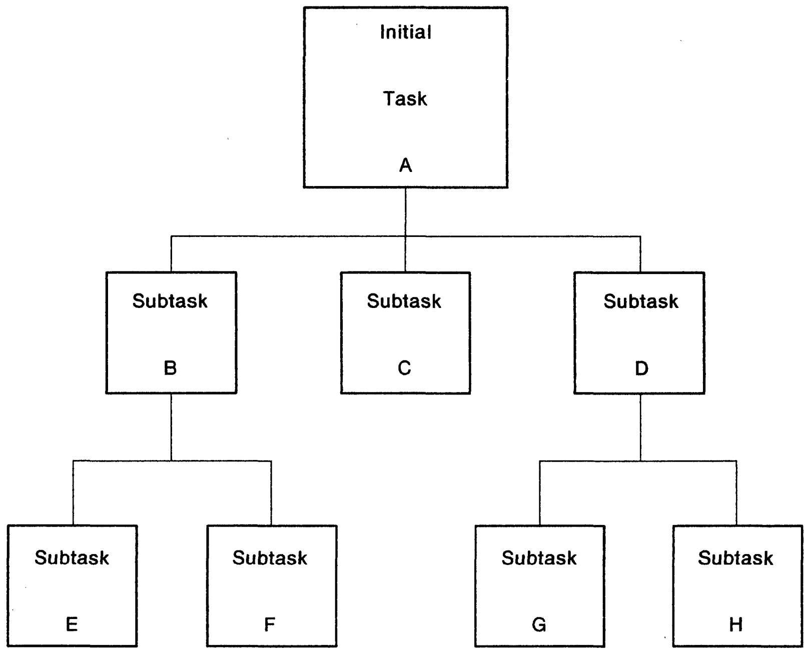


Figure 9. Diagram of a Task's Family Tree. Parent task A adds subtasks B, C, and D. Subtask B becomes the parent of subtasks E and F. Subtask C has no offspring. Subtask D becomes the parent of subtasks G and H.

## 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 execute first identify themselves with the task ID. 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 invoke the CHAP macro. CHAP works only for:

- A task that wants to change its own priority
- A parent task that wants to change the priority of one of its attached subtasks.

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 keeps timing information about the running task. If the running task exceeds the time limits the task dispatcher switches 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, or

- Until a higher priority task becomes ready to run, or
- Until it terminates, or
- Until it issues a WAIT.

## Terminating Tasks

Task termination has two facets:

1. What makes tasks terminate:

   **NORMALLY:** A task ends normally for one reason.

   A task 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.

     **Note:** 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 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 or,
   - Continue termination.

   During the exit, an abended task can ask the GCS supervisor to let it recover control and continue executing. GCS will invoke this ESTAE exit for any abend, unless certain circumstances prevail. Refer to "ESTAE" on page 108 for more information.

- 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 via ENQ
  - 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 via 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).

## Coordinating Dependent Tasks

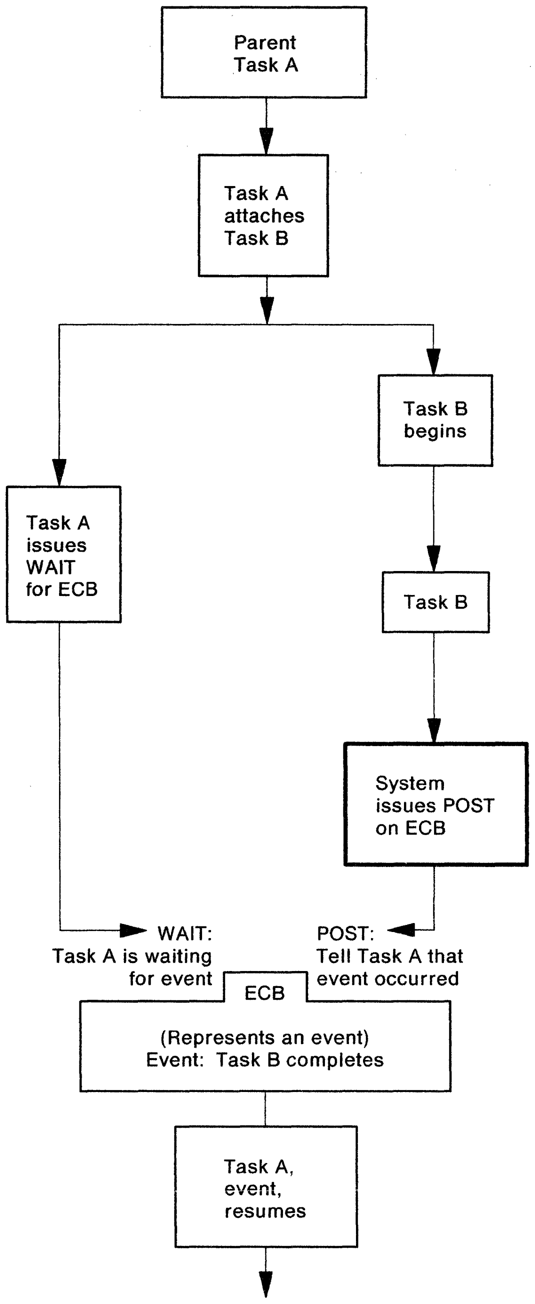
Often, tasks depend on each other to get work done. For instance, one task might have to stop running until a second task provides additional information or service. When that "event" occurs, and the first task resumes again, the two tasks have synchronized.

"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, and |
| **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. Refer to Figure 10 on page 68.



Parent Task A
Task A attaches Task B
Task B begins
Task A issues WAIT for ECB
Task B
System issues POST on ECB
WAIT: Task A is waiting for event
POST: Tell Task A that event occurred
ECB
(Represents an event) Event: Task B completes
Task A, event, resumes

Figure 10. How Tasks Can Use WAIT and POST Macros

GCS's specific function for VM/XA SP WAIT and POST work only among tasks in the same virtual machine.

## 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, identifying 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.

## OS Management Services

The OS *management* services (storage management, program management, and timer management) described in this section are GCS services that resemble (but do not duplicate) MVS 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 can't 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, and
- 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 sixteen possible storage keys for different types of code. A storage area's key depends upon what type of code it contains:

| Key | Type of Code |
|---|---|
| 0 | Saved segments and reentrant code (including GCS common storage and other shared code) |
| 1-13 | Authorized (privileged) applications |
| 14 | Unauthorized (nonprivileged) applications (also the starting key for authorized applications) |
| 15 | 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 allocating storage in the new key with the GETMAIN macro.

1. Specifying a new PSW key with the SPKA instruction, and
2. Allocating storage in the new key with the GETMAIN macro.

## Program Management

Programs running on GCS can load and execute 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 find the module in one of your system's saved segments. (In either case, the supervisor will use a copy where it locates one.) Failing to find it in a saved segment,[3] 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 11 on page 71.

[3] Each saved segment has a directory that was created with the CONTENTS macro. The GCS supervisor searches these directories when looking for a particular module.

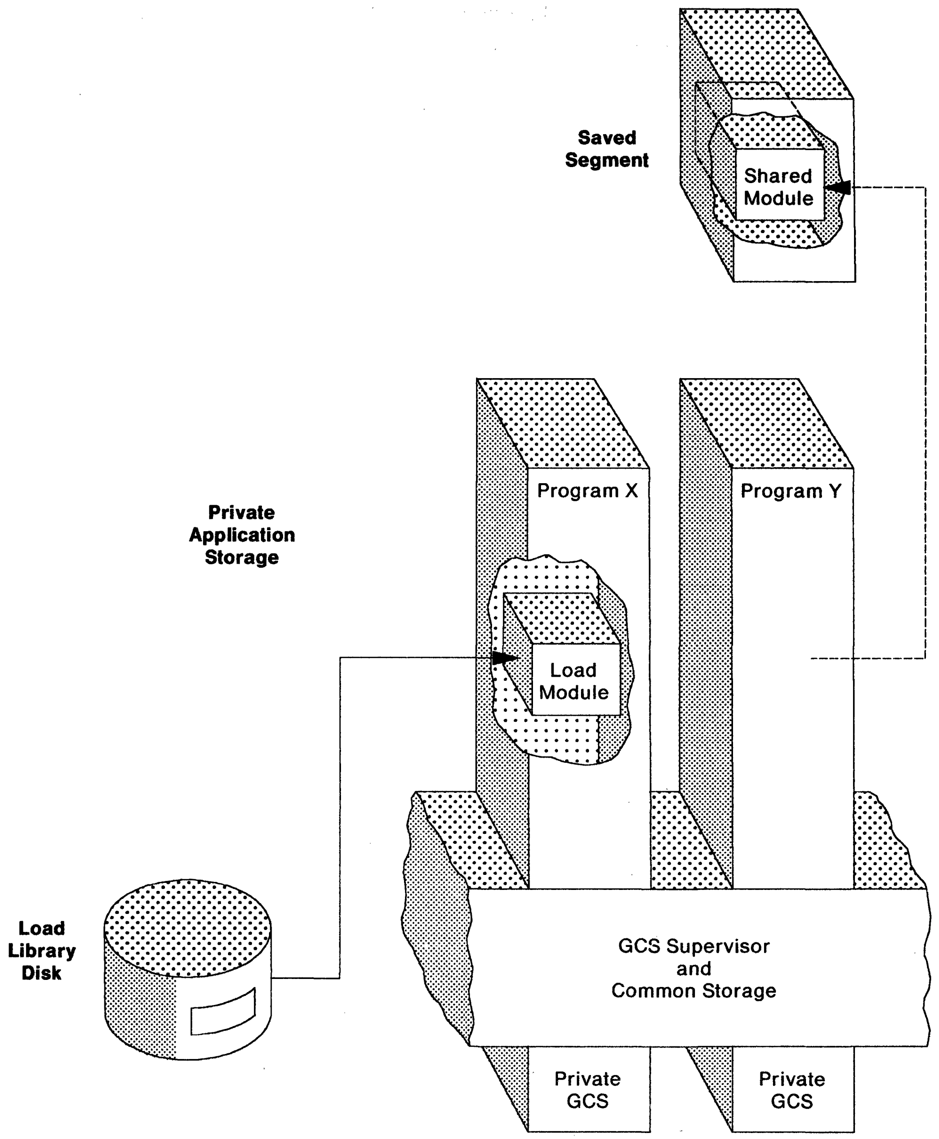


Figure 11. Obtaining Modules Requested by a GCS Program. Program X, on the left, loads a copy of a module from a disk load library. Program Y, on the right, shares a reentrant module in a saved segment, using it where it exists without actually copying it.

To load a module, a program can issue any of the following macros in Table 1:

Table 1. 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. In addition, 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, 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. |
| 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 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. |

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 transfers control 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 a time 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. |

For more information and a detailed explanation of each macro, see the STIMER, TTIMER, and TIME entries.

## Native GCS Services

The native GCS services described in this section make use of unique GCS functions. *Authorization* provides the basis for service. 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. |

The table belows identifies what AUTHCALL can and cannot do:

Table 2. 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 execute its own code in supervisor state or Key 0. |

"AUTHCALL" on page 172 describes the AUTHCALL macro in more detail.

## 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, or
- CP.

Task-users rely on two macros for IUCV 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 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 shown in Table 3 on page 75.

Table 3. A Nonpriviledged IUCV Interface

| 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 | |

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 paths
   - Can issue the following functions directly (without going through the IUCVCOM macro):

| | |
|---|---|
| IUCV PURGE | IUCV REJECT |
| IUCV QUERY | IUCV REPLY |
| IUCV QUIESCE | IUCV RESUME |
| IUCV RECEIVE | IUCV SEND |

## 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 functions that an unauthorized application can use to execute virtual channel programs on any real or virtual I/O device except DASD. Table 4 on page 76 lists the six different functions provided by GENIO:

Table 4. GENIO Supported Functions for Unauthorized Programs

| **Function** | **Description** |
|---|---|
| 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. 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 execute real channel programs on real devices as shown in Table 5:

Table 5. GENIO Supported Function for Authorized Programs

| **Function** | **Description** |
|---|---|
| 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 execute directly on the real channel, without CP first translating them. Before you can execute real channel programs, you need an authorized user ID and a special entry in your VM/XA SP directory. You make this entry by specifying the DIAG98 parameter on the OPTION directory control statement.

To execute real I/O, authorized programs use GENIO's STARTR (start real) function. For the most part, 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 SIOF instruction.

Refer to "GENIO" on page 181 for more detailed information about GENIO and its parameters.

## Securing Pages of Storage

An authorized program intending to perform real I/O using STARTR must first build a channel program[4]. 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 via the PGLOCK macro. |

Refer to "PGLOCK" on page 204 and "PGULOCK" on page 206 for more detailed information.

## 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 thereby 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's *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. 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's already *on*, the GCS supervisor will suspend the requesting program until the lock gets turned off. As soon as it's off, LOCKWD informs the waiting machine that the common lock is available. This serializes (or synchronizes) group use of common storage.

Refer to "LOCKWD" on page 196 for more detailed information.

[4] For information about building channel programs, see the chapter titled *Input/Output Operations* in the *System/370 Principles of Operation.*

## 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,[5] 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.

Refer to "VALIDATE" on page 216 for more detailed information.

## 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 (providing that it is not disabled), the exit routine gets control in supervisor state and with a PSW key of 0 (zero).

After scheduling the exit, the authorized program continues executing. And after the exit routine finishes, GCS lets the interrupted task continue executing. Refer to "SCHEDEX" on page 208 for more detailed information on the SCHEDEX macro.

## 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 execute in the group's recovery machine.

  **Note:** The recovery machine must be the first one to join your group. It has responsibility for cleaning up system resources when other machines using them reset. (See "What Makes a Machine Reset ?" on page 79.) This clean-up involves executing all currently existing exit routines set up with the MACHEXIT macro.

  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.

- 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 (refer to "Native GCS Services" on page 73), it resembles

[5] The authorized program's key does not need to match that of 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.

the act of identifying an exit routine's address. For details on transferring control to authorized entry points, see the AUTHNAME and AUTHCALL entries.

### What Makes a Machine Reset ?

- Logging off
- IPLing another system (or re-IPLing GCS)
- A machine check (under certain conditions)
- Issuing certain CP commands:

  SYSTEM RESET
  SYSTEM CLEAR
  DEFINE STORAGE

## Data Management Services

GCS applications can process CMS files on minidisks, VSAM files, and CP spool data files. With GCS's data management services, applications can perform input, output, or update operations on a file, depending on whether it's a CMS, VSAM, or CP spool data file. The types of data management services include:

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

### Processing CMS Minidisk Files

A GCS program processes CMS 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 data 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 let you use any of the following:

- OS formatted files
- 800-byte block size disk format
- 2314 series disks.

Nor does it allow:

- File mode 4 (treated instead like file mode 1)
- Spanned records
- Console or tape I/O
- Utility functions (such as formatting disks, loading files from tape, editing files).

However, 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 detailed information about disk sharing, see the *VM/XA SP 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 and the requirements that go along with each.

   The requirements for programming procedures are described in Table 6:

Table 6. 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. |
| 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. |
| 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 issue the FILEDEF command only once for each ddname. If you need to issue FILEDEF for the same ddname and same file later in the program, make sure you specify the NOCHANGE option Refer to "FILEDEF" on page 34 for the FILEDEF command format.

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

   When tasks share a single DCB, GCS permits three types of processing:

   a. inputting

   b. outputting

   c. updating.

   To coordinate their activities, tasks must use the ENQ and DEQ macros. Only one of the macros 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). Refer to "Coordinating Shared Resources" on page 68 for more information on shared resources.

## Processing CP Spool Data Files

BSAM and QSAM functions allow GCS programs to process virtual reader, printer, and punch files. Existing CP facilities, such as CP Directory, DEFINE, DETACH, SPOOL, and TAG define and manipulate the various unit record devices.

**Note:** GCS programs cannot write to virtual readers or 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'll 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 wait 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:** 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 be used to get or modify data in these control blocks.

Some other VSAM information you should consider is:

- VSAM data management services support the CHECK macro and RPL's "ASY" option, but no asynchronous activity is performed.
- GCS does not support utility functions. You have to perform disk initialization, catalog definition, and file definition (AMS functions) under CMS.
- 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 (for input or output) determines how VSE/VSAM handles shared data. See the *VSE/VSAM Programmer's Reference* for more information.
- When a task terminates, GCS tries to close all open ACBs that the task opened.

# Chapter 4. Task Management Service Macros

## ABEND

### Abnormally Terminate the Active Task

For a variety of reasons, a task running under GCS may decide that it should abnormally terminate itself.

Use the ABEND macro instruction to effect this.

The format of the ABEND macro instruction is:

| | | |
|---|---|---|
| [label] | ABEND | completion code[,DUMP][,STEP] ,USER<br>,SYSTEM |

### 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 instruction that created the ABENDing task, then the completion code is placed there. (If necessary, review the entry titled "ATTACH" on page 86.) 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, described below, 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**
Specifies that a dump will be sent to your virtual reader.

GCS sends 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 non-fetch-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 Notes

- 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.
- When a task terminates, the GCS supervisor performs normal 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 via the ESTAE macro instruction. (If necessary, review the entry titled "ESTAE" on page 108.) The exit routine may attempt to retry the failed function or request that the supervisor continue with normal task termination.
- 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 instruction 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 instruction, then GCS posts the appropriate event control block.

  If necessary, review the entry titled "ATTACH" on page 86.
- 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.

## Example

In the following example, the active task terminates itself abnormally.

```
ENDIT ABEND 899,DUMP
```

The user completion code of 899 describes the reason for the abend. The task requests that a dump of its virtual storage be produced to aid in diagnosing the problem. ENDIT is the label on this instruction.

## Return Codes and Abend Codes

The ABEND macro generates no return codes.

| Abend Code | Meaning |
|---|---|
| 20D | A descendant subtask of this task issued the ABEND instruction with the STEP parameter specified. This task was abnormally terminated. |

## ATTACH

### 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. Moreover, the subtask's code must be brought into virtual storage if it is not already there.

The ATTACH macro instruction 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 instruction thereby becomes the immediate ancestor of the subtask in question.

The ATTACH macro instruction is available in standard, list, and execute format.

The ***standard*** format of the ATTACH macro instruction is:

```
[label]    ATTACH    (EP=symbol      ) [,PARAM=(addresses)[,VL=1]][,ECB=address]
                     |               |
                     |EPLOC=address  | [,ETXR=address][,DPMOD=number] [,SZERO= {YES}]
                     {               }                                [        {NO }]
                     |               |
                     |DE=address     | [,SHSPV=number ][,SM= {PROB}][,JSTCB= {YES}]
                     (               ) [,SHSPL=address][     {SUPV}][        {NO }]
```

### Parameters

**EP**

Specifies the eight-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:

- The name of the entry point as previously defined via the IDENTIFY macro instruction. If necessary, review the entry titled "IDENTIFY" on page 135.
- The name of the entry point declared in a shared segment directory via the CONTENTS macro instruction. If necessary, review the entry titled "CONTENTS" on page 427.
- 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, since the module associated with the entry point name may already be loaded.
2. Any shared segment directories that may have been created via the CONTENTS macro instruction.
3. The directories of any load libraries that may have been defined for your virtual machine via the GLOBAL LOADLIB command. For more information on the GLOBAL command see "GCS Commands" on page 20.

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 eight-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 instruction. If necessary, review the entry titled "BLDL" on page 126.

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 once it receives control.

GCS builds a parameter list containing these addresses in the order in 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.

The entries titled "WAIT" on page 122 and "POST" on page 116 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 as soon as the latter terminates.

Remember, if you specify the address of an ECB in the ATTACH instruction, then you must explicitly issue the DETACH instruction when you are finished with the subtask in question. The DETACH instruction releases all the storage associated with your subtask, including its control blocks. If necessary, review the entry titled "DETACH" on page 100.

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. Moreover, if your exit routine is to be shared by several subtasks, then it must be reentrant.

Remember, if you specify the address of an exit routine in the ATTACH instruction, then you must explicitly issue the DETACH macro instruction when you are finished with the subtask in question. Normally the DETACH instruction 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. Likewise, 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. Likewise, if this sum is greater than 250, then the dispatching priority of your subtask will be 250.

**Note:** If the task issuing the ATTACH instruction 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 instruction for storage in subpool 0, then GCS automatically frees the storage when the task terminates. Likewise, 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. Hence, 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).

If a main task issues the GETMAIN instruction for storage from subpools 1 through 127, then GCS automatically frees the storage when the task terminates. Likewise, 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 instruction, 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.

Since 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. In addition, the first byte in the list must contain the number of bytes remaining in the list. Each subsequent byte 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 in which your new subtask will run. This parameter is valid only if the task issuing the ATTACH instruction 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 via the DETACH instruction when it is no longer needed.

NO
: Indicates that your subtask is not an independent application. This is the case, by default.

## Usage Notes

- The ATTACH macro does not transfer control to your new subtask. It merely sets up a task block for your subtask based upon the information you provide in the ATTACH instruction.

  When the new subtask is dispatched the first time, it receives control. At this point, the programs it contains are enabled for interrupts. Moreover, the subtask runs in the same key in which its ancestor task ran when the latter issued the ATTACH instruction.

- 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 two low-order bytes of register 1. Further, the two high-order bytes of this register will contain the appropriate virtual machine id.

  This task id is used to refer to your new subtask if you decide to delete it from the system or change its dispatching priority. If necessary, review the entries titled "DETACH" on page 100 and "CHAP" on page 94.

  **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 instructions.

- Do not use the ATTACH macro instruction in an ESTAE exit routine.

- An end-of-task exit routine will always run in the same key and state as the task that issued the ATTACH instruction originally.

- If neither the ECB nor ETXR parameter is specified, then the subtask is automatically removed from the system as soon as it terminates.

- When an exit routine specified in an ATTACH instruction receives control, the contents of the registers are:

| Register | Contents |
|---|---|
| Register 0 | Unpredictable. |
| Register 1 | The task id for the subtask that just terminated. |
| Registers 2 - 12 | Unpredictable. |
| Register 13 | The address of an eighteen-word register save area provided by the GCS supervisor. |
| Register 14 | The return address within the GCS supervisor. |
| Register 15 | The address of the exit routine. |

- When the new subtask receives control, the contents of the registers are:

| Register | Contents |
|---|---|
| Register 0 | Unpredictable. |
| Registers 1 - 12 | Propagated to the new subtask. |
| Register 13 | The address of a new user save area. |
| Register 14 | The return address within the ancestor task. |
| Register 15 | The address of the entry point. |

- If the program that receives control once 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.

## Example

To have a task request that a new subtask be created, enter:

```
ATTACH EPLOC=(4),PARAM=((5),(6),(7)),VL=1,ECB=MYECB,SHSPL=SPLIST
```

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. Since 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.

| Return Code | Meaning |
|---|---|
| 00 | Function successfully completed. |
| 04 | An ATTACH macro instruction 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. |

## The List Format

```
[label]   ATTACH   ┌               ┐
                   │EP=symbol      │ [,PARAM=(addresses)[,VL=1]][,ECB=address]
                   │               │
                   │EPLOC=address  │ [,ETXR=address][,DPMOD=number] ┌,SZERO= ⎧YES⎫┐
                   │               │                                │        ⎨   ⎬│
                   │               │                                │        ⎩NO ⎭│
                   │               │                                └             ┘
                   │DE=address     │ ┌,SHSPV=number ┐┌,SM= ⎧PROB⎫┐┌,JSTCB= ⎧YES⎫┐
                   └               ┘ │              ││     ⎨    ⎬││        ⎨   ⎬│
                                     │,SHSPL=address││     ⎩SUPV⎭││        ⎩NO ⎭│
                                     └              ┘└          ┘└             ┘
                   ,SF=L
```

This format of the macro instruction 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 instruction.

## The Execute Format

```
[label]   ATTACH   ┌              ┐
                   │EP=symbol     │ [,PARAM=(addresses)[,VL=1]][,ECB=address]
                   │              │
                   │EPLOC=address │ [,ETXR=address][,DPMOD=number] ┌,SZERO= ( YES ) ┐
                   │              │                                │        (  NO ) │
                   │              │                                └                ┘
                   │DE=address    │ ┌               ┐┌             ┐┌              ┐
                   └              ┘ │,SHSPV=number  ││,SM= ( PROB )││,JSTCB= ( YES )│
                                    │,SHSPL=address ││     ( SUPV )││        ( NO  )│
                                    └               ┘└             ┘└              ┘
                   ┌                                ┐
                   │,MF=(E,address)                 │
                   │,SF=(E,address)                 │
                   │,MF=(E,address),SF=(E,address)  │
                   └                                ┘
```

This format of the macro instruction generates code that executes the function, using a parameter list whose address you specify.

### Added Parameter

**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 to be used by the macro that you generated using the list format of the instruction.

You can add or modify values in this parameter list by specifying them in this instruction.

# CHAP

## Change the Dispatching Priority of a Given Task

GCS is a multitasking system. This means that considerably more than one task can execute 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 in which 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 instruction to change the dispatching priority of any application task within your virtual machine.

The format of the CHAP macro instruction is:

| | | |
|---|---|---|
| [label] | CHAP | priority change value [,task id address<br>,'S'] |

## 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 instruction for it. (If necessary, review the entry titled "ATTACH" on page 86.) 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 two low-order bytes at this address. GCS ignores the two 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 instruction 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 instruction is the one to be changed.

If you omit both the S and the TASK ID ADDRESS parameters, then GCS treats the instruction as though the S parameter were specified.

Note that this parameter must be surrounded by single quotation marks.

## Usage Notes

- No task can change the dispatching priority of any other task unless the former issued the ATTACH macro instruction 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.
- You cannot use the CHAP instruction to change the priority of the system task.

## Return Codes and Abend Codes

The CHAP macro generates no return codes.

| Abend Code | Meaning |
|---|---|
| 12C | The task ID specified was invalid for one of the following reasons:<br>• The task ID specified is associated with the system task, not the user task.<br>• The task associated with the task ID does not exist.<br>• The task ID does not refer to one of its immediate descendant tasks.<br>• The task specified has already terminated. |
| 22C | The address of the parameter list is invalid. |

## DEQ

### 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. The use of these SRRs should be coordinated carefully. Two programs may seek to update the resource simultaneously, leading to invalid results. Meanwhile, another program may be looking at the same data, causing more confusion.

The solution to this is the ENQ macro instruction. Using this instruction, a task can 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 instruction. But they cannot alter the contents of the resource in any way. (If necessary, review the entry titled "ENQ" on page 102.)

Use the DEQ macro instruction to release your task's control of a serially reusable resource.

The DEQ macro instruction is available in standard, list, and execute format.

The ***standard*** format of the DEQ macro instruction is:

|  |  |  |
|---|---|---|
| [label] | DEQ | (qname address,rname address[,rname length]) [,RET=HAVE / ,RET=NONE] [,RELATED=value] |

### 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 eight 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 instruction that gave the task control of the resource in the first place.

If you omit this parameter, then the RNAME is considered, by default, to be as long as 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 instruction to an ENQ macro instruction. The value you assign to this parameter has nothing to do with the execution of the macro itself. It merely relates one macro instruction (DEQ) to a macro instruction 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 Notes

- Control of a resource is surrendered under one of two circumstances:
  - The task with control issues the DEQ macro instruction.
  - The task with control ends. In this case the task terminates abnormally, since it did not release the SRR itself.
- 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

In the following example, a task is releasing a certain resource from its control.

```
LETGO DEQ (MARK,(8),16),RET=NONE
```

The QNAME of the resource can be found at the address associated with the label MARK. Its RNAME can be found at the address in register 8. Since the RNAME was specified by a register, the RNAME LENGTH was also specified—in this case,

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.

In the second example, a task is releasing a certain resource from its control.

```
DEQ ((3),RN),RET=HAVE
```

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 non-zero 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, the return and abend codes are as follows:

| Return Code | Meaning |
|---|---|
| 00 | The resource specified has been released. |
| 04 | Your task requested control of the resource but has not yet received it. This return code results if a DEQ instruction is issued within an exit routine that received control because of some interrupt. |
| 08 | 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 instruction. 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. |

## The List Format

|  |  |  |
|---|---|---|
| [label] | DEQ | (qname address,rname address[,rname length]) [,RET=HAVE / ,RET=<u>NONE</u>] [,RELATED=value],MF=L |

This format of the macro instruction 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 instruction.

## The Execute Format

|  |  |  |
|---|---|---|
| [label] | DEQ | (qname address,rname address[,rname length]) [,RET=HAVE / ,RET=<u>NONE</u>] [,RELATED=value],MF=(E,address) |

This format of the macro instruction generates code that executes the function, using a parameter list whose address you specify.

### Added Parameter

**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.

# DETACH

## Remove a Subtask From Your Virtual Storage

When you no longer have any use for a subtask for which you issued the ATTACH macro instruction, it should be removed from storage.

Use the DETACH macro instruction to remove a subtask and its task block from storage and to break the logical link between it and its immediate ancestor task.

The format of the DETACH macro instruction is:

| | | |
|---|---|---|
| [label] | DETACH | task id address |

## Parameter

**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 instruction for it. (If necessary, review the entry titled "ATTACH" on page 86.) 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 two low-order bytes at this address. GCS ignores the two high-order bytes.

You can write this parameter as an RX-type address or as register (1) through (12).

## Usage Notes

- The task that issues the DETACH instruction for a particular subtask must be the one that issued the ATTACH instruction for it in the first place.
- If a DETACH macro instruction 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 using the ESTAE macro instruction, then the former is not executed. (If necessary, review the entry titled "ESTAE" on page 108.) 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 instruction 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.

| Return Code | Meaning |
|---|---|
| 00 | Function completed normally. |

| Abend Code | Meaning |
|---|---|
| 13E | This subtask was detached in mid-execution. Therefore, it has terminated abnormally. |
| 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 instruction. |

## ENQ

### 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 invalid results. Meanwhile, another program may be looking at the same data, causing more confusion.

Use the ENQ macro instruction to request control of a serially reusable resource and to define the nature of the control sought by your task.

The ENQ macro instruction is available in standard, list, and execute format.

The ***standard*** format of the ENQ macro instruction is:

```
[label]   ENQ    (qname address,rname address  ┌  ┐ [,rname length]) ┌          ┐
                                               │,E│                  │,RET=TEST │
                                               │  │                  │          │
                                               │,S│                  │,RET=CHNG │
                                               └  ┘                  │,RET=HAVE │
                                                                     │,RET=NONE │
                                                                     │          │
                                                                     │,RET=USE  │
                                                                     └          ┘
                 [,RELATED=value]
```

### 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 as long as its 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.

**<u>NONE</u>**

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 instruction to a DEQ macro instruction.

The value you assign to this parameter has nothing to do with the execution of the macro itself. It merely relates one macro instruction (ENQ) to a macro instruction 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 Notes

- Control of a resource is surrendered under one of two circumstances:
  - A program within the task with control issues the DEQ macro instruction. Review the entry titled "DEQ" on page 96.
  - The task with control ends. In this case, the task terminates abnormally, since it did not release the resource itself.
- After it issues the ENQ instruction, your task may be placed in the WAIT state for one of the following reasons:
  - It has requested exclusive unconditional control of a resource that is under exclusive or shared control of another task.
  - It has requested control of a resource that is under the exclusive control of another task.
  - Your task requested shared control but there is a request for exclusive control ahead of it.
- The ENQ instruction affects only the tasks within the virtual machine in which it was issued. 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.
- 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 below.

## Examples

In the following example, the task is requesting exclusive, unconditional control over a certain serially reusable resource.

```
GETIT ENQ (MARK,(4),E,32)
```

The QNAME of the resource can be found at the address associated with the assembler program label MARK. The RNAME can be found at the address in register 4. Since a register was specified for the RNAME, the length of the RNAME is also specified, in this case 32. GETIT is the label on this instruction.

In the second example, the task is requesting immediate, shared control of a resource. If that resource is not immediately available, the task does not wish to wait.

```
ENQ ((3),RN,S),RET=USE
```

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 non-zero 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 are described as follows, according to whether the condition specified in the RET parameter is CHNG, RET, or USE.

When RET=CHNG, the return codes are as follows:

| Return Code | Meaning |
|---|---|
| 00 | The task now has exclusive control of the resource. |
| 04 | The task cannot get exclusive control of the resource. |
| 08 | The resource has not been queued. |
| 20 | A previous request for control of the same resource was made by this same task. The task does not have control of the resource. |

When RET=HAVE, the return codes are as follows:

| Return Code | Meaning |
|---|---|
| 00 | Control of the resource has been given to the task. |
| 08 | 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. |
| 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, the return codes are as follows:

| Return Code | Meaning |
|---|---|
| 00 | The resource is available immediately. |
| 04 | The resource is not available immediately. |

| Return Code | Meaning |
|---|---|
| 08 | 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. |
| 20 | The task has made a previous request for control of this resource. The task is not given control. |

When RET=USE, the return codes are as follows:

| Return Code | Meaning |
|---|---|
| 00 | Control of the resource has been given to the task. |
| 04 | The resource is not available immediately. |
| 08 | 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. |
| 20 | The task has made a previous request for control of this resource. The task is not given control. |

Abend codes for all functions are as follows:

| 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. |

## The List Format

```
[label]   ENQ   (qname address,rname address  ,E  [,rname length])  ,RET=TEST
                                              ,S
                                                                    ,RET=CHNG
                                                                    ,RET=HAVE
                                                                    ,RET=NONE

                                                                    ,RET=USE

                [,RELATED=value],MF=L
```

This format of the macro instruction 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 instruction.

## The Execute Format

```
[label]   ENQ   (qname address,rname address  ,E  [,rname length])  ,RET=TEST
                                              ,S
                                                                    ,RET=CHNG
                                                                    ,RET=HAVE
                                                                    ,RET=NONE

                                                                    ,RET=USE

                [,RELATED=value],MF=(E,address)
```

This format of the macro instruction generates code that executes the function, using a parameter list whose address you specify.

### Added Parameter

**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.

## ESTAE

### Specify an Exit Routine for a Task that will Gain Control if the Task ABENDs

When a task terminates abnormally, GCS usually performs task termination activities on behalf of 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 instruction to specify and describe this exit routine.

The ESTAE macro instruction is available in standard, list, and execute format.

The ***standard*** format of the ESTAE macro instruction is:

```
[label]    ESTAE     [exit address] [,CT] [,PARAM=address] [,XCTL= {NO } ]
                                    [,OV]                  [       {YES} ]

                     [,ASYNCH= {YES} ]
                     [         {NO } ]
```

### Parameters

**exit address**
Specifies the address of the exit routine that is to gain control if your task terminates abnormally.

If you specify an address of zero, then the exit routine you most recently defined via the ESTAE instruction is cancelled.

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.

Since 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. To omit a certain parameter here is to say, "leave the parameter as it is."

**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 via the XCTL macro instruction. If necessary, review the entry titled "XCTL" on page 152.

**NO**

Indicates that if your task transfers control to a module via 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 via 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.

## Usage Notes

- Your task may use the ESTAE macro instruction many times while processing. However, only the most recent exit routine specified remains current. Any others are pushed down in a stack. If the current exit routine is cancelled, then the next one in the stack percolates 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.
- The current exit routine loses its status as the current exit routine under one of the following conditions:
  - The module that defined it, via the ESTAE instruction, terminates.
  - Your task issues the ESTAE instruction, specifying zero as the EXIT ADDRESS.
  - Your exit routine terminates abnormally.
  - Your exit routine allows termination of the task that defined it to continue.
  - Your task attempts to transfer control using the XCTL instruction when XCTL=YES is not specified.

In each case, the exit routine defined by the previous ESTAE instruction percolates to the top of the stack and assumes the role of current exit routine.

- ESTAE instructions that cancel the current exit routine or overlay parameters thereof must be issued by the same program that defined the current exit routine in the first place.
- 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 normal termination activities and shut the task down.
- Whenever a task ABENDs, GCS attempts to build a system diagnostic work area (SDWA), as described in the entry titled "IHASDWA" on page 114.
- If storage was available for the SDWA, then when your exit routine receives control, the registers contain the following:

| Register | Contents |
|---|---|
| Register 0 | A return code of 16(10), signifying that no I/O processing was performed. |
| Register 1 | Address of the SDWA. |
| Register 2-12 | Unpredictable. |
| Register 13 | Address of a register save area. |
| Register 14 | A return address. |
| Register 15 | Address of the current exit routine. |

In this case, the SETRP macro instruction should be issued to notify the GCS supervisor of the action that is to be taken. If necessary, review the entry titled "SETRP" on page 119.

- If storage was not available for the SDWA, then when your exit routine receives control, the registers contain the following:

| Register | Contents |
|---|---|
| Register 0 | A return code of 12(C), signifying that no SDWA was obtained. |
| Register 1 | The completion code passed by the ABEND macro instruction. If necessary, review the entry titled "ABEND" on page 84. |
| Register 2 | The address of the parameter list intended for the exit routine. Or, if none was intended, zero. |
| Register 3-13 | Unpredictable. |
| Register 14 | Address of an SVC 3 instruction. |
| Register 15 | Address of the current exit routine. |

- 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.

| Register | Contents |
|---|---|
| Register 0 | The address of a recovery routine, if one is to be scheduled. |
| Register 15 | A return code. Specifically: **0** Termination should be continued. Any previously defined exit routines will percolate 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. |

- 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.
- If storage was available for an SDWA, then when the recovery routine gains control, the registers contain the following:

| Register | Contents |
|---|---|
| Register 0 | Zero, indicating that storage for the SDWA was available. |
| Register 1 | Address of the SDWA. |
| Register 2-13 | Unpredictable. |
| Register 14 | Address of an SVC 3 instruction. |
| Register 15 | Address of the recovery routine. |

- If storage was not available for an SDWA, then when the recovery routine gains control, the registers contain the following:

| Register | Contents |
|---|---|
| Register 0 | 12(C), indicating that storage for the SDWA was not available. |
| Register 1 | The value of the PARAM parameter that was specified in the ESTAE instruction associated with the current exit routine. |
| Register 2 | Zero. |
| Register 3-13 | Unpredictable. |
| Register 14 | Address of an SVC 3 instruction. |
| Register 15 | Address of the recovery routine. |

## Example

In the following example, the task wants to define an exit routine that will gain control in case of an abend.

```
DEFEXT ESTAE (4),CT,PARAM=PLIST3
```

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.

| **Return Code** | **Meaning** |
|---|---|
| 00 | Function completed successfully. |
| 04 | The OV parameter was specified along with a valid EXIT ADDRESS. However, either there is no current exit routine defined, or the ESTAE instruction was not issued by the same active program module that defined the current exit routine. |
| 0C | 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. |
| 14 | The ESTAE macro was unable to acquire the storage necessary for it to process. |

| **Abend Code** | **Meaning** |
|---|---|
| 13C | An invalid ESTAE request was made. |

## The List Format

```
[label]   ESTAE
                  [exit address] [,CT] [,PARAM=address] [,XCTL= {NO } ]
                                 [,OV]                  [       {YES} ]

                  [,ASYNCH= {YES} ]
                  [         {NO } ]

                  ,MF=L
```

This format of the macro instruction 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 instruction.

## The Execute Format

| | | |
|---|---|---|
| [label] | ESTAE | [exit address] [,<u>CT</u> \| ,OV] [,PARAM=address] [,XCTL={<u>NO</u> \| YES}] [,ASYNCH={<u>YES</u> \| NO}] ,MF=(E,address) |

This format of the macro instruction generates code that executes the function, using a parameter list whose address you specify.

### Added Parameter

**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.

## IHASDWA

### 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. Review the entry titled "ESTAE" on page 108 for an explanation of this.

When the ABEND macro instruction is issued for a specific task, a system diagnostic work area (SDWA) is created. If necessary, review the entry titled "ABEND" on page 84.

The SDWA is an area of storage that contains important information about the task that has just terminated abnormally. (Study the format of the SDWA provided below.) The exit routine uses this information to analyze the failure.

Use the IHASDWA macro instruction to produce a template of the system diagnostic work area that will make programming your exit routine much easier. The IHASDWA macro instruction 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.

The format of the IHASDWA macro instruction is:

| | | |
|---|---|---|
| [label] | IHASDWA | $\left[\text{DSECT=}\left\{\begin{array}{l}\underline{\text{YES}}\\ \text{NO}\end{array}\right\}\right]$ |

### Parameters

**DSECT**
Indicates that you are about to specify whether the template produced will be a DSECT (dummy control section).

**<u>YES</u>**
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 template will not be a DSECT.

### Usage Notes

- 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.

- The template is created as part of the expansion of the IHASDWA macro instruction as follows:

```
0         SDWAPARM    ESTAE parameter list address
4         SDWACMPF    Flags:
          SDWAREQ     80 ---> Dump requested
          SDWASTEP    40 ---> STEP parameter specified in ABEND instruction
5         SDWACMPC    Completion code[6]
8         SDWACTL1    BC mode PSW at entry to ABEND macro
16(10)                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
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:
          SDWAPERC    10 ---> Recovery routine percolated,
235(EB)   SDWANRBE    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
663(297)              END SDWA
```

## Return Codes and Abend Codes

The IHASDWA macro generates no return codes and no abend codes.

[6] This field contains the completion code specified in the ABEND macro instruction. The SETRP macro instruction may modify this field via its COMPCOD parameter.

# POST

## Signal a Task that the Event It is Waiting for Has Taken Place

A task that has issued the WAIT macro instruction cannot continue until a certain event has taken place. (Review the entry titled "WAIT" on page 122.) 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 instruction to inform a task that the event it is waiting for has taken place.

The format of the POST macro instruction is:

| | | |
|---|---|---|
| [label] | POST | ecb address[,completion code][,RELATED=value] |

## 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 in 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.

**RELATED**
Specifies documentation data that you are using to relate this macro instruction to a WAIT macro instruction. The value you assign to this parameter has nothing to do with the execution of the macro itself. It merely relates one macro instruction (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 Notes

- It is the dual responsibility of the task issuing the WAIT instruction and the task issuing the POST instruction to provide 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. The POST macro sets the POST bit of the appropriate ECB to 1. It then resets the WAIT bit to 0. These actions signal the waiting task that the event in question has taken place.
- The remaining thirty bits of the ECB hold the completion code, once the ECB is posted.
- Tasks are not always placed in the WAIT state after having issued the WAIT instruction. Let us say that event Z takes place. The ECB associated with that event is posted, yet no task is presently waiting for the event. Moments later, a task issues the WAIT instruction, specifying the ECB associated with event Z. Since the task is immediately satisfied, there is no reason for it to go into the WAIT state.
- It is possible for a program to perform a branch entry into the POST macro code. That is, to branch directly to the entry point in the macro code labelled CSIWAIPB. This, however, is seldom done.

  Those programmers who find it necessary to perform such a branch entry must be disabled, and running in supervisor state and key 0. Moreover, they must do the following before taking this branch.

  1. 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.
  2. Further, you must be certain that the registers contain the following information:

     | Register | Contents |
     |---|---|
     | Register 0 | The COMPLETION CODE in the low-order 30 bits. |
     | Register 1 | The address of the ECB in question. |
     | Register 13 | The address of the 224-byte save area. |
     | Register 14 | The return address within your program. |
     | Register 15 | The address of the entry point in the POST macro to which you are branching. |

  3. Since the point to which you will branch will be in low storage, use the FLS macro instruction to generate the FLS DSECT. Review the entry titled "FLS" on page 436. Include the

     ```
     USING FLS,0
     ```

     instruction in your program, and branch to the address stored at the address associated with the label FLSPOST.
- Be certain that none of your tasks changes 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

In the following example a certain event has taken place.

```
DONE POST (3),657
```

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.

The second example means the same as the first example with two exceptions: the address of the ECB is in register 8, and the completion code is 0, by default.

```
POST (8)
```

## Return Codes and Abend Codes

Note that these return codes are possible only when a branch entry to the POST macro is involved.

| **Return Code** | **Meaning** |
|---|---|
| 04 | The address of an ECB was invalid. |
| 08 | The state block that is waiting for the ECB to be posted is not in the virtual machine's task block/state block structure. |

Note that these ABEND codes are possible only during a normal SVC call from the POST macro.

| **Abend Code** | **Meaning** |
|---|---|
| 102 | The ECB in question is not addressable by the program issuing the POST instruction. |
| 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. |

## SETRP

### Set Certain Parameters in the System Diagnostic Work Area (SDWA)

Often an application identifies an exit routine for each task that will receive control if the task terminates abnormally. Review the entry titled "ESTAE" on page 108 for an explanation of this.

When the ABEND macro instruction is issued for a specific task, a system diagnostic work area (SDWA) is created. If necessary, review the entry titled "ABEND" on page 84.

The SDWA is an area of storage that contains important information about the task that has just terminated abnormally. The exit routine uses this information to analyze the failure. To appreciate the SETRP macro instruction fully, you should also have a sound understanding of the IHASDWA macro instruction. Review the entry titled "IHASDWA" on page 114.

Use the SETRP macro instruction in an exit routine that you defined via the ESTAE instruction. The SETRP macro instruction 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.

The format of the SETRP macro instruction is:

```
[label]    SETRP     [WKAREA=(reg)][,REGS=(reg1[,reg2])]

                     [,COMPCOD= { number                       } ]
                     [          { (number, [USER  ])           } ]
                     [          {          [SYSTEM]            } ]

                     [,DUMP= { IGNORE } ] [,RC= { 0                  } ]
                     [       { YES    } ] [     { 4,RETADDR=address  } ]
                     [       { NO     } ] [     {                    } ]
```

(USER and IGNORE are underlined as defaults.)

### 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 general 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. Obviously, a subset of this order is permissible, but be mindful of the order when specifying a range.

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 via the ESTAE instruction. 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 in 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 non-fetch-protected and key 14 data will be included in the dump.

**IGNORE**

Indicates that you want this SETRP instruction not to change any dump specification made by a previous SETRP or ABEND instruction. That is, whatever any previous SETRP or ABEND instruction 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 in 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 instruction.

**RC**

Specifies the return code that the exit routine you specified via the ESTAE instruction 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 execute 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).

## Example

In the following example, the task requests that certain fields in the SDWA be set and that the failed routine be tried again.

```
RETRY SETRP WKAREA=(3),REGS=(14,12),COMPCOD=635,RC=4,RETADDR=(12)
```

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 instruction.

## Return Codes and Abend Codes

The SETRP macro generates no return codes and no abend codes.

# WAIT

## Wait for an Event to Take Place Before Continuing Processing

Often a task reaches a point where it cannot continue until something else happens. For example, your task may be unable to continue until it receives input from a certain file.

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.

Use the WAIT macro instruction to cause your task to wait for a certain event to take place before your task resumes processing.

The format of the WAIT macro instruction is:

|  |  |  |
|---|---|---|
| [label] | WAIT | [number of events,] {ECB=address } [,RELATED=value] |
|  |  | {ECBLIST=address} |

## Parameters

**number of events**
Specifies the 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 instruction 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 instruction to a POST macro instruction.

The value you assign to this parameter has nothing to do with the execution of the macro itself. It merely relates one macro instruction (WAIT) to a macro instruction 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 Notes

- It is the responsibility of the task issuing the WAIT instruction to provide 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. The WAIT bit is also reset to 0. (These actions are performed by the program effecting the event your task is waiting for. This other program issues the POST macro instruction to alert your program that the event has taken place. This fact is communicated to your task through this POST bit. Review the entry titled "POST" on page 116.)

- If the program issuing the related POST instruction has chosen to pass it, then the remaining thirty bits of the ECB will contain a completion code. This code will describe the manner in which the event your program is waiting for took place.

  This completion code only has meaning to the applications involved.

- You know that the event in question has occurred when your task regains control.

- Implicit in using the ECBLIST parameter is that you do not care which of several events occurs. The occurrence of any one of the events associated with the ECBs in the list will allow your task to continue.

- Tasks are not always placed in the WAIT state after having issued the WAIT instruction. Let us say that event Z takes place. The ECB associated with that event is posted, yet no task is presently waiting for the event. Moments later, a task issues the WAIT instruction specifying the ECB associated with event Z. The task is immediately satisfied.

- Be certain to reset to zero each bit of the ECBs in question before you issue the WAIT instruction. Likewise, once your program regains control, be certain to reset these bits after the ECB is analyzed. If you do not, and the event occurs again, your program will not know it.

- No task should change any of the bits in any 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

In the following example, a task is waiting for an event to occur.

```
HOLDIT WAIT 1,ECB=(2)
```

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 instruction.

In the second example, a task is waiting for one of several events to occur.

```
WAIT ECBLIST=(4)
```

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

The WAIT macro instruction generates no return codes.

| Abend Code | Meaning |
|---|---|
| 101 | The problem program specified a number of 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. |

# Chapter 5. Program Management Service Macros

## BLDL

### 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 invoke 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 execute it, GCS needs certain information about it. It needs to know the module's name, the name of 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 instruction to build a directory entry list for each load library member expected to be invoked. 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 issue the BLDL macro instruction, 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 issue the BLDL instruction once. That way, the module can be loaded once and executed several times using the same directory entry list.

The format of the BLDL macro instruction is:

| | | |
|---|---|---|
| [label] | BLDL | 0,list address |

### 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 "GCS Commands" on page 20.

**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. These matters are discussed below.

You can write this address as an RX-type address, as register (0), or as register (2) through (12).

## The Directory Entry List

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 as shown in Table 7:

Table 7. Directory Entry List Basic Format

| LIST INFORMATION | LIST ENTRY #1 | LIST ENTRY #2 | LIST ENTRY #3 | . . . . | LIST ENTRY #64K-1 |
|---|---|---|---|---|---|

### List Information

The list information for the directory entry list is contained in the first two fields, as illustrated below. 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 58, and it must be even.

### The List Entry

As illustrated in Table 7, the directory entry list comprises one or more list entries. Each list entry corresponds to one module from a load library that you intend to invoke. 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 44) |
|---|---|---|---|---|---|

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 eight 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 zeroes.

**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 one 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 22 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.

## Usage Notes

- The only load libraries that the BLDL macro will consider are those you specify in the GLOBAL LOADLIB command.
- 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.
- If there is more than one list entry in the directory entry list, then it is wise to arrange them alphamerically according to the NAME field. However, this is not a requirement.
- 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.
- Many programmers find it convenient to use the BLDL macro instruction 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.

## 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 15. The reason code is returned to the caller in the low-order byte of register 0. There are no abend codes.

| Return Code | Reason Code | Meaning |
|---|---|---|
| 00 | 00 | Function successfully completed. |
| 04 | 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. |
| 08 | 00 | A permanent I/O error was found when GCS attempted to search a load library directory. |
| 08 | 04 | Insufficient virtual storage space was available for file management. |

# CALL

## Pass Control to an Entry Point in this or Another Control Section

Use the CALL macro instruction to pass control to an entry point in the same control section or in some other control section. Implicit in the use of this macro instruction is the fact that ultimately you expect control to return to the point from which it was passed.

The CALL macro instruction is available in standard, list, and execute formats.

The ***standard*** format of the CALL macro instruction is:

| | | |
|---|---|---|
| `[label]` | `CALL` | `entry point name[,(parameter addresses)[,VL]][,ID=number]` |

## Parameters

**entry point name**
Specifies the name of the entry point that is to receive control.

Since 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 instruction.

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 list them in the instruction. The parameter list comprises 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 whole 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 issue several CALL macro instructions.

You can assign this parameter a unique, mnemonic value that will be inserted in any dump you might request. This allows you to associate an area within the dump with a specific CALL instruction.

You can write this parameter as any number or symbol.

## Usage Notes

- If you specify the entry point name as register 15, then the load module that contains the entry point must be in virtual storage. Moreover, register 15 must contain the address of the entry point.
- Use of the CALL macro instruction implies that the issuing program ultimately expects to regain control.
- It is the responsibility of the program issuing the CALL instruction 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.
- The supervisor is not involved in passing control to the entry point. Therefore, it is your task's responsibility to maintain the reusability 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. Moreover, your task must ensure that only one user has control of the program at any given time.

## Example

In the following, the program requests that control be passed to the entry point whose address is in register 15.

```
CALL (15),(PARAM1,PARAM2),VL
```

Since register 15 is specified, GCS assumes that the entry point is in virtual storage. 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. Since 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.

## The List Format

| | | |
|---|---|---|
| [label] | CALL | [(parameter addresses)[,VL],]MF=L |

This format of the macro instruction 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 parameters listed above are valid in the list format of this instruction.

### Added Parameter

**MF=L**
Specifies the list format of this macro instruction.

## The Execute Format

|  |  |  |
|---|---|---|
| [label] | CALL | entry point name[,(parameter addresses)[,VL]][,ID=number]<br>,MF=(E,address) |

This format of the macro instruction generates code that executes the function, using a parameter list whose address you specify.

### Added Parameter

**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.

## DELETE

### Relinquish Control of a Load Module

Once 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 instruction cancels the effect of the LOAD macro instruction. (If necessary, review the entry titled "LOAD" on page 142.) Use the DELETE macro instruction to release your task's control over a load module and, if it is no longer needed, to remove it from virtual storage.

The format of the DELETE macro instruction is:

| | | |
|---|---|---|
| [label] | DELETE | { EP=symbol \| EPLOC=address \| DE=address } [,RELATED=value] |

### 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 eight bytes long. If it is less than eight 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 instruction. If necessary, review the entry titled "BLDL" on page 126.

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 instruction to a LOAD macro instruction.

The value you assign to this parameter has nothing to do with the execution of the macro itself. It merely relates one macro instruction (DELETE) to a macro instruction 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 Notes

- 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.
- The task that issues the DELETE instruction to release a given load module must be the same task that issued the LOAD instruction for it in the first place.

## Example

In the following example, the task relinquishes control over the load module containing the entry point XYZ. This DELETE instruction is cross-referenced with a related LOAD instruction by use of the RELATED parameters in each.

```
LOADIT  LOAD EP=XYZ,RELATED=DELETEIT
          .
          .
          .
DELETETIT DELETE EP=XYZ,RELATED=LOADIT
```

## Return Codes and Abend Codes

When this macro completes processing, it passes to the caller a return code in register 15.

| Return Code | Meaning |
|---|---|
| 00 | Function successfully completed. |
| 04 | Either your task did not issue a corresponding LOAD instruction, or the load module has already been deleted. |

| Abend Code | Meaning |
|---|---|
| 206 | Invalid parameter list. |

# IDENTIFY

## 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.

Use the IDENTIFY macro instruction to define an entry point in a load module.

The format of the IDENTIFY macro instruction is:

| | | |
|---|---|---|
| [label] | IDENTIFY | {EP=symbol \| EPLOC=address} ,ENTRY=address |

## 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 if you wish. It must not, however, correspond to any name, alias, or entry point name known to the system.

This name can be up to eight 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 eight bytes long. If it is less than eight 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 Notes

- 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 you previously issued a LOAD macro instruction. If necessary, review the entry titled "LOAD" on page 142.
  - The last load module given control using the OSRUN command, or the ATTACH, LINK, or XCTL macro instruction. For more information on OSRUN see "ATTACH" on page 86, "LINK" on page 137, "XCTL" on page 152, or "GCS Commands" on page 20.
- The IDENTIFY instruction cannot be issued by any asynchronous exit routine.

- You cannot use the IDENTIFY instruction to define an entry point that has been declared via the CONTENTS instruction. If necessary, review the entry titled "CONTENTS" on page 427.
- All Program Management Macros consider the code at entry points that are defined via the IDENTIFY instruction to be reentrant. You must be certain that this code is indeed reentrant, otherwise unpredictable results are possible.

## Example

In the following example, a new entry point is defined within a certain load module.

```
NAMEIT IDENTIFY EP=ABC3,ENTRY=(6)
```

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. It may be one of the following:

| Return Code | Meaning |
|---|---|
| 00 | Function successfully completed. |
| 04 | The non-main entry point name you specified is already assigned to the address you specified. |
| 08 | 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. |
| 0C | 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. |
| 10 | The IDENTIFY instruction was issued by an asynchronous exit routine. The entry name was not assigned to the address you specified. |
| 14 | An IDENTIFY instruction was previously issued defining the same non-main entry point name but at a different address. The entry point name specified in the present IDENTIFY instruction was not assigned to the address specified. |
| 18 | The parameter list was invalid. The entry point name was not assigned to the address you specified. |
| 28 | 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. |

## LINK

### Pass Control to a Program, Expecting to Regain Control Later

GCS provides several techniques for passing control from one program to another. Use the LINK macro instruction 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.

The LINK macro instruction is available in standard, list, and execute formats.

The *standard* format of the LINK macro instruction is:

```
[label]      LINK        ( EP=symbol   )
                        <  EPLOC=addr  > [,ID=number][,PARAM=(addresses)[,VL=1]]
                         ( DE=addr     )
```

### 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 via the IDENTIFY macro instruction. If necessary, review the entry titled "IDENTIFY" on page 135.
- The name of the entry point declared in a shared segment directory via the CONTENTS macro instruction. If necessary, review the entry titled "CONTENTS" on page 427.
- 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, since the module associated with the entry point name may already be loaded.
2. Any shared segment directories that may have been created via the CONTENTS macro instruction.
3. The directories of any load libraries that may have been defined for your virtual machine via the GLOBAL LOADLIB command. For more information on the GLOBAL command see "GCS Commands" on page 20.

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 eight bytes long. If it is fewer than eight 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 instruction. (If necessary, review the entry titled "BLDL" on page 126.)

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 program being called.

GCS builds a parameter list containing these addresses in the order in which you specify them. Then, the system passes the address of this parameter list to the program being 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 being 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 being called expects a set number of parameters.

## Usage Notes

- If you issue the LINK instruction 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 issue the LOAD instruction yourself just one time. This avoids all the overhead processing involved in having GCS repeatedly load the module for you!

- The relationship between the program issuing the LINK instruction and the program receiving control is the same as that established by a BAL assembler language instruction. Once the program being called has completed execution, control is returned to the program that issued the LINK instruction.

- It is the responsibility of the program issuing the LINK instruction to provide the program receiving control with the address of an area wherein the former's registers will be saved. This address must be placed in register 13 by the program issuing the LINK instruction.

- Likewise, it is the responsibility of the program being called to place the value of the other program's registers in this save area once 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.
- You can use the LINK instruction to link to a serially reusable program. If the program is being used by someone else, then you will placed in the WAIT state until the other user is finished.
- 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

In the following example, control is passed to an entry point named PROGRAMB.

```
LINK EP=PROGRAMB,PARAM=(ADDRA,ADDRB,ADDRC)
```

PROGRAMB expects exactly three parameters be passed to it. These parameters may be found at addresses ADDRA, ADDRB, and ADDRC, respectively.

In the second example, control is passed to an entry point whose name can be found at the address corresponding to the label PROGADDR.

```
LINKIT LINK EPLOC=PROGADDR,PARAM=((2),(3)),VL=1
```

PROGADDR 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 the second in register 3. LINKIT is the label on this instruction.

In the last example, control is passed to a certain entry point.

```
LINK DE=BLDLNAM,ID=6
```

The system is to look 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 is to place the value 6 in bytes three and four of the final instruction that it generates.

## Input to the Program Receiving Control

| Register | Contents |
|---|---|
| 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 once the called program completes execution. |
| Register 15 | The address of the entry point in the program being called. |

## Return Codes and Abend Codes

No return codes are generated. The abend codes are as follows:

| 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 unrecoverable 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 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. |

## The List Format

```
[label]    LINK     ┌            ┐
                    │EP=symbol   │
                    │EPLOC=addr  │ [,ID=number],SF=L
                    │DE=addr     │
                    └            ┘
```

This format of the macro instruction 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 parameters listed above are valid in the list format of this instruction.

## Added Parameter

**SF=L**
Specifies the list format of this macro instruction.

## The Execute Format

```
[label]   LINK    ┌            ┐
                  │EP=symbol   │                 ⎧,SF=(E,addr)[,options]          ⎫
                  │EPLOC=addr  │ [,ID=number]   ⎨,MF=(E,address)                  ⎬
                  │DE=addr     │                 ⎩,SF=(E,addr),MF=(E=address)     ⎭
                  └            ┘

                  options:
                  PARAM=(addrs)[,VL=1]
```

This format of the macro instruction generates code that executes the function, using a parameter list whose address you specify.

Note that only the parameters listed above are valid in the execute format of this instruction.

## Added Parameter

**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 via the list format of this instruction.

You can add or modify values in this parameter list by specifying them in this instruction.

**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 instruction.

# LOAD

## Bring a Load Module that You Intend to Invoke More than Once into Virtual Storage

Use the LOAD macro instruction to bring a load module, containing a specified entry point, into virtual storage. This makes the code at that entry point available for your use.

The format of the LOAD macro instruction is:

```
[label]    LOAD     ( EP=symbol      )
                   <  EPLOC=address   > [,RELATED=value]
                    ( DE=address     )
```

## 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 via the IDENTIFY macro instruction. If necessary, review the entry titled "IDENTIFY" on page 135.
- The name of the entry point declared in a shared segment directory via the CONTENTS macro instruction. If necessary, review the entry titled "CONTENTS" on page 427.
- 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, since the module associated with the entry point name may already be loaded.
2. Any shared segment directories that may have been created via the CONTENTS macro instruction.
3. The directories of any load libraries that may have been defined for your virtual machine via the GLOBAL LOADLIB command. For more information on the GLOBAL command, see "GCS Commands" on page 20.

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 eight bytes long. If it is fewer than eight bytes long, it must be padded on the right with blanks. Again, the entry point name can refer to one of the same three things as 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.

GCS assumes that you have created this list entry within the directory using the BLDL macro instruction. Review the entry titled "BLDL" on page 126. When using the BLDL macro instruction for this particular purpose, be certain to specify at least 60 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 instruction to a DELETE macro instruction.

The value you assign to this parameter has nothing to do with the execution of the macro itself. It merely relates one macro instruction (LOAD) to a macro instruction 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.

## Usage Notes

- 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.
- 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 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.
- For each LOAD instruction that you issue, you must also issue a corresponding DELETE instruction. Review the entry titled "DELETE" on page 133.
- 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.

## Example

The following is an example of bringing a load module containing the entry point XYZ into virtual storage. This LOAD instruction is cross-referenced with a related DELETE macro instruction by use of the RELATED parameters in each.

```
LOADIT  LOAD EP=XYZ,RELATED=DLEETIT
         .
         .
         .
DLEETIT DELETE EP=XYZ,RELATED=LOADIT
```

## Return Codes and Abend Codes

The program issuing the LOAD macro instruction receives the following information in its registers.

| Register | Contents |
|---|---|
| Register 0 | The address of the entry point specified in the LOAD instruction. |
| Register 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. |
| Register 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 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 | | Invalid parameter list. |
| 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 unrecoverable I/O error occurred when the GCS supervisor 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 and/or USE COUNT for the load module have reached the maximum of 32767. |

## RETURN

### Return Control to a Program

Use the RETURN macro instruction to return control from one program to the program that called it.

The RETURN macro instruction 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.

The format of the RETURN macro instruction is:

| | | |
|---|---|---|
| [label] | RETURN | (reg1[,reg2])[,T][,RC=number] |

### Parameters

**(reg1) or**
**(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 instruction uses the same conventions for restoring registers that the SAVE macro instruction 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 above general order is permissible but be mindful of 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 saved the values in the first place) 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. In this case, 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 Note

- 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.

## Example

In the following example, the program requests that control be returned to the program that called it.

```
GOBACK RETURN (14,7),T,RC=40
```

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

### Save the Contents of the Registers

By convention, it is the responsibility of any program called by another to save the values in the registers when it receives control. Likewise, it is the responsibility of the calling program to provide storage wherein 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.

Use the SAVE macro instruction in a called program to save the values of certain registers belonging to the program that called it.

Note that the SAVE macro uses the standard conventions for saving registers. That is, they are saved in an area composed of eighteen 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.

The format of the SAVE macro instruction is:

| | | |
|---|---|---|
| [label] | SAVE | (reg1[,reg2])[,T][,id name] |

### Parameters

**(reg1) or**
**(reg1,reg2)**

Specifies the single register (reg1) or range of registers (reg1,reg2) whose values are to be stored in the save area.

To specify a range of registers, consider the general order in 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 be mindful of 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 as well as those in another subset of registers, of which 14 and 15 are not a part. 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 instruction.

You can use this identifier as a debugging aid when you issue several SAVE instructions. You can assign this parameter a unique, mnemonic value that will be inserted in any dump you might request. This allows you to associate a section within the dump with a specific SAVE instruction, and thereby with a specific save area.

A byte containing the length of the ID NAME appears in the dump four bytes after the address in register 15. The ID NAME itself begins five 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 Note

- The SAVE macro instruction 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 instruction, which it might not were the SAVE instruction issued later.

## Examples

In the following example, the program requests that the values in registers 14, 15, and registers 0 through 12 be saved.

```
SAVE (14,12),,*
```

Since 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.

In the second example, the program requests that the values in registers 5, 6, and 7 be saved.

```
SAVE (5,7),T
```

Since registers 14 and 15 are not within this range and since 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.

## SYNCH

### Schedule a Synchronous Exit from One Program to Another, Possibly with a Change in State

The SYNCH macro instruction schedules a synchronous exit from one program to another. If desired, the SYNCH macro instruction allows a supervisor state program to call another program and choose the state in which the latter will function. In addition, the SYNCH macro instruction allows you to control the restoration of registers belonging to the calling program.

The SYNCH macro instruction is available in standard, list, and execute formats.

The ***standard*** format of the SYNCH macro instruction is:

| | | |
|---|---|---|
| [label] | SYNCH | entry point [,RESTORE= {YES / NO}] [,STATE= {SUPV / PROB}] |

### 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 instruction. 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.

**STATE**
Indicates the state in 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 will function in supervisor state.

**PROB**
Indicates that the program being called will function in problem state.

## Usage Notes

- The SYNCH macro makes no validity checks on the entry point address. Regardless of what is at that address, control is transferred to it.
- It is not necessary for the program that issues the SYNCH instruction 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.
  - If the program issuing the SYNCH instruction is a supervisor state program, then there is a choice. Use the STATE parameter to specify in which state the called program is to function.
- It is important to remember that any program called via the SYNCH instruction 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 that of the problem state program before it issues the SYNCH instruction.
- It is risky to use the SYNCH macro instruction to transfer control to a program that is not reentrant. While this practice is not prohibited, the results are unpredictable.

## Examples

In the following example, the first SYNCH macro instruction schedules an exit to the entry point whose address is in register 2.

```
SYNCH (2),RESTORE=NO,STATE=SUPV
```

The program being 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 instruction, no registers will be restored.

The second SYNCH macro instruction was issued to schedule an exit to an address named ENCRYPT.

```
SYNCHIT SYNCH ENCRYPT,RESTORE=YES,STATE=PROB
```

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.

## Input to the Exit Program

The exit program receives the following information in its registers.

| Register | Contents |
|---|---|
| Registers 0-13 | Unchanged. |
| Register 14 | The address to which control is to return once the exit program completes execution. |
| Register 15 | The address of the entry point in the exit program being called. |

## Return Codes and Abend Codes

The SYNCH macro generates no return codes.

| Abend Code | Reason Code | Meaning |
|---|---|---|
| 106 | 0C | 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. |

## The List Format

| | | |
|---|---|---|
| [label] | SYNCH | [RESTORE= {YES, <u>NO</u>}] [,STATE= {SUPV, <u>PROB</u>}] ,MF=L |

This format of the macro instruction 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 parameters listed above are valid in the list format of this instruction.

### Added Parameter

**MF = L**
Specifies the list format of this macro instruction.

## The Execute Format

| | | |
|---|---|---|
| [label] | SYNCH | entry point [,RESTORE= {YES, <u>NO</u>}] [,STATE= {SUPV, <u>PROB</u>}] ,MF=(E,address) |

This format of the macro instruction generates code that executes the function, using a parameter list whose address you specify.

### Added Parameter

**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.

## XCTL

### 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 instruction allows you to pass control from one program to another, with the former never again to regain it.

The XCTL macro instruction is available in standard, list, and execute formats.

The ***standard*** format of the XCTL macro instruction is:

```
[label]     XCTL      [(reg1[,reg2])] ⎧,EP=symbol ⎫
                                      ⎨,EPLOC=addr⎬
                                      ⎩,DE=addr   ⎭
```

### Parameters

**(reg1), or**
**(reg1,reg2)**
Specifies the register, or range of registers, that was saved by the issuer of the XCTL instruction and that is to be restored and passed to the program being called. The saving of these registers, then, becomes the responsibility of the program being called by the XCTL macro instruction.

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.

Be certain to 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 is to receive control.

The entry point name can be any one of the following:

- The name of the entry point as previously defined via the IDENTIFY macro instruction. If necessary, review the entry titled "IDENTIFY" on page 135.
- The name of the entry point declared in a shared segment directory via the CONTENTS macro instruction. If necessary, review the entry titled "CONTENTS" on page 427.
- 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, since the module associated with the entry point name may already be loaded.
2. Any shared segment directories that may have been created via the CONTENTS macro instruction.

3. The directories of any load libraries that may have been defined for your virtual machine via the GLOBAL LOADLIB command. For more information on the GLOBAL command see "GCS Commands" on page 20.

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 eight bytes long. If less than eight 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 instruction. Review the entry titled "BLDL" on page 126.

You can write this parameter as an assembler program label or as register (2) through (12).

## Usage Notes

- If you issue the XCTL instruction 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 issue the LOAD instruction once yourself. This avoids all the overhead processing involved in having GCS repeatedly load the module for you. If necessary, review the entry titled "LOAD" on page 142.

- It is the responsibility of the program issuing the XCTL instruction 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 instruction is issued. Registers 2 through 12 (or a subset thereof) can be restored at the same time or via the (reg1,reg2) parameter.

  The program issuing the XCTL macro instruction 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 instruction to restore registers 2 through 14 by itself.

- It is the responsibility of the program receiving control via the XCTL instruction to save the registers that the program that called it was saving.

- The program being called, using the standard format of the XCTL instruction, may expect certain parameters be passed to it. Since the program is using the standard format of the instruction, it must see to it that register 1 contains the address of the parameter list, if one is expected.

- You can use the XCTL instruction 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.

- 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

In the following example, the XCTL macro will first restore registers 2 through 12, which the program issuing it was saving.

```
XCTL (2,12),EP=PROGRAMC
```

GCS assumes that this program restored registers 13 and 14 on its own. Then control will pass to a program named PROGRAMC.

In the second example, control is passed to an entry point whose name can be found at the address in register 6.

```
TRANSCTL XCTL (4),EPLOC=(6)
```

The XCTL macro need only restore register 4. GCS assumes that the program issuing the XCTL instruction restored registers 2, 3, and 5 through 14. TRANSCTL is the label on this instruction.

In the last example, control is passed to a certain entry point.

```
XCTL DE=BLDLNAM
```

The system is to look 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. The XCTL macro need restore no registers. GCS assumes that they were all restored by the program issuing the XCTL instruction.

## Return Codes and Abend Codes

When control is passed to the program being called, the registers contain the following information.

| Register | Contents |
|---|---|
| Register 0-13 | Unchanged. Register 1 contains the address of the parameter list, if it was specified. |
| Register 14 | The address to which control is to return once the called program completes execution. |
| Register 15 | The address of the entry point in the program being called. |

| 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 unrecoverable 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 and/or USE COUNT for the load module have reached the maximum of 32767. |
| A06 | | Your task is already waiting for this serially reusable module. |

## The List Format

```
[label]     XCTL     [(reg1[,reg2])]  ┌            ┐
                                      │,EP=symbol  │,SF=L
                                      │,EPLOC=addr │
                                      │,DE=addr    │
                                      └            ┘
```

This format of the macro instruction 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 instruction.

## The Execute Format

```
[label]   XCTL   ┌                          ┐ ┌                                  ┐
                 │EP=symbol                 │ │,SF=(E,addr)[,options]            │
                 │EPLOC=addr [,ID=number]   │ │,MF=(E,address)                   │
                 │DE=addr                   │ │,SF=(E,addr),MF=(E=address)       │
                 └                          ┘ └                                  ┘

                 options:
                 PARAM=(addrs)[,VL=1]
```

This format of the macro instruction generates code that executes the function, using a parameter list whose address you specify.

Note that only the parameters listed above are valid in the execute format of this instruction.

### Added Parameters

**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 via the list format of this instruction.

You can add or modify values in this parameter list by specifying them in this instruction.

**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 being called.

You can write these parameters as assembler program labels or as registers (2) through (12).

**VL=1**
Indicates that the program being 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 being called expects a set number of parameters.

# Chapter 6. Timer Service Macros

## STIMER

### Set a Timer

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.

Use the STIMER macro instruction to set a timer to a given time period. When time is up, your task will be notified.

The format of the STIMER macro instruction is:

| | | |
|---|---|---|
| [label] | STIMER | {REAL[,exit routine address] \| WAIT} {,BINTVL=address \| ,DINTVL=address \| ,TOD=address} |

### 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 time interval.

This exit routine must be resident in virtual storage and can be specified only with the REAL parameter.

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).

**DINTVL**

Specifies the address containing the duration of time allocated for the event.

You must store the amount of time as unpacked decimal digits in a doubleword on a doubleword boundary in the following format:

**HHMMSSth**

**HH** stands for the number of hours; **MM** for the number of minutes; **SS** for the number of seconds; **t** for the number of tenths of a second; and **h** for the number of hundredths of a second. The maximum amount of time you can specify is twenty-four hours.

You can write this parameter as an RX-type address or as register (1) through (12).

**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 unpacked decimal digits in a doubleword on a doubleword boundary. Moreover, you must store it according to the **HHMMSSth** format described above, using twenty-four hour clock notation.

## Usage Notes

- It is the responsibility of the task issuing the STIMER instruction to provide storage for the amount of time. Likewise, the task must see to it that the appropriate time value is stored there before issuing the STIMER instruction.
- 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.
- 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 instruction. Once your exit routine completes execution, it returns control to the supervisor.

  Input to the exit routine is:

| **Register** | **Contents** |
|---|---|
| Registers 0 - 12 | Unpredictable. |
| Register 13 | The address of a supervisor-provided save area. |
| Register 14 | The address to which control will transfer once the exit routine completes processing. |
| Register 15 | The address of the exit routine. |

- No task can have more than one timer set at the same time. If you issue an STIMER instruction before the time period associated with a previous STIMER instruction expires, then the second STIMER instruction cancels and replaces the first.
- All time is measured continuously in real time.

## Examples

In the first example, the task wishes to set a timer.

```
CLOCKIT STIMER REAL,(6),TOD=(7)
```

Since 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.

In the second example, the task wishes to set a timer.

```
STIMER WAIT,DINTVL=(5)
```

Since 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.

| Abend Code | Meaning |
|---|---|
| 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. |

## TIME

### Request Today's Date and the Correct Time

Use the TIME macro instruction to ask the supervisor to send today's date and the correct time of day to your program.

The format of the TIME macro instruction is:

```
[label]    TIME     ┌   ┐
                    │DEC│
                    │   │
                    │BIN│
                    └   ┘
```

(DEC is underlined as the default.)

### Parameters

<u>DEC</u>
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:

**HHMMSSth**

**HH** stands for the number of hours; **MM** for the number of minutes; **SS** for the number of seconds; **t** for the number of tenths of a second; and **h** for the number of hundredths of a second.

Today's date is also returned to your program in packed decimal form.

If you omit all parameters from the TIME instruction, 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.

Today's date, however, will be returned to your program in packed decimal form.

### Usage Notes

- The time of day is returned to your program in register 0.
- Today's date is returned to your program in register 1.
- The date is stored in the following format:

  **00YYDDDF**

  **00** is a byte of zeroes. **YY** are the last two digits of the year. **DDD** is the Julian day of the year. **F** is a four-bit sign character that helps you unpack and print the date, if you request it.
- Note that the accuracy of the time and date depends upon 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

### Cancel a Timer

Use the TTIMER macro instruction to cancel a timer that you set via an STIMER macro instruction. If necessary, review the entry titled "STIMER" on page 158.

The format of the TTIMER macro instruction is:

| | | |
|---|---|---|
| [label] | TTIMER | CANCEL |

### Parameter

**CANCEL**
Indicates that you wish to cancel the effect of the last STIMER instruction. That is, the timer is to stop keeping track of elapsed time. Also, the specified branch to an exit routine, if any, is cancelled.

This is the only parameter on the TTIMER instruction and is required.

### Usage Note

- The TTIMER instruction has no effect if the STIMER instruction 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. |

# Chapter 7. Console I/O Service Macros

# WTO

## 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 instruction for this purpose. Use of this macro instruction implies that you do not require a response to your message.

The WTO macro instruction is available in standard, list, and execute formats.

The *standard* format of the WTO macro instruction is as follows.

| | | |
|---|---|---|
| `[label]` | `WTO` | `'message'` |

## Parameter

'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 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.

## Usage Notes

- GCS does not support multiple line messages. Nor does it support multiple console message handling.
- 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. |

## The List Format

| | | |
|---|---|---|
| `[label]` | `WTO` | `'message',MF=L` |

This format of the macro instruction generates an in-line parameter list, based on the parameter values that you specify. However, this format generates no executable code.

### Added Parameter

**MF=L**
Specifies the list format of this macro instruction.

## The Execute Format

| | | |
|---|---|---|
| [label] | WTO | MF=(E,address) |

This format of the macro instruction generates code that executes the function using a parameter list whose address you specify.

Note that only the parameters listed above are valid in the execute format of this instruction.

### Added Parameter

**MF=(E,address)**
ADDRESS specifies the address of the parameter list to be used by the macro.

# WTOR

## 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 instruction to send a message to the virtual machine console, to which you expect a reply.

The WTOR macro instruction is available in standard, list, and execute formats.

The *standard* format of the WTOR macro instruction is as follows.

| | | |
|---|---|---|
| [label] | WTOR | 'message',reply address,reply length,ecb |

## 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. Since 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-justified 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 in the entries titled "WAIT" on page 122 and "POST" on page 116.

You can write this parameter as an assembler program label or as register (2) through (12).

## Usage Notes

- 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.
- GCS does not support multiple line messages. Nor does it support multiple console message handling.
- GCS performs no translation on your message at all. It is transmitted exactly as coded.

## 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. |

## The List Format

|  |  |  |
|---|---|---|
| [label] | WTOR | 'message'[,reply address][,reply length][,ecb],MF=L |

This format of the macro instruction 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 instruction.

## The Execute Format

|  |  |  |
|---|---|---|
| [label] | WTOR | [reply address][,reply length][,ecb],MF=(E,address) |

This format of the macro instruction generates code that executes the function using a parameter list whose address you specify.

Note that only the parameters listed above are valid in the execute format of this instruction.

## Added Parameter

**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.

# Chapter 8. Unauthorized GCS Service Macros

# AUTHCALL

## 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.[7] 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.

The AUTHCALL macro instruction allows an unauthorized program to call an authorized program. However, AUTHCALL is not an authorized GCS function.

The format of the AUTHCALL macro instruction is:

| | | |
|---|---|---|
| [label] | AUTHCALL | {EP=name / EPLOC=addr} [,UWORD=addr] |

## 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 eight bytes.

**UWORD**
Specifies an optional fullword parameter 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.

[7] In this context, an "authorized program" is one running in supervisor state. An "unauthorized program" is one running in problem state.

## Usage Notes

- It is impossible for an unauthorized program to call an authorized program via the AUTHCALL instruction unless the AUTHNAME instruction is issued for that authorized program first. If necessary, review the entry titled "AUTHNAME" on page 192.
- Any program invoked via the AUTHCALL instruction runs in key 0.

## Examples

In the following example, the first AUTHCALL macro instruction calls an authorized program named PAT.

```
AUTHCALL EP=PAT
```

In the second example, the AUTHCALL macro instruction is used to call an authorized program whose name can be found at the address in register 2.

```
AUTHCALL EPLOC=(2),UWORD=(5)
```

Register 5 contains information that the program expects to receive from the program that called it.

## Input to the Authorized Program

The program being called receives the following information in its registers.

| Register | Contents |
|---|---|
| Register 0 | The user word (UWORD) specified in the associated AUTHNAME instruction. |
| Register 1 | The user word (UWORD) specified in the AUTHCALL instruction. |
| Register 13 | The address of the register save area. |
| Register 14 | The address to which control is to return once the authorized program completes execution. |
| Register 15 | The address of the entry point in the program being called. |

## Return Codes and Abend Codes

Except for the return code noted below, the authorized program will pass its return code to the program that called it in register 15. The AUTHCALL macro generates the return code described below.

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.

| Return Code | Meaning |
|---|---|
| -3 | The system could not find the program whose address you specified. |

| 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 instruction, issued by GCS, was unable to obtain enough storage to complete your request. |

## CMDSI

### Issue a Command from a Program

Occasionally you will find it necessary to issue a "command" from a program running under GCS. In using the word "command" we do not refer to the normal instructions peculiar to the programming language you are using. Obviously your program will contain those. "Command," in this context, means one of several things.

- Any command that ordinarily would be issued directly from the console. This includes GCS commands, CP commands, and EXECs.
- Any command that you previously defined to GCS using the LOADCMD command. For more information on the LOADCMD command see "GCS Commands" on page 20.

If you include any such command in your program, you need a way of telling GCS, lest it mistake the command for something else or not recognize it at all. Use the CMDSI macro instruction to identify to GCS a "command" that you have included in one of your programs.

The CMDSI macro instruction is available in standard, list, list address, and execute formats.

The ***standard*** format of the CMDSI macro instruction is:

| | | |
|---|---|---|
| `[label]` | `CMDSI` | `command name[,length][,FILEBLK=addr][,ERROR=addr]` |

### Parameters

**command name**
Specifies the command in question, with any necessary parameters or options.

You can specify it using one of the following:

| | |
|---|---|
| **COMMAND TEXT** | The actual text of the command, with any necessary parameters or options. The entire command statement must be surrounded by single quotation marks.<br><br>The LENGTH parameter need not be specified when using this method. If it is, it will be ignored. |
| **SYMBOL** | 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. |
| **REGISTER** | The register containing the address of the command in question. Again, with this method, you must specify the LENGTH parameter. Also, the reference to the register must be in parentheses. |

**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 in one of the following instances:

- If the System Product Interpreter is to execute a non-GCS file.
- If the Interpreter is to execute from storage.
- If the address environment is inconsistent with the filetype of the file containing the command.

This parameter specifies the address of the file block to be passed to the System Product Interpreter, which will interpret the code associated with the command.

This file block contains information necessary to invoke the code properly. This includes, among other things, the filename, filetype, and filemode of the file containing the code; its address (if in storage), and its size. If necessary, consult the *VM/XA SP System Product Interpreter User's Guide* or the *VM/XA SP System Product Interpreter Reference* for more on this topic.

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 that this error routine does not receive control if an error occurs in the command you are trying to execute. If you omit this parameter and an error occurs, then control returns to the instruction immediately following the CMDSI instruction, just as it would were there no error.

You can write this parameter as a programming language label or as register (2) through (12).

## Examples

The first example issues the command at the address associated with the label MYCMD.

```
DOIT CMDSI MYCMD,48,FILEBLK=(8),ERROR=(6)
```

The command issued is 48 characters long. Since the command invokes 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 issued for the command. DOIT is the label on this instruction.

The second example issues the GCS QUERY DISK command.

```
INQUIRE CMDSI 'QUERY DISK',ERROR=ERR1
```

No length for the issued command is needed, since 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 in case of an error 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 below. 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 invoked.

| Return Code | Meaning |
|---|---|
| 0 | The command was successfully executed. |
| -3 | The command could not be found. |

| Abend Code | Reason Code | Meaning |
|---|---|---|
| FCA | 0300 | The parameter list was invalid. |

## The List Format

| | | |
|---|---|---|
| `[label]` | `CMDSI` | `MF=L[,command name][,length][,FILEBLK=addr]` |

This format of the macro instruction 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 parameters listed above are valid in the list format of this instruction.

### Added Parameter

**MF=L**
Specifies the list format of this macro instruction.

## The List Address Format

| | | |
|---|---|---|
| `[label]` | `CMDSI` | `MF=(L,address[,label])[,command name][,length]` `[,FILEBLK=addr]` |

This format of the macro instruction 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 issue the instruction using this format, then you must do so before any related invocation of the instruction using the execute format. Also, note that only the parameters listed above are valid in the list format of this instruction.

### Added Parameter

**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 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.

## The Execute Format

| | | |
|---|---|---|
| [label] | CMDSI | MF=(E,address)[,command name][,length] [,FILEBLK=addr][,ERROR=addr] |

This format of the macro instruction generates code that executes the function, using a parameter list whose address you specify.

### Added Parameter

**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.

# EXECCOMM

## Access and Manipulate REXX Variables

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.

Use the EXECCOMM macro instruction to set up the interface that allows a program to gain access to the variables within the EXEC that invoked it.

The format of the EXECCOMM macro instruction is:

| | | |
|---|---|---|
| [label] | EXECCOMM | REQLIST=address |

## Parameter

**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. In addition, it describes how that variable will be used. Your program must create one shared variable request block for each variable to which it wants access. Moreover, 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 *VM/XA SP System Product Interpreter Reference*. This book also provides an appendix titled "REXX in the GCS Environment", which you may find helpful.

You can write this parameter as an RX-type address or as register (2) through (12).

## Usage Notes

- 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 System Product 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.
- EXEC variables may be inspected, modified, or deleted by a program that gains access to them.
- 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.

| Return Code | Meaning |
|---|---|
| 0 OR ANY POSITIVE NUMBER | Function completed successfully. |
| -1 | No EXEC was active within the task. |
| -2 | Insufficient storage is available to process your request. |

| Abend Code | Reason Code | Meaning |
|---|---|---|
| FCB | 0D01 | An invalid address exists in a shared variable request block, or the address of the block itself is invalid. |

## GENIO

### Use General Input/Output Devices

The GENIO macro instruction allows a program to obtain, use, and release any I/O device.[8] It is an unauthorized GCS function, except for GENIO STARTR, which is an authorized function.

The GENIO macro instruction is available in standard, list, list address, and execute formats.

The *standard* format of the GENIO macro instruction is:

```
[label]      GENIO      ⎧OPEN,EXIT=exit[,UWORD=uword]⎫ ,DEV=dev[,ERROR=addr]
                        ⎪CLOSE                        ⎪
                        ⎪CHAR,DATA=address            ⎪
                        ⎨MODIFY,CCW=address           ⎬
                        ⎪START,CCW=address            ⎪
                        ⎪STARTR,CCW=address           ⎪
                        ⎩HALT                         ⎭
```

### Parameters

**OPEN**

Indicates that the device specified in the instruction 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 UWORD.

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 one of three conditions:

- An I/O interrupt occurs on the device that was opened, signalling the end of an I/O operation.
- An I/O operation terminates because of error.
- An asynchronous interrupt occurs.

This exit routine is responsible for handling all interrupts occurring on the specified device.

[8] Except for DASD devices and the virtual machine console.

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.

**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 instruction with this parameter had to have opened the device in the first place. The use of 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. And, all I/O activity for the device is terminated.

Remember that your exit routine cannot receive control resulting from an interrupt occurring on a closed device. Also, remember that the GENIO instruction, with the CLOSE parameter specified, cannot be issued from an I/O exit routine.

**CHAR**
Indicates that the characteristics of the specified device should be returned to the program making the request. These characteristics include such things as the device's class, type, and model.

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, issue the GENIO instruction 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:

- Change a TIC instruction to a NOP instruction.

- Change a NOP instruction to a TIC instruction.
- Change 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 terminates 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 instruction 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. Moreover, the program is responsible for building 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 macro instructions. Review the entries titled "PGLOCK" on page 204 and "PGULOCK" on page 206.

**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.

**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.

**DEV**
Specifies the virtual address of the I/O device that the GENIO macro instruction 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 two 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 Notes

- If you request it, GCS will return the characteristics of the I/O device to a storage area your program has reserved for this purpose. This information is returned in a specific format. For information on this specific format, I/O device classes, types, and models, consult the *VM/XA SP CP Programming Services.*
- It is an error if you issue the GENIO macro instruction with the START, STARTR, HALT, MODIFY, or CLOSE parameter specified before the device has been opened.
- Only an authorized supervisor state program can issue the GENIO instruction 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 thereby bypassed.

  An unauthorized program must use the START parameter.

  A virtual machine must be authorized to issue the DIAGNOSE X'98' code. This authorization is granted by specifying DIAG98 in the directory entry of the virtual machine (OPTION statement). If the machine is not authorized for DIAGNOSE X'98', a return code is passed to the program issuing the GENIO STARTR function. Refer to *VM/XA SP CP Programming Services* for more detailed information on DIAGNOSE X'98'.
- 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. Moreover, I/O interrupts are handled after the exit routine terminates.
- 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 via 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 instruction.

  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.
- 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.
- 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.

- The GENIO macro passes the following information to the exit routine.

| Register | Contents |
|---|---|
| Register 0 | The UWORD parameter, as specified when the device in question was opened. |
| Register 1 | The address of the interrupt control block, defined below. |

The contents of the interrupt control block are as follows:

| Offset | Contents | |
|---|---|---|
| 0 (0) | Flag byte<br>Synchronous Interrupt = 00<br>Asynchronous Interrupt = 01 | X |
| 1 (1) | Reserved | 3X |
| 4 (4) | Device address | F |
| 8 (8) | Channel status word (CSW) | D |
| 10 (16) | Sense bytes | 6F |
| 28 (40) | End | |

If there was a unit check and the sense data could not be obtained, then the first two bytes of the sense data will contain X'107E'.

Even though 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 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 CSW stored as the result of a PCI will be saved in the interrupt control block.

## Examples

In the following examples, the three GENIO macro instructions are issued by the same program, affecting the same device.

In the first example, the program requests that the device be opened.

```
GENIO OPEN,DEV=(2),EXIT=GOODBYE
```

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.

In the second example, the program now asks that the device it just opened be started.

```
GENIO START,DEV=(2),CCW=(3)
```

Register 3 contains the address of the first CCW in the channel control program to be executed. If the device is not busy, then the I/O operation is begun. When the operation is finished, the exit program at the address associated with the label GOODBYE receives control.

The third example is of an GENIO macro issued when the program no longer needs the device and wants to close it.

```
GENIO CLOSE,DEV=DEVADDR
```

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. The return codes are tabulated below according to general application to the GENIO macro or to one of the specific GENIO functions. The table of abend codes follows the return code tables.

**General Return Codes:**

| Return Code | Meaning |
|---|---|
| 0 | Function completed successfully. |
| 48 | An invalid function was requested. |
| 52 | No device address was specified. |

**For the OPEN Function:**

| Return Code | Meaning |
|---|---|
| 4 | DASD devices cannot be opened as general I/O devices. |
| 8 | The specified I/O device does not exist. |
| 12 | The specified device is already opened. |
| 16 | You did not specify the address of an exit routine. |

**For the CLOSE function:**

| Return Code | Meaning |
|---|---|
| 4 | The specified device is not opened. |
| 8 | The program that closes a device must be the same as the one that opened it. |

| Return Code | Meaning |
|---|---|
| 12 | An I/O exit routine cannot issue the GENIO macro instruction with the CLOSE parameter specified. |

**For the CHAR function:**

| Return Code | Meaning |
|---|---|
| 8 | The device specified does not exist. |
| 12 | The data area for storage of the device characteristics was not specified. |

**For the MODIFY function:**

| Return Code | Meaning |
|---|---|
| 4 | No I/O is active on the device. |
| 8 | The device is not open or not operational. |
| 12 | The specified CCW address is not accessible to you. |
| 16 | The specified CCW address does not fall on a doubleword boundary. |
| 20 | No CCW could be found that corresponds with the specified address and/or device. |
| 24 | The CCW is neither a TIC nor a NOP instruction. |
| 28 | The new address of the modified CCW TIC instruction is not accessible to you. |
| 32 | The new address of the modified CCW TIC instruction does not fall on a doubleword boundary. |
| 36 | DEVICE END and CHANNEL END have already occurred. |
| 44 | The modified CCW cannot be a NOP instruction with command chaining if it is the last CCW in a real channel control program. |
| 56 | Since the I/O is queued, there is no reason to issue a GENIO MODIFY instruction. |
| 60 | No CCW address was specified. |

**For the START or STARTR functions:**

| Return Code | Meaning |
|---|---|
| 4 | Your virtual machine is not authorized for real I/O. |

| Return Code | Meaning |
|---|---|
| 8 | The specified I/O device is not open. |
| 12 | The specified I/O device is busy. |
| 16 | Channel control word (CCW) address was not specified. |
| 20 | You cannot perform real I/O functions while in key 0. |
| 24 | A real I/O device is required for the STARTR function. |

**For the HALT function:**

| Return Code | Meaning |
|---|---|
| 8 | The specified I/O device is not open. |
| 12 | The I/O activities of the device could not be halted. |
| 16 | The specified device is not operational. |

**Abend codes for all functions:**

| Abend Code | Reason Code | Meaning |
|---|---|---|
| FCA | 0500 | The specified parameter list is invalid. |
| FCA | 0501 | Your task is not authorized to perform real I/O functions. |

## The List Format

```
[label]      GENIO      ┌                                   ┐
                        │OPEN[,EXIT=exit][,UWORD=uword]     │,MF=L
                        │CLOSE                              │
                        │CHAR[,DATA=address]                │
                        │MODIFY[,CCW=address]               │
                        │START[,CCW=address]                │
                        │STARTR[,CCW=address]               │
                        │HALT                               │
                        └                                   ┘
```

This format of the macro instruction 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 parameters listed above are valid in the list format of this instruction.

### Added Parameter

**MF=L**
Specifies the list format of this macro instruction.

## The List Address Format

```
[label]    GENIO    ┌                                      ┐
                    │OPEN[,EXIT=exit][,UWORD=uword]        │,MF=(L,address[,label])
                    │                                      │
                    │CLOSE                                 │
                    │                                      │
                    │CHAR[,DATA=address]                   │
                    │                                      │
                    │MODIFY[,CCW=address]                  │
                    │                                      │
                    │START[,CCW=address]                   │
                    │                                      │
                    │STARTR[,CCW=address]                  │
                    │                                      │
                    │HALT                                  │
                    │                                      │
                    │,DEV=dev                              │
                    └                                      ┘
```

This format of the macro instruction 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 issue the instruction using this format, then you must do so before any related invocation of the instruction using the execute format.

### Added Parameter

**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 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.

## The Execute Format

```
[label]     GENIO     ┌                                   ┐
                      │OPEN[,EXIT=exit][,UWORD=uword]     │,MF=(E,address)
                      │CLOSE                              │
                      │CHAR[,DATA=address]                │
                      │MODIFY[,CCW=address]               │
                      │START[,CCW=address]                │
                      │STARTR[,CCW=ccw]                   │
                      │HALT                               │
                      │,DEV=dev[,ERROR=addr]              │
                      └                                   ┘
```

This format of the macro instruction generates code that executes the function, using a parameter list whose address you specify.

Note that only the parameters listed above are valid in the execute format of this instruction.

### Added Parameter

**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.

# Chapter 9. Authorized GCS Service Macros

## AUTHNAME

### 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.[9] 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.

The AUTHNAME macro instruction 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 in 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.

The AUTHNAME macro instruction is available in standard, list, list address, and execute formats.

The ***standard*** format of the AUTHNAME macro instruction is:

```
[label]     AUTHNAME     {SET,EP=addr[,UWORD=addr]} ,NAME= {'name'  } [,ERROR=addr]
                         {CLR                     }        {register}
```

## Parameters

**SET**

Indicates that a control block is to be created for the authorized program in question.

Once this is done, the unauthorized program will be able to 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 via the CONTENTS macro instruction. If necessary, review the entry titled "CONTENTS" on page 427.

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).

[9] In this context, an "authorized program" is one running in supervisor state. An "unauthorized program" is one running in problem state.

**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 label 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 zeroes.

**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, surrounded by single quotation marks. 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 Notes

- The authorized program is always loaded at GCS initialization time. It is possible for one virtual machine to invoke this program after another machine has cleared it. This is due to the time lag between issuing the CLR function and completing it. The authorized program should be designed with this in mind. If necessary, review the entry titled "AUTHCALL" on page 172.
- It is impossible for an unauthorized program to call an authorized program via 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.
- Generally, the AUTHNAME macro is issued by an authorized program (RSCS or VTAM, for example) on behalf of unauthorized programs.

- The authorized program called by the unauthorized program (via AUTHCALL) will have the same PSW key as the program that issued the corresponding AUTHNAME instruction.
- The AUTHNAME macro places the control block for an authorized program in common storage. Hence, any unauthorized application in the group can call it.

## Examples

In the first example, an AUTHNAME macro is issued to make an authorized program available to unauthorized programs.

```
AUTHNAME SET,NAME=BLUE,EP=(3)
```

The authorized program will be known to the unauthorized programs as BLUE. The address of this authorized program is in register 3.

In the second example, an AUTHNAME macro was issued to make an authorized program available to unauthorized programs.

```
AUTHNAME SET,NAME=RED,EP=PURPLE,ERROR=REDERR
```

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. There are no abend codes.

| Return Code | Meaning |
|---|---|
| 0 | Request was completed normally. |
| 4 | A control block already exists for this authorized program. |
| 8 | The address you specified for the EP parameter is not in a shared segment. |
| 24 | Parameter list is invalid. |
| 44 | No authorized program has the name you specified. |

## The List Format

| | | |
|---|---|---|
| [label] | AUTHNAME | ⎡SET,[EP=addr] [,UWORD=addr]⎤ [,NAME='name']<br>⎣CLR⎦<br>,MF=L |

This format of the macro instruction 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 parameters listed above are valid in the list format of this instruction.

## Added Parameter

**MF=L**
Specifies the list format of this macro instruction.

## The List Address Format

| | | |
|---|---|---|
| [label] | AUTHNAME | ┌ SET,[EP=addr][,UWORD=addr] ┐ ,NAME='name'<br>└ CLR ┘<br>,MF=(L,address[,label]) |

This format of the macro instruction 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 issue the instruction using this format, then you must do so before any related invocation of the instruction using the execute format.

Note that only the parameters listed above are valid in the list address format of this instruction.

## Added Parameter

**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 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.

## The Execute Format

| | | |
|---|---|---|
| [label] | AUTHNAME | ┌ SET,EP=addr[,UWORD=addr] ┐ ,NAME='name'[,ERROR=addr]<br>└ CLR ┘<br>,MF=(E,address) |

This format of the macro instruction generates code that executes the function using a parameter list whose address you specify.

Note that only the parameters listed above are valid in the list address format of this instruction.

## Added Parameter

**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.

# LOCKWD

## 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. Likewise 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, etc.

The LOCKWD macro instruction helps you to manage this competition. It can allow a virtual machine to acquire exclusive use of common storage while it accesses, and possibly modifies, the data therein. Likewise it can allow 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 instruction to release its lock so others can use the resource.

The LOCKWD macro instruction is an authorized GCS function.

The format of the LOCKWD macro instruction is:

```
[label]     LOCKWD     { ACQUIRE } ,LOCK= { LOCAL  }
                       { RELEASE }        { COMMON }
                       { TEST    }
```

## Parameters

**ACQUIRE**
Indicates that the virtual machine or task wants to establish the lock specified in the instruction.

**RELEASE**
Indicates that the virtual machine or task wants to give up the lock it acquired previously. That lock is specified in the instruction.

**TEST**
Indicates that the virtual machine or task wants to know if it holds a lock on common storage.

This parameter 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 Notes

- 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.
- The supervisor acquires and releases locks on behalf of a virtual machine or task.
- 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.
- 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.
- There are two ways to release a lock:
  1. A virtual machine or task explicitly reissues the LOCKWD macro instruction with the RELEASE parameter and lock properly specified.
  2. A virtual machine or task that is holding a lock terminates.
- The LOCKWD macro instruction can help manage the natural competition for storage access among virtual machines and among tasks. But to realize only this would be to ignore LOCKWD's richer ability to coordinate activity among virtual machines and among tasks.
- Often an authorized program will be called to perform work on behalf of an unauthorized program. Usually the authorized program runs in a different key from the unauthorized program. In such cases, the LOCKWD macro instruction is required before the authorized program issues the VALIDATE macro instruction. Review the entry titled "VALIDATE" on page 216.
- 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 and to no one else.
- 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

In the following example, a task requests a lock on common storage. Presumably, the task has already acquired a lock on its own local resources.

```
LOCKWD ACQUIRE,LOCK=COMMON
```

In the second example, the task wants to know if it holds the lock on common storage.

```
LOCKWD TEST,LOCK=COMMON
```

## Return Codes and Abend Codes

When this macro completes processing, it passes to the caller a return code in register 15.

**General Return Codes:**

| Return Code | Meaning |
|---|---|
| 00 | The lock was successfully acquired or released. |
| 04 | 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:**

| Return Code | Meaning |
|---|---|
| 0 | The lock is free. |
| 4 | Your machine and task hold the lock on common storage. |
| 8 | Another machine and task hold the lock on common storage. |

**Abend codes for all functions:**

| 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

### 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 resets.[10]

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.

Use the MACHEXIT macro instruction to declare or cancel a machine termination exit routine for an entire virtual machine group.

The MACHEXIT macro instruction is an authorized GCS function.

The MACHEXIT macro instruction is available in standard, list, list address, and execute formats.

The ***standard*** format of the MACHEXIT macro instruction is:

| | | |
|---|---|---|
| [label] | MACHEXIT | ⎧SET,EP=address[,UWORD=address]⎫ ,NAME= ⎧name⎫ [,ERROR=addr]<br>⎩CLR⎭ ⎩reg⎭ |

### Parameters

**SET**
Indicates that you are declaring a machine termination exit routine for your virtual machine group.

**CLR**
Indicates that you are cancelling 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 cancelled by the same machine that declared it.

[10] 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, SET ECMODE, and DEFINE CHANNELS. A virtual machine also resets when its GCS supervisor terminates abnormally or when it issues the IUCV SEVER or IUCV RETRIEVE BUFFER instruction. It may also be forced to reset by the CP operator.

**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 via the CONTENTS macro instruction. If necessary, review the entry titled "CONTENTS" on page 427.

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. Hence, this option provides a way for any authorized machine to refer to the exit, as, for example, when clearing it.

Note that the name for the routine is declared by the authorized machine that declares the exit routine in the first place. That machine must supply both the name and the address of the routine being 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 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).

## Usage Notes

- Only an authorized virtual machine can issue the MACHEXIT macro instruction.
- 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 that of the virtual machine that declared it, and it always runs in supervisor state.
- 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 in which they were declared.
- 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 cancelled.
- In a typical scenario, a machine termination exit routine may be scheduled for execution when one virtual machine resets and later be cancelled by another virtual machine. Nevertheless, the routine would still run because it has already been scheduled. You should take this into account when designing your over-all processing procedure.
- No machine termination exit routine can receive control via the AUTHCALL macro instruction. Such a routine receives control only if it is properly declared via the MACHEXIT instruction and if some virtual machine within the group resets.
- When the machine termination exit routine receives control, its registers contain the following.

| Register | Contents |
|---|---|
| 0 | Bits 0 - 15: The machine ID of the virtual machine that was reset.<br><br>Bits 16 - 31: Reserved. |
| 1 | The UWORD parameter specified in the MACHEXIT instruction 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. There are no abend codes.

| Return Code | Meaning |
|---|---|
| 0 | Function completed successfully. |
| 4 | The specified machine termination exit routine has already been declared. |
| 8 | The specified machine termination exit routine is not in common storage. |
| 24 | Invalid parameter list. |
| 44 | The name of the machine termination exit routine that you want to cancel could not be found. |

## The List Format

```
[label]    MACHEXIT    MF=L ┌,SET[,EP=address]  [,UWORD]┐ ┌,NAME= {name}┐
                            └,CLR                       ┘ └       {reg } ┘
```

This format of the macro instruction 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 parameters listed above are valid in the list format of this instruction.

### Added Parameter

**MF = L**
Specifies the list format of this macro instruction.

## The List Address Format

```
[label]    MACHEXIT    MF=(L,address[,label])

                            ┌,SET[,EP=addr]  [,UWORD]┐ ┌,NAME= {name}┐
                            └,CLR                    ┘ └       {reg } ┘
```

This format of the macro instruction 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 issue the instruction using this format, then you must do so before any related invocation of the instruction using the execute format.

**Note:** Only the parameters listed above are valid in the list address format of this instruction.

### Added Parameter

**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 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.

## The Execute Format

| | | |
|---|---|---|
| [label] | MACHEXIT | MF=(E,address)<br>[,SET[,EP=addr] [,UWORD] \| ,CLR] [,NAME= {name \| reg}] [,ERROR=addr] |

This format of the macro instruction generates code that executes the function using a parameter list whose address you specify.

### Added Parameter

**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.

# PGLOCK

## 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 instruction locks a specified page of your virtual storage into real storage. This makes the page ineligible for page-out.

The PGLOCK macro instruction is an authorized GCS function.

The format of the PGLOCK macro instruction is:

| | | |
|---|---|---|
| [label] | PGLOCK | (reg) |

## Parameter

**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 Notes

- The task that issues the PGLOCK macro instruction must be running in supervisor state. Moreover, the DIAG98 parameter must be specified in the OPTION control statement in the virtual machine's directory entry.
- Use of the PGLOCK macro instruction can enhance your program's efficiency by making the CP virtual-to-real translation step unnecessary. Moreover, 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.

  A virtual machine must be authorized to issue the DIAGNOSE X'98' code. This authorization is granted by specifying DIAG98 in the directory entry of the virtual machine (OPTION statement). If the machine is not authorized for DIAGNOSE X'98', a return code is passed to the program issuing the PGLOCK macro. Refer to *VM/XA SP CP Programming Services* for more detailed information on the DIAGNOSE X'98'.
- The PGLOCK macro returns the real address of the locked page in register 1.
- The page that contains the address you specify will be locked into real storage.
- There are two ways for a page locked by the PGLOCK macro to be unlocked:
  - The task that issued the PGLOCK macro instruction terminates.
  - A task explicitly issues the PGULOCK macro instruction, correctly specifying the virtual address of the page to be unlocked.
- A supervisor state program often must build a channel control program in real storage. When it does, it should use the PGLOCK instruction to lock into real storage the page in which it is building the channel control program. If necessary, review the entry titled "GENIO" on page 181.

- 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 accommodate 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 via the GENIO macro instruction. If necessary, review the entry titled "GENIO" on page 181.

## 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. The possible meanings of the return code are as follows:

| **Return Code** | **Meaning** |
|---|---|
| 00 | Function completed successfully. |
| 04 | The user is running V=R. |
| 08 | The virtual address of the page in question is invalid. |
| 12 | GCS is unable to lock the specified page. No real page frames are available. |
| 16 | The specified page is already locked. |
| 20 | The virtual machine issuing this instruction is not authorized to perform any real I/O operations. |

# PGULOCK

## Unlock a Certain Page of Virtual Storage that was Locked in Real Storage Using PGLOCK

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 instruction unlocks a certain page of virtual storage that was previously locked in real storage using the PGLOCK macro instruction. Unlocking such a page makes it eligible for page-out once again.

The PGULOCK macro instruction is an authorized GCS function.

The format of the PGULOCK macro instruction is:

| | | |
|---|---|---|
| [label] | PGULOCK | (reg) |

## Parameter

**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 Notes

- The task that issues the PGULOCK macro instruction must be running in supervisor state. Moreover, the DIAG98 parameter must be in the OPTION control statement in the virtual machine's directory entry.
- If a PGULOCK macro instruction is not issued for a page that is locked, then the page is automatically unlocked when the task that locked it terminates.
- 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. The possible meanings of this code are as follows:

| Return Code | Meaning |
|---|---|
| 00 | Function completed successfully. |
| 04 | The user is running V=R. |
| 08 | The virtual address of the page in question is invalid. |
| 12 | The specified page is not locked. |
| 16 | The virtual machine issuing this instruction is not authorized to perform any real I/O operations. |

## SCHEDEX

### 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 instruction 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 instruction is an authorized GCS function.

The format of the SCHEDEX macro instruction is:

| | | |
|---|---|---|
| [label] | SCHEDEX | ID=id,EXIT=exit[,UWORD=addr] |

### 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. Once 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.

## Usage Notes

- 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.
- The use of SCHEDEX certainly 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 in 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.
- The task issuing the SCHEDEX macro instruction resumes normal 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.
- Any exit routine scheduled via the SCHEDEX instruction runs in key 0.
- A zero return code from the SCHEDEX macro does not necessarily mean that the exit has been scheduled. What it does mean is that the request has been sent to CP. If the virtual machine wherein the exit resides is part of the group, then the exit will be scheduled.

## Example

In the following example, the SCHEDEX macro instruction schedules 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.

```
         SCHEDEX ID=IDENT,EXIT=(3)
         .
         .
         .
IDENT    DC  H'2'
         DC  H'4'
```

## Input to the Exit Program

The program to which an exit is scheduled receives the following information in its registers.

| Register | Contents |
|---|---|
| Register 1 | The user word (UWORD) specified in the SCHEDEX instruction. |
| Register 13 | The address of the register save area. |
| Register 14 | The address to which control is to return once the exit program completes execution. |
| Register 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.

| Return Code | Meaning |
|---|---|
| 0 | Your request has been sent to CP. |
| 4 | The virtual machine identifier that you specified was invalid. |
| 8 | The address of the exit routine you specified is not in a shared segment. |
| 12 | The task identifier is invalid. This return code is meaningful only if the exit is scheduled to run on your virtual machine. |

| Abend Code | Reason Code | Meaning |
|---|---|---|
| FCB | 0A01 | Insufficient storage was available to satisfy a GETMAIN instruction that the system issued. |

## TASKEXIT

### 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.

Use the TASKEXIT macro instruction to declare a task termination exit routine for an entire virtual machine group.

The TASKEXIT macro instruction is an authorized GCS function.

The TASKEXIT macro instruction is available in standard, list, list address, and execute formats.

The ***standard*** format of the TASKEXIT macro instruction is:

| | | |
|---|---|---|
| [label] | TASKEXIT | {SET,EP=addr[,UWORD=addr] \| CLR} ,NAME= {name \| register} [,ERROR=addr] |

### 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 cancelled.

**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 via the CONTENTS macro instruction. If necessary, review the entry titled "CONTENTS" on page 427.

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 instruction 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 instruction:

- 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 instruction, just as it would were there no error.

## Usage Notes

- Only an authorized user can issue the TASKEXIT macro instruction.
- The exit routine that you define via the TASKEXIT instruction must reside in a shared segment.
- Remember that the identifier you specify in the NAME parameter is strictly for your benefit and that of the TASKEXIT macro. To specify the SET and NAME parameters together is, in effect, to "declare" the name that is to be associated with the exit routine in question.
- You can declare more than one task termination exit routine for your virtual machine group. However, since the TASKEXIT instruction can declare only one exit routine at a time, you will have to issue it more than once. Each exit routine that you declare will run when a task in your virtual machine group terminates. However, the order in which they will run is unpredictable.
- GCS associates the PSW key and the enable flags of the task that issues the TASKEXIT instruction with those of the task termination exit routine.
- 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.
- 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 via the ESATE macro instruction that will run if the task terminates abnormally.

  Should this be the case, and the task terminates, GCS sees to it that the task's exit routines are run first. Afterward the task termination exit routine is executed.
- When the task that declared the task termination exit routine terminates, then the latter executes one last time. After that, it disappears.

- When the task termination exit routine gains control, its registers contain the following:

| Register | Contents |
|---|---|
| 0 | The high-order two bytes contain the virtual machine id in which the terminated task was running. The low-order bytes contain the task id. |
| 1 | The UWORD. |
| 13 | Address of the register save area. |
| 14 | Return address within the GCS supervisor. |
| 15 | The address of the task termination exit routine. |

## Example

In the following example, an authorized member of a virtual machine group wants to define a task termination exit routine for its entire group.

```
DCLTE TASKEXIT SET,EP=(4),NAME=TE6,ERROR=(7)
```

The entry point of this routine is at the address in register 4. Since 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. There are no abend codes.

| Return Code | Meaning |
|---|---|
| 0 | Function completed successfully. |
| 4 | This task termination exit routine has already been declared for this virtual machine group. |
| 8 | The address you specified for the task termination exit routine is not in a shared segment. |
| 24 | Invalid parameter list. |
| 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. |

## The List Format

```
[label]     TASKEXIT    ┌                              ┐  ┌                   ┐
                        │SET,EP=addr[,UWORD=addr]      │  │,NAME= { name     }│,MF=L
                        │                              │  │       { register }│
                        │CLR                           │  └                   ┘
                        └                              ┘
```

This format of the macro instruction 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 parameters listed above are valid in the list format of this instruction.

### Added Parameter

**MF = L**
Specifies the list format of this macro instruction.

## The List Address Format

```
[label]     TASKEXIT    ┌                              ┐  ┌                   ┐
                        │SET,EP=addr[,UWORD=addr]      │  │,NAME= { name     }│
                        │                              │  │       { register }│
                        │CLR                           │  └                   ┘
                        └                              ┘
                        ,MF=(L,address,[label])
```

This format of the macro instruction 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 issue the instruction using this format, then you must do so before any related invocation of the instruction using the execute format.

Note that only the parameters listed above are valid in the list address format of this instruction.

### Added Parameter

**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 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.

## The Execute Format

```
[label]      TASKEXIT     ┌                         ┐  ┌              ┐
                          │SET,EP=addr[,UWORD=addr] │  │,NAME= ⎧name    ⎫│
                          │                         │  │       ⎩register⎭│
                          │CLR                      │  └              ┘
                          └                         ┘
                          ,MF=(E,address)
```

This format of the macro instruction generates code that executes the function using a parameter list whose address you specify.

Note that only the parameters listed above are valid in the execute format of this instruction.

### Added Parameter

**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.

# VALIDATE

## By Comparing Keys, Confirm that a Virtual Machine, Program, etc., 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 4-kilobyte 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.

What's more, 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 can be denied either type of access.

The VALIDATE macro instruction 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 instruction is an authorized GCS instruction.

The format of the VALIDATE macro instruction is:

| | | |
|---|---|---|
| [label] | VALIDATE | ADDR=addr[,KEY=key][,LENGTH=length] |

## 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 four 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 Notes

- The VALIDATE instruction 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.
- The supervisor determines whether the area of storage in question is addressable. If it is, then the key specified in the VALIDATE macro instruction 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.
- If the key of the storage area matches the key specified in the VALIDATE macro instruction, or if the program is running in key 0, then store access to the area is possible.
- 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.
- 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.
- Authorized programs often are asked to perform work on behalf of unauthorized programs. Before an authorized program accesses an area of storage on behalf of 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 instruction. In addition, system routines frequently use the VALIDATE instruction to accomplish much the same thing.
- Before an authorized program issues the VALIDATE instruction for shared storage, it should place a lock on the storage in question via the LOCKWD instruction. This is required to prevent the key of the storage from changing. Review the entry titled "LOCKWD" on page 196.

## Examples

In the first example, the first VALIDATE macro instruction is issued to confirm that the address is accessible by a program running in key 14.

```
         VALIDATE ADDR=ADDRESS,KEY=KEY1
         .
         .
         .
ADDRESS  DS   F'5672'
KEY1     DS   X'E0'
```

In the second example, a VALIDATE macro instruction was issued to 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.

```
VALIDATE ADDR=(6),KEY=(7),LENGTH=(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.

| Return Code | Meaning |
|---|---|
| 0 | The key of the storage area matches the key specified in the macro instruction or, if none was specified, the key of the program that issued the instruction. |
| 4 | The keys do not match but the storage area has no fetch protection. Therefore, fetch access is possible. |
| 8 | The keys do not match and the storage area has fetch protection. Therefore, no type of access is possible. |
| 12 | The storage area in question is not addressable. |
| 16 | The specified length of the storage area is less than 0 or greater than $2^{24}$-1 bytes. |

# Chapter 10. Storage Management Service Macros

# FREEMAIN

## 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.

Use the FREEMAIN macro instruction to free a contiguous block of storage.

The FREEMAIN macro instruction is available in standard, list, and execute formats.

The *standard* format of the FREEMAIN macro instruction is:

```
[label]     FREEMAIN    ⎧RC,LV=length,A=address⎫
                        ⎪RU,LV=length,A=address⎪
                        ⎪R,LV=length,A=address ⎪ [,SP=number]
                        ⎨E,LV=length,A=address ⎬
                        ⎪EU,LV=length,A=address⎪
                        ⎪V,A=address           ⎪
                        ⎩VU,A=address          ⎭
```

## 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.

**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 three 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. In this case 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 instruction.

For a definition of all subpool numbers, review the section on the SP parameter in the entry titled "GETMAIN" on page 225.

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 three low-order bytes must contain the length of the storage block.

## Examples

In the first example, the task requests that 400 bytes of storage in subpool 10 be freed.

```
FREEMAIN RC,LV=400,A=(2),SP=10
```

Register 2 contains the address of the storage block. Since this is a conditional request, a return code of 0 would result if the storage were indeed freed. If it were not, then a return code of 4 would result and the storage in question would remain unchanged.

In the second example, the task requested a variable amount of storage within a certain range.

```
GETMAIN VC LA=RANGE,A=DBLWD
.
.
.
FREEMAIN V,A=DBLWD
```

The range was specified in the 2-word list at the address associated with the label RANGE. The task provided a 2-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 an unconditional request, it passes to the caller a return code in register 15.

**For the RC parameter only:**

| Return Code | Meaning |
|---|---|
| 0 | Function completed successfully. |
| 4 | Function was not completed. |

| Abend Code | Meaning |
|---|---|
| 305 | A FREEMAIN macro instruction contained a subpool specification error. |
| 605 | Either a FREEMAIN macro instruction contained an invalid address in the A parameter or an invalid parameter list address was passed to the macro. |
| 705 | An unrecoverable machine, system, or other error occurred while processing the FREEMAIN macro. |
| 905 | The address of the storage area specified in a FREEMAIN macro instruction 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 via the PGLOCK macro instruction. |
| D05 | One of several things happened: • The FREEMAIN macro attempted to free an area of storage not allocated to your task. • Or, you specified zero or a negative number in the LV parameter. • Or, the key is different from what it was when the storage was allocated. |
| E05 | You specified a parameter that GCS does not support. |
| 30A | A FREEMAIN macro instruction, with the R parameter specified, contained a subpool specification error. |
| 70A | An unrecoverable machine, system, or other error occurred while processing the FREEMAIN macro with the R parameter specified. |

| Abend Code | Meaning |
|---|---|
| 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 via the PGLOCK instruction and never unlocked. |
| D0A | One of several things happened: <br>• The FREEMAIN macro, with the R parameter specified, attempted to free an area of storage not allocated to your task. <br>• Or, you specified zero or a negative number in the LV parameter. <br>• Or, the key is different from what it was when the storage was allocated. |
| 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 unrecoverable 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 instruction, with the RU parameter specified, was not on a doubleword boundary. |
| A78 | The area to be freed by the FREEMAIN instruction, with the RU parameter specified, overlapped a free area of storage or is an area that was locked via the PGLOCK instruction. |
| D78 | One of several things happened: <br>• The FREEMAIN macro, with the RU parameter specified, attempted to free an area of storage not allocated to your task. <br>• Or, you specified zero or a negative number in the LV parameter. <br>• Or, the key is different from what it was when the storage was allocated. |
| E78 | A FREEMAIN instruction, with the RU parameter specified, specified another parameter that is not supported by GCS. |

## The List Format

```
[label]      FREEMAIN    ┌                ┐
                         │E[,LV=length]   │[,A=address][,SP=number],MF=L
                         │V               │
                         └                ┘
```

This format of the macro instruction 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 parameters listed above are valid in the list format of this instruction.

### Added Parameter

**MF=L**
Specifies the list format of this macro instruction.

## The Execute Format

```
[label]      FREEMAIN    ┌                ┐
                         │E[,LV=length]   │[,A=address][,SP=number],MF=(E,address)
                         │V               │
                         └                ┘
```

This format of the macro instruction generates code that executes the function, using a parameter list whose address you specify.

Note that only the parameters listed above are valid in the execute format of this instruction.

### Added Parameter

**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.

# GETMAIN

## Obtain a Contiguous Block of Storage

The storage management function of GCS enables a task to dynamically obtain and free contiguous blocks of virtual storage as required.

Use the GETMAIN macro instruction to obtain a contiguous block of virtual storage.

The GETMAIN macro instruction is available in standard, list, and execute formats.

The ***standard*** format of the GETMAIN macro instruction is:

```
[label]  GETMAIN  ⎧RC,LV=length                       ⎫
                  ⎪RU,LV=length                       ⎪
                  ⎪                                   ⎪                 ┌               ┐
                  ⎪R,LV=length                        ⎪ [,SP=number]    │,BNDRY=⎧DBLWD⎫│
                  ⎨                                   ⎬                 │       ⎨     ⎬│
                  ⎪EU,LV=length,A=address             ⎪                 │       ⎩PAGE ⎭│
                  ⎪                                   ⎪                 └               ┘
                  ⎪EU,LV=length,A=address             ⎪
                  ⎪VC,LA=length address,A=address     ⎪
                  ⎩VU,LA=length address,A=address     ⎭
```

## Parameters

**RC**
: Indicates that your register request for storage is conditional. That is, 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.

**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 three 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.

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).

**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 three low-order bytes must contain the length of the requested storage block.

Subpool numbers are defined as follows:

**0** Specifies private, fetch-protected storage. If the main task issued the GETMAIN instruction, then GCS automatically frees the storage when the task terminates. This is also true for a subtask that was attached to a main task with the SZERO=NO parameter specified in an ATTACH macro instruction. Review the entry titled "ATTACH" on page 86.

However, if the subtask were 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. Hence, the storage block is not automatically freed by GCS when the subtask terminates. The storage is freed only when the oldest ancestor task terminates.

Any program can obtain storage from this subpool.

**1 - 127** Specifies private, fetch-protected storage. If the main task issued the GETMAIN instruction, then GCS automatically frees the storage when the task terminates. 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 instruction.

However, if the subtask was attached with this subpool specified in the SHSPV or SHSPL parameter in the ATTACH instruction, 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.

Any program can obtain storage from these subpools.

**229** Specifies private, fetch-protected storage. GCS will automatically free the storage when the task terminates.

Only privileged programs can obtain storage from this subpool.

**230** Specifies private, non-fetch-protected storage. GCS will automatically free the storage when the task terminates.

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 terminates. This storage is persistent in that it must be explicitly freed by some privileged program.

Only privileged programs can obtain storage from this subpool.

**241** Specifies common, non-fetch-protected storage. GCS does not free the storage when the task that acquired it terminates. This storage is persistent in that it 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 terminates. This storage is persistent in that it must be explicitly freed by some privileged program.

Only privileged programs can obtain storage from this subpool.

**244** Specifies private, non-fetch-protected storage. GCS does not free the storage when the task that acquired it terminates. 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 terminate your program abnormally. If your request were conditional, then you will receive a return code of 4.

In summary,

| Subpool | Private | Fetch-protected | Privileged | Persistent |
|---|---|---|---|---|
| 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 |

**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. Indeed, 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 4-kilobyte page boundary.

**DBLWD**
Indicates that the storage block is to begin on a doubleword boundary.

## Usage Note

- GCS sets the key of the requested storage block to the PSW key of the task issuing the GETMAIN instruction.

## Examples

In the following example a task requests a certain amount of storage space.

```
GETMAIN RU,LV=(5),SP=0,BNDRY=PAGE
```

The amount requested has previously been stored in register 5. If the task cannot get the storage, it will not continue processing, since 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.

In the second example a task requests a certain amount of storage space.

```
GETMAIN EC,LV=STORE,A=BLOCK
```

The amount requested 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:**

| Return Code | Meaning |
|---|---|
| 0 | Function completed successfully. |
| 4 | Function was not completed. |

| Abend Code | Meaning |
|---|---|
| 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 unrecoverable 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 instruction contained an error in the specification of the subpool. |
| 70A | An unrecoverable machine, system, or other error occurred while processing the GETMAIN macro with the R parameter specified. |

| **Abend Code** | **Meaning** |
|---|---|
| 80A | Either there was insufficient virtual storage to execute the GETMAIN instruction with the R parameter specified, or a length of zero was specified. |
| B0A | A GETMAIN instruction, with the R parameter specified, contained an error in the specification of the subpool. |
| 778 | An unrecoverable machine, system, or other error occurred while processing the GETMAIN macro instruction with the RU parameter specified. |
| 878 | Either there was insufficient virtual storage to execute the GETMAIN instruction with the RU parameter specified, or the LV parameter specified a zero or a negative number. |
| B78 | A GETMAIN instruction, with the RU parameter specified, contained an error in the specification of the subpool. |
| E04 | A GETMAIN instruction specified a parameter that GCS does not support. |

## The List Format

```
[label]     GETMAIN    ┌                                    ┐
                       │EC[,LV=length][,A=address]          │
                       │                                    │
                       │EU[,LV=length][,A=address]          │
                       │                                    │
                       │VC[,LA=length address][,A=address]  │ [,SP=number]
                       │                                    │
                       │VU[,LA=length address][,A=address]  │
                       └                                    ┘

                       ,MF=L ┌                   ┐
                             │,BNDRY= ┌ DBLWD ┐  │
                             │        └ PAGE  ┘  │
                             └                   ┘
```

This format of the macro instruction 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 parameters listed above are valid in the list format of this instruction.

### Added Parameter

**MF=L**
Specifies the list format of this macro instruction.

## The Execute Format

```
[label]      GETMAIN    ┌                                          ┐
                        │EC[,LV=length][,A=address]                │
                        │                                          │
                        │EU[,LV=length][,A=address]                │
                        │                                          │
                        │VC[,LA=length address][,A=address]       │ [,SP=number]
                        │                                          │
                        │VU[,LA=length address][,A=address]       │
                        └                                          ┘

                        ,MF=(E,address) ┌                  ┐
                                        │,BNDRY= ⎧ DBLWD ⎫ │
                                        │        ⎩ PAGE  ⎭ │
                                        └                  ┘
```

This format of the macro instruction generates code that executes the function, using a parameter list whose address you specify.

Note that only the parameters listed above are valid in the execute format of this instruction.

### Added Parameter

**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.

# Chapter 11. Serviceability Macros

## GTRACE

### 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.

Use the GTRACE macro instruction to record user data in the GCS trace table.

The GTRACE macro instruction is available in standard, list, and execute formats.

The ***standard*** format of the GTRACE macro instruction is:

| | | |
|---|---|---|
| `[label]` | `GTRACE` | `DATA=address,LNG=length,ID=number[,FID=number]` |

### 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 256, as a hexadecimal number from 00 to 100, 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, making them suitable for printing. Since 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. When you are ready to print the data, run TRACERED under CMS. TRACERED will invoke your formatting routine, using the file containing the trace entries as input. For more information on TRACERED, see *VM/XA SP Dump Viewing Facility Operation Guide and 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 |

Since 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.

## Usage Notes

- For the information given to the GTRACE macro to be recorded, you must have previously issued the ETRACE or ITRACE commands. For more information on the ETRACE or ITRACE commands, see "GCS Commands" on page 20.
- The GCS trace table can be displayed on a terminal screen or it can be part of a printed dump but only if the receiver of the dump is authorized to view common storage.

  To identify data recorded by the GTRACE facility, look for X'0E' in the first two bytes of the record. Then, among this data, look in bytes 23 and 24 for the number you specified in the ID parameter.

## Return Codes and Abend Codes

The GTRACE macro generates no abend codes.

When this macro completes processing, it passes to the caller a return code in register 15. The return codes are as follows:

| Return Code | Meaning |
|---|---|
| 00 | Function completed successfully. |
| 04 | The GTRACE facility (monitor class 14) is not enabled. |
| 08 | You specified an invalid value for the LNG parameter. It was less than 1 or greater than 256. |
| 0C | You specified an invalid address for the DATA parameter. |
| 10 | You specified an invalid value for the FID parameter. It was less than 0 or greater than 255. |
| 14 | 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 1 to 80 for the FID parameter. This requires that the value specified for the ID parameter be from 0 to 1023. |
| 1C | Invalid parameter list address. |

## The List Format

| | | |
|---|---|---|
| `[label]` | `GTRACE` | `[DATA=address][,LNG=length][,ID=number][,FID=number],MF=L` |

This format of the macro instruction 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 instruction.

## The Execute Format

| | | |
|---|---|---|
| `[label]` | `GTRACE` | `[DATA=address][,LNG=length][,ID=number][,FID=number]` `,MF=(E,address)` |

This format of the macro instruction generates code that executes the function, using a parameter list whose address you specify.

## Added Parameter

**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.

# SDUMP

## 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. Refer to the *VM/XA SP CP Diagnosis Reference* for information about the dump viewing facility.

Use the SDUMP macro instruction to produce a dump of part or all of your virtual machine's storage.

The SDUMP macro instruction is available in standard, list, and execute formats.

The ***standard*** format of the SDUMP macro instruction is:

```
[label]       SDUMP      ┌                             ┐┌                       ┐
                         │HDR='dump descriptor'        ││,STORAGE=(start,end)   │
                         │HDRAD=dump descriptor address││,LIST=list address     │
                         └                             ┘└                       ┘
```

## Parameters

**HDR**
Specifies a string of characters that you can use to describe 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.

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 Notes

- If both the STORAGE and LIST parameters are omitted from the SDUMP instruction, then GCS assumes that all virtual storage in the machine is to be recorded in the dump. This includes any discontiguous saved segments the virtual machine may be using.
- It is important to understand the rules governing who receives the spool data 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 instruction 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.

## Examples

In the first example, a dump of the entire virtual machine's storage is requested.

```
DUMPALL SDUMP HDR='ALL MY STORAGE'
```

The character string ALL MY STORAGE is to be placed in the dump for ready identification. The dump is to be sent to the member of the virtual machine group authorized to receive it. If no one is so authorized, then the dump will be sent to the issuer of the instruction. 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.

In the second example, a dump of certain portions of virtual storage is requested.

```
SDUMP HDRAD=(5),LIST=RANGES
```

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 to the caller a return code in register 15.

| Return Code | Meaning |
|---|---|
| 0 | All requested areas have been included in the dump. |
| 4 | Only a portion of the requested areas was included in the dump. |
| 8 | GCS was unable to produce a dump. |

| Abend Code | Reason Code | Meaning |
|---|---|---|
| 233 | 8 | Invalid parameter list address. |

## The List Format

```
[label]     SDUMP      ┌                          ┐┌                       ┐
                       │HDR='char string'         ││,STORAGE=(start,end)   │
                       │HDRAD=char string address ││,LIST=list address     │
                       └                          ┘└                       ┘
                       ,MF=L
```

This format of the macro instruction 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 instruction.

## The Execute Format

```
[label]     SDUMP      ┌                          ┐┌                       ┐
                       │HDR='char string'         ││,STORAGE=(start,end)   │
                       │HDRAD=char string address ││,LIST=list address     │
                       └                          ┘└                       ┘
                       ,MF=(E,address)
```

This format of the macro instruction generates code that executes the function, using a parameter list whose address you specify.

## Added Parameter

**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.

# Chapter 12. QSAM and BSAM Data Management Service Macros

# CHECK (BSAM)

## Test the Completion of a READ or WRITE Operation

Whenever you issue a READ or WRITE macro instruction, your task needs some way to confirm that the I/O operation completed successfully.

Use the CHECK macro instruction immediately after each READ and WRITE instruction to determine if and how the I/O operation was completed.

The format of the CHECK macro instruction is:

| | | |
|---|---|---|
| [label] | CHECK | decb address |

## Parameter

**decb address**

Specifies the address of the data event control block (DECB) associated with the READ or WRITE instruction 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 269 and "WRITE (BSAM)" on page 276.

You can write this parameter as an RX-type address or as register (1) through (12).

## Usage Notes

- The CHECK macro tests for errors in the last READ or WRITE operation involving the specified DECB.

  If you issue a READ instruction 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 via the EODAD parameter of the DCB instruction. (If necessary, review the entry "DCB (BSAM/QSAM)" on page 249.)

  If you did not specify an end-of-file exit routine or an error occurred after you issued a WRITE instruction, then GCS will give control to the error analysis routine that you specified via the SYNAD parameter in the DCB instruction. If you failed to specify an error analysis routine, then your task will terminate abnormally.

- For each READ or WRITE instruction you issue you must also issue a CHECK instruction. Moreover, you must issue the CHECK instruction immediately after the READ or WRITE instruction 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.

- GCS does not support the MVS parameter DSORG on this macro instruction. If you include it, then an error will occur.

## Return Codes and Abend Codes

No return codes are generated.

| **Abend Code** | **Meaning** |
|---|---|
| 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 instruction, the data event control block (DECB), or the data control block (DCB). |

# CLOSE (BSAM/QSAM)

## Close a File with which Your Task has Finished

After a task has finished with a particular file, the file must be closed.

Use the CLOSE macro instruction to close a file that your task had previously opened.

The CLOSE macro instruction is available in standard, list, and execute formats.

The ***standard*** format of the CLOSE macro instruction is:

| | | |
|---|---|---|
| [label] | CLOSE | (dcb address[,,dcb address ... ]) |

## Parameter

**dcb address**
Specifies the address of the data control block associated with your file. More specifically, it is the address of the label on the DCB macro instruction associated with your file. If necessary, review the entry titled "DCB (BSAM/QSAM)" on page 249.

More than one file can be closed by a single CLOSE instruction. Note that 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 Notes

- 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 instruction is restored.

  The file is then "logically disconnected" from the main processor.

  Finally, the input or output buffer that GCS set up for the file, when its DCB was opened, is released.

- 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 issue a CHECK macro instruction 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 instruction confirms whether there are any outstanding output events pending for the file in question, as from a WRITE macro instruction. If necessary, review the entries titled "CHECK (BSAM)" on page 244 and "WRITE (BSAM)" on page 276.

- If you have access method control blocks (ACBs) that you wish to close, as well as DCBs, then you can specify a combination of both in the same CLOSE instruction. GCS is able to distinguish the address of one from the address of the other, as long as you separate each with a double-comma.

## 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. |

## The List Format

| | | |
|---|---|---|
| [label] | CLOSE | [([dcb address][,,dcb address ... ])],MF=L |

This format of the macro instruction 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 three 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 instruction must be equal to the maximum length required by an execute format instruction that refers to the same list. A maximum length list can be constructed in one of two ways.

1. Issue the instruction using the list format with the maximum number of parameters required by the execute format of the instruction that refers to the same list.
2. Use an appropriate number of commas in the list format of the instruction 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 instruction in the execute format were filled in by a previous instruction.

### Added Parameter

**MF=L**
Specifies the list format of this macro instruction.

## The Execute Format

| | | |
|---|---|---|
| [label] | CLOSE | [([dcb address][,,dcb address ... ])],MF=(E,address) |

This format of the macro instruction generates code that executes the function using a parameter list whose address you specify.

## Added Parameter

**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.

# DCB (BSAM/QSAM)

## Create a Data Control Block for One of Your Files

For a program to process a file via 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.

Normally the DCB macro instruction 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 instruction provide all the information necessary to the data control block.[11] For more information on the FILEDEF command see "FILEDEF" on page 34.

Use the DCB macro instruction to create a data control block for one of your files. The format of the DCB macro instruction is:

```
[label]        DCB           [BLKSIZE=absolute expression][,DDNAME=label]
                             ,DSORG=PS[,EODAD=address][,EXLST=address]
                             [,LRECL=nnnnn]

                             ,MACRF= ( R               )     [,RECFM= ( VB  )]
                                     | W               |     [        | VBA |]
                                     | RP              |     [        | FB  |]
                                     | WP              |     [        | FBA |]
                                     | GM              |     [        | F   |]
                                     | GL              |     [        { V   }]
                                     { PM              }     [        | FA  |]
                                     | PL              |     [        | VA  |]
                                     | R[P],W[P]       |     [        | U   |]
                                     |                 |     [        ( UA  )]
                                     | [ G{M},P{M} ]   |
                                     ( [  {L}  {L} ]   )

                             [,OPTCD=J][,SYNAD=address]
```

## Parameters

It is required that the DSORG and MACRF parameters be specified in the DCB macro instruction. The other parameters may be supplied in one or more of the following ways.

- Use of the DCB macro instruction.
- Use of the FILEDEF command.
- Given the physical characteristics of the file.

[11] It is possible for the DCB macro instruction to provide all data for the data control block without the help of the FILEDEF command.

- 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 instruction. Review the entry titled "DCBD (BSAM/QSAM)" on page 256.

  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 instruction, since that macro needs those values to process correctly.

**BLKSIZE**
Specifies the maximum block length for the file, in bytes. In the case of 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 four 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 four bytes for the BDW.[12]

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 once it is known by your program. Alternatively, it can be specified in the LENGTH parameter of a READ or WRITE instruction. If necessary, review the entries titled "READ (BSAM)" on page 269 and "WRITE (BSAM)" on page 276.

**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 alphameric characters. The first character must be alphabetic or national.

**DSORG=PS**
Indicates that your file consists of physical sequential records.

Since 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
Specifies the address of your program's exit list. 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 input or output file in the MACRF parameter described below.

When GCS receives a request for input (for example, via a READ macro instruction) and the subsequent CHECK macro instruction 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, described below. If you omit both the EODAD and the

[12] The OPEN macro simulation routine will not accept a BLKSIZE of less than eight.

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. Review the entry titled "OPEN (BSAM/QSAM)" on page 262.

If you specify this parameter, then it is the responsibility of your task to provide and maintain this exit list. Moreover, your task must provide the routines to which it refers. The list must begin on a fullword boundary, with each entry therein comprising a fullword. The basic format of the exit list is:

| 1 Byte | 3 Bytes |
|---|---|
| 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 three bytes. Table 8 shows the meanings of these codes. Note that these are the only codes that have meaning to GCS. Any others are ignored.

Table 8. Exit List Table Codes

| Code | Meaning |
|---|---|
| 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**

In the case of 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.

In the case of 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 four bytes for the record descriptor word (RDW).

It may happen that you omit this parameter in both the FILEDEF command and the DCB instruction. 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 macro instructions 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. Moreover, you are stating what mode of data transmission you will employ in moving data in to or out of the file.

**R**

(BSAM) Specifies that the READ macro instruction will be used. Review the entry titled "READ (BSAM)" on page 269.

**W**

(BSAM) Specifies that the WRITE macro instruction will be used. Review the entry titled "WRITE (BSAM)" on page 276.

**RP**

(BSAM) Specifies that the READ and POINT macro instructions will be used. Review the entry titled "POINT (BSAM)" on page 265.

Specifying the RP parameter gives you the added capability of using the NOTE macro instruction. Review the entry titled "NOTE (BSAM)" on page 260.

**WP**

(BSAM) Specifies that the WRITE and POINT macro instructions will be used.

The WP parameter gives you the added capability of using the NOTE macro instruction.

**GM**

(QSAM) Specifies that the GET macro instruction in MOVE mode will be used. MOVE mode is defined in the entry titled "GET (QSAM)" on page 258.

**GL**

(QSAM) Specifies that the GET macro instruction in LOCATE mode will be used. LOCATE mode is defined in the entry titled "GET (QSAM)" on page 258.

**PM**

(QSAM) Specifies that the PUT macro instruction in MOVE mode will be used. MOVE mode is defined in the entry titled "PUT (QSAM)" on page 267.

**PL**

(QSAM) Specifies that the PUT macro instruction in LOCATE mode will be used. LOCATE mode is defined in the entry titled "PUT (QSAM)" on page 267.

**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 ASA 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 ASA 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 ASA control characters.

**VA**
Indicates that your file is composed of variable-length records that contain ASA 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 ASA 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 unrecoverable I/O error occurs.

Under BSAM, this SYNAD routine receives control when the CHECK macro instruction is issued. Under QSAM, it receives control automatically during the processing of the GET or PUT macro instruction.

If you provide no error routine and an unrecoverable 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 instruction, 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. If necessary, review the entry titled "RETURN" on page 145.

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 9 shows the contents of the registers when your error routine receives control.

Table 9. Contents of the Register when your 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 instruction 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 Notes

- 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 instruction 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 instruction 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 instruction, then the latter will supersede the former.

- Any READ or WRITE macro instruction issued by your program must be tested for completion by the CHECK macro instruction. If necessary, review the entry titled "CHECK (BSAM)" on page 244.
- If you provide a list of exit routine addresses via 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 8 on page 251.
- 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.
- Your SYNAD routine can terminate in one of two ways:
  - It can pass control to another routine in your program. For example, it could pass control to a program that closes the file being processed.
  - It can return 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:

  1. 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 instruction 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, via the SYNADRLS instruction, when they are no longer needed. If necessary, review the entries titled "SYNADRLS (BSAM/QSAM)" on page 274 and "SYNADAF (BSAM/QSAM)" on page 272.
  2. Your SYNAD routine must preserve the contents of registers 13 and 14 as passed to it by GCS. Depending upon 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.
- Note that GCS does not support the MVS parameter LRECL=X on this macro instruction. If you include it, then an error will occur.

## Return Codes and Abend Codes

The DCB macro instruction generates no return codes and no abend codes.

# DCBD (BSAM/QSAM)

## 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 249, there are three ways of assigning a value to a field in a data control block.

- Via the DCB macro instruction.
- Via the FILEDEF command.
- Direct insertion of a parameter's value or attribute into the data control block by your program.

The DCBD macro instruction 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 instruction 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.

The format of the DCBD macro instruction is:

| | | |
|---|---|---|
| [label] | DCBD | [DSORG=[BS][,PS][,QS]] |

## Parameters

**DSORG**

Specifies the type of real data control block for which you want a DSECT created.

Note that data control blocks for BSAM files (BS parameter below) and QSAM files (QS parameter below) are constructed somewhat differently, though they do have fields in common. Note also that the PS parameter, described below, 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 Notes

- 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.
- You can use the same DSECT to insert data into more than one data control block. Just assign another DCB address your base register.
- Since you are the one inserting data into 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 instruction, since that macro needs those values to execute properly.

## Return Codes and Abend Codes

Check the DCBD macro expansion in your source listing for a complete list of the symbolic names and their relative addresses.

The DCBD macro generates no return codes and no abend codes.

## GET (QSAM)

### Obtain the Next Logical Record from a QSAM File

For a record in a QSAM data file to be processed, it must be transferred from its secondary storage device to main storage.

Use the GET macro instruction to obtain the next logical record of a QSAM file for your program to process.

The format of the GET macro instruction is:

| | | |
|---|---|---|
| [label] | GET | dcb address[,area address] |

### 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 via the DCB macro instruction. If necessary, review the entry titled "DCB (BSAM/QSAM)" on page 249.

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 Notes

- 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 via the DCB macro instruction.

  **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 accommodate the largest record that may emerge from the file. If your file comprises variable-length records, then the work area must be large enough to accommodate 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, as you wish.

- GCS assumes that the file being processed has been properly opened using the OPEN macro instruction. If necessary, review the entry titled "OPEN (BSAM/QSAM)" on page 262.

## Return Codes and Abend Codes

The GET macro instruction generates no return codes.

| Abend Code | Meaning |
|---|---|
| 005 | Either an invalid address appears in the GET instruction, or a required address parameter is missing. |

## NOTE (BSAM)

### Get the Relative Position of the Last Block Read or Written in a 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.

Use the NOTE macro instruction to obtain the relative position of the last block you processed in a BSAM file.

The format of the NOTE macro instruction is:

| | | |
|---|---|---|
| [label] | NOTE | dcb address |

### Parameter

**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 via the DCB macro instruction. If necessary, review the entry titled "DCB (BSAM/QSAM)" on page 249.

You can write this parameter as an RX-type address or as register (1) through (12).

### Usage Notes

- Before you issue the NOTE instruction, you must confirm that the last I/O operation was completed successfully. Use the CHECK macro instruction to accomplish this. If necessary, review the entry titled "CHECK (BSAM)" on page 244.
- 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 secondary storage device. The macro stores the record id in the following format:

  **NNNz**

  NNN represents the three-byte file system record number, and z, a byte of zeroes. You must retain this value in a register or in virtual storage for future reference.
- You can use the NOTE and POINT macro instructions on any BSAM file. (If necessary, review the entry titled "POINT (BSAM)" on page 265.) However, you must inform GCS in advance of your intention to do so using the MACRF parameter in the DCB macro instruction.

## Return Codes and Abend Codes

No return codes are generated.

| Abend Code | Meaning |
|---|---|
| 00A | Either you specified an invalid address in the NOTE macro instruction, or an invalid address exists in the data control block associated with your file. |

## OPEN (BSAM/QSAM)

### Prepare a File 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. In general, this process is called "opening the file."

Use the OPEN macro instruction to open a file and prepare it for processing.

The OPEN macro instruction is available in standard, list, and execute format.

The *standard* format of the OPEN macro instruction is as follows:

```
[label]      OPEN                       ┌          ┐
                     (dcb address, │(INPUT)   │, ... )
                                   │(OUTPUT)  │
                                   │(UPDAT)   │
                                   └          ┘
```

### Parameters

**dcb address**
Specifies the address of the data control block associated with the file you want to open. More specifically, it is the address of the label on the DCB macro instruction associated with your file. If necessary, review the entry titled "DCB (BSAM/QSAM)" on page 249.

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 Notes

- Use of the OPEN macro instruction to open a file assumes that the DCB macro instruction has also been issued for that file.
- The OPEN macro prepares your file for processing, then "logically connects" it to your program.

  First, the information you supplied using the DCB macro instruction and the FILEDEF command are merged into one data control block. For more information on the FILEDEF command, see "FILEDEF" on page 34.

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 instruction 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.

- More than one file may be opened by a single OPEN instruction. Just be certain that a comma delimits each entry in the list and that the entire list is surrounded by parentheses.
- When choosing from among the INPUT, OUTPUT, and UPDAT parameters, be mindful of what was specified by the DCB instruction in the MACRF parameter. In this respect, the OPEN and DCB instructions 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 instruction.
- Only the task that opened a file can close it.
- To try to open a file that is already opened, with the same DCB, amounts to issuing a NOP (NO OPERATION) instruction.
- 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.
- If you have access method control blocks (ACBs) that you wish to open, as well as DCBs, then you can specify a combination of both in the same OPEN instruction. GCS is able to distinguish the address of one from the address of the other, as long as 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. |

## The List Format

```
[label]     OPEN      [([dcb address],  ┌(INPUT) ┐,...)],MF=L
                                        │(OUTPUT)│
                                        └(UPDAT) ┘
```

This format of the macro instruction 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 parameters listed above are valid in the list format of this instruction.

The parameter list consists of a one-word entry for each DCB in the parameter list. The high-order byte is reserved while the three 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 instruction must be equal to the maximum length required by an execute format instruction that refers to the same list. A maximum length list can be constructed in one of two ways.

1. Issue the instruction using the list format, with the maximum number of parameters required by the execute format of the instruction that refers to the same list.
2. Use an appropriate number of commas in the list format of the instruction 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 instruction in the execute format were filled in by a previous instruction.

### Added Parameter

**MF=L**
Specifies the list format of this macro instruction.

## The Execute Format

| | | |
|---|---|---|
| [label] | OPEN | [([dcb address], [(<u>INPUT</u>) / (OUTPUT) / (UPDAT)] ,...)],MF=(E, {address / (1)} ) |

This format of the macro instruction generates code that executes the function using a parameter list whose address you specify.

Note that only the parameters listed above are valid in the execute format of this instruction.

### Added Parameter

**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.

## POINT (BSAM)

### Return to a Specified Block within a File

As described in the entry titled "NOTE (BSAM)" on page 260, the NOTE macro instruction 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 instruction to return to one of the locations in a BSAM file that you saved via the NOTE instruction. If you then issue a READ or WRITE macro instruction, it is the block to which you have returned that will be read or written. (If necessary, review the entries titled "READ (BSAM)" on page 269 and "WRITE (BSAM)" on page 276.)

The format of the POINT macro instruction is:

| | | |
|---|---|---|
| [label] | POINT | dcb address,block address |

### 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 via the DCB macro instruction. If necessary, review the entry titled "DCB (BSAM/QSAM)" on page 249.

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 Notes

- Before you issue the POINT instruction, you must confirm that the last I/O operation was completed successfully. Use the CHECK macro instruction to accomplish this. If necessary, review the entry titled "CHECK (BSAM)" on page 244.
- The POINT macro processes no file blocks. It merely positions a pointer to the block that is to be processed next.
- 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 secondary storage device. The macro stores the record id in the following format:

**NNNz**

NNN represents the three-byte file system record number, and z, a byte of zeroes. Presumably you retained this value in a register or in virtual storage.

- 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.
- If you are processing an output BSAM file, then you should issue one last WRITE instruction before you close the file. This ensures that any altered block is written in the file.

## Return Codes and Abend Codes

No return codes are generated.

| Abend Code | Meaning |
|---|---|
| 00A | You specified an invalid address in the POINT macro instruction. |

## PUT (QSAM)

### Place the Next Logical Record in a QSAM File

Use the PUT macro instruction to write the next logical record in a QSAM file.

The format of the PUT macro instruction is:

| | | |
|---|---|---|
| [label] | PUT | dcb address[,area address] |

### 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 via the DCB macro instruction. If necessary, review the entry titled "DCB (BSAM/QSAM)" on page 249.

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. 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 Notes

- 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 using the DCB instruction.

| | |
|---|---|
| **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 secondary 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 issue the PUT instruction, 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 issue the PUT instruction again. |

- GCS assumes that the file being processed has been properly opened via the OPEN macro instruction. If necessary, review the entry titled "OPEN (BSAM/QSAM)" on page 262.

## Return Codes and Abend Codes

The PUT macro instruction 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)

### Using BSAM, Get a Block of Data from a File

When obtaining input from a file, your application is responsible for blocking and unblocking the data.

Use the READ macro instruction to retrieve a block of data from a BSAM disk or reader file and place it into a specified area of your virtual storage.

The READ macro instruction is available in standard, list, and execute formats.

The ***standard*** format of the READ macro instruction is:

| | | |
|---|---|---|
| [label] | READ | decb name,SF,dcb address,area address [,length / ,'S'] |

### 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. For now, suffice to say that the DECB, as it 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 normal, sequential, forward retrieval access method will be employed in obtaining the block from your file.

Since 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 via the DCB macro instruction. If necessary, review the entry titled "DCB (BSAM/QSAM)" on page 249.

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 at which 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 instruction. If necessary, review the entry titled "FILEDEF" on page 34, and the entry titled "DCB (BSAM/QSAM)" on page 249 in this book.

## Usage Notes

- Control may return to your program before the READ macro completes processing. Therefore, you must issue the CHECK macro instruction after each READ instruction to be certain that the latter executed properly. By using the CHECK instruction you confirm whether the input from your file has succeeded, failed, or is incomplete. If necessary, review the entry titled "CHECK (BSAM)" on page 244.
- If you specified the UPDAT parameter in the OPEN instruction that opened your file, then both the READ and WRITE macro instructions must use the same DECB name. If necessary, review the entries titled "OPEN (BSAM/QSAM)" on page 262 and "WRITE (BSAM)" on page 276.

## Input to the READ Macro

The data event control block is created as part of the READ macro expansion. It defines the input "event" using the following format.

| Offset | Contents |
|---|---|
| 0 (0) | ECB |
| 4 (4) | Type of I/O request, thus:<br>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) | Zeroes |

Note that the address of the logical input block is placed in the DECB at 12 (C). It is through this address that you manipulate 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. |

## The List Format

```
[label]     READ
                   decb name,SF[,dcb address][,area address] ┌        ┐,MF=L
                                                              │,length │
                                                              │,'S'    │
                                                              └        ┘
```

This format of the macro instruction 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 instruction.

## The Execute Format

```
[label]     READ
                   decb name,SF[,dcb address][,area address] ┌        ┐,MF=E
                                                              │,length │
                                                              │,'S'    │
                                                              └        ┘
```

This format of the macro instruction generates code that executes the function. The access method uses the DECB whose name you specify as the parameter address.

### Added Parameter

**MF=E**
Specifies the execute format of this macro instruction.

# SYNADAF (BSAM/QSAM)

## 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 terminates abnormally,
- Or, if you have provided one, your SYNAD routine receives control.

A SYNAD routine is a program that you provide to analyze the cause of any permanent I/O error your task encounters. When you define the data control block (DCB) associated with a file, you can also identify a SYNAD routine for that file. If necessary, review the entry titled "DCB (BSAM/QSAM)" on page 249.

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,
- The data event control block (DECB) associated with the I/O "event" that caused the error (this applies only to BSAM files),
- The exceptional condition code,
- The standard status and sense indicators.

Often it is simpler to issue the SYNADAF macro instruction, which will return a message and error code to you describing the I/O error.

The format of the SYNADAF macro instruction is:

| | | |
|---|---|---|
| [label] | SYNADAF | ACSMETH= { BSAM \| QSAM } [,PARM1=(register)][,PARM2=(register)] |

## 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 Notes

- 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:

| Bytes | Contents |
|---|---|
| 0-43 | Blank. You can add your own comments to the message in this field, if you wish. |
| 44-83 | CSISER306S INPUT ERROR nnn ON ddname<br>OR<br>CSISER307S OUTPUT ERROR nnn ON ddname<br><br>nnn specifies an I/O error code. Consult the *VM/XA SP System Messages and Codes Reference* for an explanation of messages CSI306S or CSI307S. ddname specifies the name of the file in question. |
| 84-119 | Blank |

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

No return codes are generated.

| Abend Code | Meaning |
|---|---|
| 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)

### Release the Message Buffer and Save Areas Created by the SYNADAF Macro

When you issue the SYNADAF macro instruction from your SYNAD routine, a message buffer, parameter save area, and register save area are created. This is explained in the entry titled "SYNADAF (BSAM/QSAM)" on page 272.

These storage areas must be released once they are no longer needed. Moreover, the values contained in the registers prior to your issuing the SYNADAF instruction must be restored. Use the SYNADRLS macro instruction to effect this.

The format of the SYNADRLS macro instruction is:

| | | |
|---|---|---|
| [label] | SYNADRLS | |

### Parameters

The SYNADRLS macro instruction accepts no parameters.

### Usage Note

- Before you issue the SYNADRLS instruction, 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 instruction.

  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, since the next area in the chain was just released.

  Everything is then restored to its condition prior to your issuing the SYNADAF instruction.

## 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.

| Return Code | Meaning |
|---|---|
| 00 | Function completed successfully. |
| 08 | 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. |

| Abend Code | Meaning |
|---|---|
| 944 | Either the address of SYNADAF's save area or the pointer to the caller's save area was invalid. |

## WRITE (BSAM)

### Using BSAM, Add or Replace a Block of Data in a File

When using BSAM to place output in a file, your application is responsible for blocking and unblocking the data.

Use the WRITE macro instruction to add or replace a block of data in a disk file, or to add a block to a punch or printer file.

The WRITE macro instruction is available in standard, list, and execute formats.

The ***standard*** format of the WRITE macro instruction is:

| | | |
|---|---|---|
| [label] | WRITE | decb name,SF,dcb address,area address [,length<br>,'S'] |

### 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. For now, suffice to say that the DECB, as it expands within the macro, requires a label which you must supply.

You must write this parameter as an assembler program label.

**SF**
Indicates that a normal, sequential, forward retrieval access method will be employed in placing the block in your BSAM file.

Since 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 via the DCB macro instruction. If necessary, review the entry titled "DCB (BSAM/QSAM)" on page 249.

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.

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 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 instruction. If necessary, review the entry titled "FILEDEF" on page 34, and the entry titled "DCB (BSAM/QSAM)" on page 249 in this book.

## Usage Notes

- Control may return to your program before the WRITE macro completes execution. Therefore, it is required that you issue the CHECK macro instruction after each WRITE instruction to be certain that the latter executed properly. By using the CHECK instruction you confirm whether the output to your file has failed or is incomplete. If necessary, review the entry titled "CHECK (BSAM)" on page 244.
- If you specified the UPDAT parameter in the OPEN macro instruction when you opened your file, then both the READ and WRITE macro instructions must use the same DECB name. If necessary, review the entries titled "OPEN (BSAM/QSAM)" on page 262 and "READ (BSAM)" on page 269.

## Input to the WRITE Macro

The data event control block (DECB) is created as part of the WRITE macro expansion. It defines the output "event" using the following format.

| Offset | Contents |
|---|---|
| 0 (0) | ECB |
| 4 (4) | Type of I/O request, thus:<br>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) | Zeroes |

## Return Codes and Abend Codes

The WRITE macro generates no return codes.

| Abend Code | Meaning |
|---|---|
| 005 | Either you specified an invalid address or an address was missing. |

## The List Format

```
[label]    WRITE    decb name,SF[,dcb address][,area address] ┌,length┐,MF=L
                                                               └,'S'   ┘
```

This format of the macro instruction 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 instruction.

## The Execute Format

```
[label]    WRITE    decb name,SF[,dcb address][,area address] ┌,length┐,MF=E
                                                               └,'S'   ┘
```

This format of the macro instruction generates code that executes the function. The access method uses the DECB whose name you specify.

### Added Parameter

**MF=E**
Specifies the execute format of this macro instruction.

# Chapter 13. VSAM Data Management Service Macros

## Using VSAM under GCS

### VSAM I/O Operations under GCS

GCS applications can access VSAM disks using the VSAM interface. This interface comprises the macro instructions described on the following pages.

This VSAM interface, as supported by GCS, is very similar to that supported by CMS in VM/SP Release 3. Of particular significance is that VSAM disks are in VSE/VSAM format. Moreover, in VM/SP 3, MVS/VSAM requests are mapped to VSE/VSAM requests and executed using VSE/VSAM code.

The macros described on the following pages 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).

Since 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 macro instructions to test and examine the contents of these data structures.

Using the RPL macro instruction, 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.

Keep in mind that, although GCS supports an OS/MVS macro interface, VSAM operations are performed by the VSE/VSAM program product. And, although you can use the macro instructions discussed in this section only for VSAM, many of them are also used for VTAM, QSAM, and BSAM—but, in other environments.

### Control-Block Manipulation Macros

The GENCB, MODCB, SHOWCB, and TESTCB macro instructions are control-block manipulation instructions. The list, list address, and execute formats of these instructions allow you to save virtual storage by using one parameter list for two or more macro invocations. You can also make your program reenterable, that is, executable by more than one task at a time. However, while the generate format of these macros enables you to make programs reenterable, it does not allow shared parameter lists.

## The List Format

The list format of the GENCB, MODCB, SHOWCB, and TESTCB instructions 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, since 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.

## The 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 reenterable, and you must supply the area via the GETMAIN instruction when your program is executed. (If necessary, review the entry titled "GETMAIN" on page 225.) 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.

## The Execute Format

The execute format produces code that executes the function. The execute format is identical to 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 instruction 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.

## The Generate Format

The generate format of these instructions builds the parameter list in a remote area, the address of which you specify, and passes it to VSAM for execution. It allows you to make your program reenterable, 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 reenterable, you should code ADDRESS using register notation. You must obtain this area via the GETMAIN instruction 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.

## Operand Notation for the GENCB, MODCB, SHOWCB, and TESTCB Instructions

The addresses, names, numbers, and options required with parameters in the GENCB, MODCB, SHOWCB, and TESTCB instructions can be expressed in a variety of ways:

- As an absolute numeric expression. For example,

  ```
  COPIES=10
  ```

- As a character string. For example,

  ```
  DDNAME=DATASET
  ```

- As a code or a list of codes separated by commas and enclosed in parentheses. For example,

  ```
  OPTCD=(KEY,DIR,IN)
  ```

- An expression valid for a relocatable A-type address constant, for example,

  ```
  AREA=MYAREA+4
  ```

- As a register from 2 through 12 that contains an address or numeric value. For example,

  ```
  SYNAD=(3)
  ```

  Equated labels can be used to designate a register. For example,

  ```
  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, COPIES, 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 alphameric 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. Moreover, register and S-type address constants cannot be used when you code MF=L.

The tables that follow summarize the manner in which the keyword parameters of VSAM control block manipulation macro instructions can be expressed.

## The GENCB Macro Instruction

| Keyword | Absolute Numeric | 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 |
| *BLK=ACB:* | | | | | | | |
| 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 | |
| PASSWD | | | | x | x | x | x |
| STRNO | x | | | x | x | x | |
| *BLK=EXLST:* | | | | | | | |
| 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 | | | | | |
| *BLK=RPL:* | | | | | | | |
| ACB | | | | x | x | x | x |
| AREA | | | | x | x | x | x |
| AREALEN | x | | | x | x | x | |
| ARG | | | | x | x | x | x |

| 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 | |

## The MODCB Macro Instruction

| 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 |
| *ACB:* | | | | | | | |
| 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 | |
| *EXLST:* | | | | | | | |
| 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 | | | | | |
| *RPL:* | | | | | | | |
| 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 | |
| NXTRPL | | | | x | x | x | x |
| OPTCD | | x | | | | | |
| RECLEN | x | | | x | x | x | |

## The SHOWCB Macro Instruction

| 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 |
| *ACB:* | | | | | | | |
| AREA | | | | x | x | x | x |
| FIELDS | | x | | | | | |
| LENGTH | x | | | x | x | x | |
| OBJECT | | x | | | | | |
| *EXLST:* | | | | | | | |
| AREA | | | | x | x | x | x |
| FIELDS | | x | | | | | |
| LENGTH | x | | | x | x | x | |
| *RPL:* | | | | | | | |
| AREA | | | | x | x | x | x |
| FIELDS | | x | | | | | |
| LENGTH | x | | | x | x | x | |

## The TESTCB Macro Instruction

| 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 |
| *ACB:* | | | | | | | |
| ACBLEN | x | | | x | x | x | |
| ATRB | | x | | | | | |
| AVSPAC | x | | | x | x | x | |
| BSTRNO | 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 | |
| NDELR | 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 | | | | | |

| Keyword | Absolute Numeric | Code | Character String | Register | S-Type Address | Indirect S-Type Address | A-Type Address |
|---|---|---|---|---|---|---|---|
| OFLAGS | | x | | | | | |
| OPENOBJ | | x | | | | | |
| PASSWD | | | | x | x | x | x |
| RKP | x | | | x | x | x | |
| STMST | | | | | | x | |
| STRNO | x | | | x | x | x | |
| ***EXLST:*** | | | | | | | |
| 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 | | | | | |
| ***RPL:*** | | | | | | | |
| 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 | | | | | |
| 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 | |

## Feedback Field Codes

VSAM request macro instructions include ENDREQ, ERASE, GET, POINT, and PUT. After you issue one of these instructions, or the CHECK instruction, register 15 contains a return code that indicates the manner in 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 defined 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 instruction, 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 encountered 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: <br>• An attempt was made to store a record with a duplicate key. <br>• A duplicate record was found for an alternate index with the UNIQUEKEY option. <br>• A record already exists at the accessed record location. |
| 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. |

| Error Code | Meaning |
|---|---|
| 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.<br><br>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:<br><br>• No secondary allocation was specified. Moreover, no space of the required class was available for primary space suballocation on an additional volume (if one was specified).<br><br>• The maximum number of extensions for the file has been exceeded.<br><br>• No space of the required class is available for additional secondary allocations. |
| 32 | An invalid RBA was specified. |
| 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 via the RPL instruction's AREA parameter is not large enough for the requested record. |
| 64 | As many requests are active as the number specified in the STRNO parameter in the ACB instruction. Therefore, another request cannot be started. |

| Error Code | Meaning |
|---|---|
| 68 | The type of accessing for the request does not match the type of accessing in the access method control block. For example,<br>• ADR or CNV was specified, but keyed access is requested.<br>• INPUT was specified, explicitly or by default, but an UPDATE request was made.<br>• GET UPD ADR was requested. However, ADR was not specified on the ACB instruction when the SHAREOPTIONS(4) KSDS was opened. |
| 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 instruction either for an entry-sequenced file (directly or via 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 | A positioning error occurred. The problem program did one of the following things:<br>• It issued a sequential GET instruction without having VSAM positioned first.<br>• It changed from addressed to keyed access without having VSAM positioned for keyed-sequential retrieval.<br>• It issued a sequential PUT instruction for a relative-record file without having VSAM positioned first.<br>• It attempted to improperly switch between forward and backward processing. |
| 92 | You issued a PUT for UPDATE or an ERASE instruction without a preceding GET for UPDATE instruction. |
| 96 | 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.<br>This produced a sequence error during sequential updating. For example, during REPRO REPLACE, two separate updates to the same record were attempted. |
| 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 | You specified invalid or conflicting RPL options or parameters, as follows: <br>• SKP together with BWD. <br>• LRD without BWD. <br>• CNV together with BWD. <br>• ARG parameter was not specified when required. |
| 108 | The RECLEN value specified for the RPL was one of the following: <br>• Larger than the allowed maximum. <br>• Equal to zero. <br>• Smaller than key length plus relative key position. <br>• Not equal to record (slot) size specified for a relative-record file. <br>For alternate index upgrade processing, the alternate index contains too many duplicate keys. Increase the maximum record length to accommodate more keys. |
| 112 | The length of the generic key specified for the RPL is too large or is equal to zero. |
| 116 | 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. |
| 132 | An attempt was made to retrieve a spanned record in LOCATE mode. |
| 136 | An attempt was made to retrieve a spanned record of a keyed-sequenced file with addressed access. |
| 140 | VSAM encountered an inconsistent spanned record, that is, one or more segments were incompletely updated or destroyed. <br>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. <br>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. <br>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. |

| Error Code | Meaning |
|---|---|
| 144 | VSAM encountered a pointer in an alternate index without an associated base record. |
| 148 | The maximum number of pointers in the alternate index has been exceeded. |
| 156 | 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. |
| 192 | VSAM encountered an invalid relative-record number. |
| 196 | An addressed request was issued for a relative-record file. |
| 200 | An addressed or control-interval access was attempted via a path. |
| 204 | The program issued a PUT to insert a record while in backward mode. |

### When the Return Code in Register 15 is 12:

| Error Code | Meaning |
|---|---|
| 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. |

## ACB

### Generate an Access Method Control Block at Assembly Time

An access method control block (ACB) defines certain characteristics of a file that you intend to process via VSE/VSAM. When the file is opened, other characteristics of the file that you defined via the DLBL command are merged with the ACB to complete the picture. For more information on the DLBL command see "GCS Commands" on page 20.

Use the ACB macro instruction to create an ACB and define therein certain characteristics of your file.

This discussion of the ACB macro instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

The format of the ACB macro instruction is shown on the following page.

| | | |
|---|---|---|
| [label] | ACB | [MACRF=( [ADR<br>CNV<br>KEY]<br>[,NDF]<br>[,DIR<br>,SEQ<br>,SKP]<br>[,IN<br>,OUT]<br>{,NRM<br>,AIX}<br>{,NRS<br>,RST}<br>[,NSR]<br>{,NUB<br>,UBF} )]<br>[,BUFND=number] [,BUFNI=number] [,BUFSP=number]<br>[,DDNAME=ddname] [,EXLST=address] [,MAREA=address]<br>[,MLEN=number] [,PASSWD=address] [,STRNO=number] |

## 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. 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, much less process it.

Carefully check the format box above. Note that the processing options are arranged in groups, each with a value that will be assumed by default should you fail to specify from that group. Since they are not positional parameters, they can be specified in any order. Carefully note whether a set of brackets surrounds each individual member of a group, or a set of braces surrounds the entire group. In the former case, you may select more than one of the parameters from the group. In the latter case, you may select only one of them.

**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 instruction will not be deferred for a direct PUT instruction.

**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 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 via the alternate index.

<u>NRS</u>
: 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. If necessary, review the entry titled "OPEN" on page 354. 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 performs 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.

<u>NSR</u>
: Indicates that the resources are not shared.

<u>NUB</u>
: 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 instructions 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 instruction. If necessary, review the entries titled "RPL" on page 361 and "GENCB" on page 327.

**BUFND**
: Specifies the number of I/O buffers to be 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 via 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 accommodate 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 via the BUFSP parameter on the DLBL command that identifies your file. This value overrides the BUFSP parameter in the ACB macro instruction. Likewise, 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. Then, any left-over space is used for data buffers. Finally, if any left-over space remains that is insufficient to accommodate 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 instruction determine the size of the user buffer area. (If necessary, review the entry titled "RPL" on page 361.)

**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 via the MODCB macro instruction. If necessary, review the entry titled "MODCB" on page 336.

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 via the EXLST or GENCB macro instruction. If necessary, review the entries titled "EXLST" on page 311 or "GENCB" on page 323.

If you used the EXLST instruction to create this list, then you can write this parameter as the label on that instruction. If you used the GENCB instruction, 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 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 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 Notes

- Note that the ACB macro creates an access control block at assembly time. Contrast this with the GENCB instruction which generates an ACB at execution time. If necessary, review the entry titled "GENCB" on page 315.
- Be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

## Return Codes and Abend Codes

The ACB macro generates no return codes and no abend codes.

## CHECK

### Suspend Processing While Waiting for the Completion of a VSAM Request

Use the CHECK macro instruction to place your task in the WAIT state while it waits for a certain VSAM request to take place.

The format of the CHECK macro instruction is:

| | | |
|---|---|---|
| [label] | CHECK | RPL=address |

### 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 via the RPL macro instruction. If necessary, review the entry titled "RPL" on page 361.

You can write this parameter as an assembler program label or as register (2) through (12).

### Usage Notes

- VSAM requests are associated with the following macro instructions:
  - ENDREQ
  - ERASE
  - GET
  - POINT
  - and PUT.

  If you specified asynchronous processing (OPTCD=ASY) in the RPL instruction, then issue the CHECK instruction after each of these instructions.

- 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.

- Be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

## 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 via the SHOWCB or TESTCB macro instructions. If necessary, review the entries titled "SHOWCB" on page 375 or "TESTCB" on page 390.

| Return Code | Meaning |
|---|---|
| 0 | Function completed successfully. |
| 4 | Your request was not accepted. The RPL in question is active for another request. |
| 8 | A logical error occurred. |
| 12 | A physical error occurred. |

| Abend Code | Meaning |
|---|---|
| 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

### Logically Disconnect your Program from a VSAM File

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.

Use the CLOSE macro instruction to close a VSAM file that your program has finished with.

This discussion of the CLOSE macro instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

The format of the CLOSE macro instruction is:

| | | |
|---|---|---|
| [label] | CLOSE | (acb address,acb address ...)[,TYPE=T] |

### Parameters

**acb address**
Specifies the address of the access method control block (ACB) associated with the file you wish to close.

Note that 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 normal closing operations performed on the file in question, except that you do not want your program logically disconnected from the file.

### Usage Notes

- 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.
- 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. Hence, if you wish to load records into a file and retrieve records therefrom all in the same run, then you must issue a CLOSE instruction between these two activities.

- 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.
- Under no circumstances will GCS attempt to close ACBs during normal task termination. This is the program's responsibility.
- The parameters in the CLOSE macro instruction are positional. Therefore, write them in the order indicated in the format box above and provide a comma for any that you omit.
- Be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.
- If you have data control blocks (DCBs) that you wish to close, as well as ACBs, then you can specify a combination of both in the same CLOSE instruction. GCS is able to distinguish the address of one from the address of the other, as long as you separate each with a comma. This is a way of saying that, for all practical purposes, this instruction and the one described in the entry titled "CLOSE (BSAM/QSAM)" on page 246 are one and the same. However, note that neither of these instructions, 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.

| Return Code | Meaning |
|---|---|
| 0 | All files were successfully closed. |
| 4 | At least one file was not closed successfully. |
| 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 instruction to display the ERROR field of each access method control block. If necessary, review the entry titled "SHOWCB" on page 367. 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 unrecoverable 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 unrecoverable I/O error occurred while VSAM was completing outstanding I/O requests. |

| 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. |

## ENDREQ

### Terminate a VSAM Request

A VSAM request is associated with one of the following macro instructions: CHECK, ENDREQ, ERASE, GET, POINT, and PUT. (If necessary, review the entry titled "CHECK" on page 302, "ERASE" on page 309, "GET" on page 334, "POINT" on page 357, or "PUT" on page 359). On occasion, you may wish to cancel one such request that you previously made.

Use the ENDREQ macro instruction to cancel a certain VSAM request.

This discussion of the ENDREQ macro instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

The format of the ENDREQ macro instruction is:

| | | |
|---|---|---|
| [label] | ENDREQ | RPL=address |

### Parameter

**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 via the RPL macro instruction. (If necessary, review the entry titled "RPL" on page 361.)

You can write this parameter as an assembler program label or as register (2) through (12).

### Usage Notes

- The ENDREQ macro causes VSAM to end a request. That is, VSAM will forget its position for the specified RPL and will release its associated buffers to another RPL. Therefore, before you issue the ENDREQ instruction specifying an RPL for which the ENDREQ macro was previously executed, you must reposition VSAM.
- Each time you issue the ENDREQ instruction, you must provide the system with a 72-byte save area. Be certain that before you issue the instruction you place the address of this save area in register 13.
- You are limited to as many concurrent active requests as you have specified in the STRNO parameter of the ACB instruction. (If necessary, review the entry titled "ACB" on page 295.) If you want to issue more requests, then you must issue the ENDREQ instruction first.
- If an I/O operation is in progress when you issue the ENDREQ instruction, it will be allowed to complete. This includes operations that are necessary to maintain the integrity of the file.
- 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 instruction 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 will be performed to complete the split and update the index. However, VSAM will then return control to the processing program without adding the second new record.

- 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.
- Be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

## Completion 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 via the SHOWCB or TESTCB macro instructions. If necessary, review the entries titled "SHOWCB" on page 375 or "TESTCB" on page 390.

| Completion Code | Meaning |
|---|---|
| 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. |

| Abend Code | Meaning |
|---|---|
| 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

### Delete a Record from a VSAM File

Use the ERASE macro instruction to delete a record. This record must be one that you have retrieved via the GET macro instruction 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. Likewise, you can delete records in a relative-record file by keyed access, but you cannot delete control intervals. If necessary, review the entry titled "GET" on page 334.

This discussion of the ERASE macro instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

The format of the ERASE macro instruction is:

| | | |
|---|---|---|
| [label] | ERASE | RPL=address |

### Parameter

**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 via the RPL macro instruction. (If necessary, review the entry titled "RPL" on page 361.)

You can write this parameter as an assembler program label or as register (2) through (12).

### Usage Notes

- Each time you issue the ERASE instruction, you must provide the system with a 72-byte save area. Be certain that before you issue the instruction you place the address of this save area in register 13.
- Be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

### 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 via the SHOWCB or TESTCB macro instructions. If necessary, review the entries titled "SHOWCB" on page 375 or "TESTCB" on page 390.

| Return Code | Meaning |
|---|---|
| 0 | Your request was accepted. |
| 4 | Your request was not accepted because the RPL you specified in the ERASE instruction is already active for another request. |
| 8 | A logical error occurred. |
| 12 | A physical error occurred. |

| Abend Code | Meaning |
|---|---|
| 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

### Generate an Exit List at Assembly Time

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 instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

Use the EXLST macro instruction to create a list of the addresses of your exit routines.

The format of the EXLST macro instruction is:

```
[label]     EXLST     ┌                ┌  ┐      ┐
                      │EODAD=(address  │,A│[,L])│
                      │                │,N│     │
                      └                └  ┘     ┘

                      ┌                 ┌  ┐      ┐
                      │,JRNAD=(address  │,A│[,L])│
                      │                 │,N│     │
                      └                 └  ┘     ┘

                      ┌                 ┌  ┐      ┐
                      │,LERAD=(address  │,A│[,L])│
                      │                 │,N│     │
                      └                 └  ┘     ┘

                      ┌                 ┌  ┐      ┐
                      │,SYNAD=(address  │,A│[,L])│
                      │                 │,N│     │
                      └                 └  ┘     ┘
```

### 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 that of an eight-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 Notes

- It is one thing to create a list of exit routine addresses. For them to be of any use to you, however, you must specify the address of this list in the EXLST parameter of the ACB or MODCB instruction. (If necessary, review the entry titled "ACB" on page 295 or "MODCB" on page 336.)

  Since the address of this list is the same as the address of the EXLST instruction in your program, simply refer to the label on this instruction when you create or modify the access method control block (ACB).

- When VSAM enters one of your exit routines, the registers contain information that may be helpful in analyzing the situation.

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 details on the format of this list, consult the manual titled *Using VSE/VSAM Commands and Macros.* |
| 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:

| Register | Contents |
|---|---|
| 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:

| Register | Contents |
|---|---|
| 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. |

- The EXLST instruction generates an exit list at assembly time. Contrast this with the GENCB instruction, which generates an exit list at execution time. If necessary, review the entry titled "GENCB" on page 323.
- You can define no more than 128 exits per GCS virtual machine.
- Be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

## Return Codes and Abend Codes

The EXLST macro generates no return codes and no abend codes.

## GENCB

### Generate an Access Method Control Block at Execution Time

An access method control block (ACB) defines certain characteristics of a file that you intend to process via VSE/VSAM. When the file is opened, other characteristics of the file that you defined via the DLBL command are merged with the ACB to complete the picture. For more information on the DLBL command see "GCS Commands" on page 20.

Use the GENCB macro instruction to create an ACB and define therein certain characteristics of your file.

This discussion of the GENCB macro instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

The format of the GENCB macro instruction is:

```
[label]         GENCB          [AM=VSAM,]
                                          ┌                 ┐
                               BLK=ACB    │,MACRF=( ┌   ┐   │
                                          │         │ADR│   │
                                          │         │CNV│   │
                                          │         │KEY│   │
                                          │         └   ┘   │
                                          │                 │
                                          │        [,NDF]   │
                                          │                 │
                                          │         ┌    ┐  │
                                          │         │,DIR│  │
                                          │         │,SEQ│  │
                                          │         │    │  │
                                          │         │,SKP│  │
                                          │         └    ┘  │
                                          │         ┌    ┐  │
                                          │         │,IN │  │
                                          │         │    │  │
                                          │         │,OUT│  │
                                          │         └    ┘  │
                                          │        {,NRM}   │
                                          │        {,AIX}   │
                                          │                 │
                                          │        {,NRS}   │
                                          │        {,RST}   │
                                          │                 │
                                          │        [,NSR]   │
                                          │                 │
                                          │        {,NUB}   │
                                          │        {,UBF})  │
                                          └                 ┘
                               [,BUFND=number][,BUFNI=number][,BUFSP=number]
                               [,DDNAME=ddname][,EXLST=address][,MAREA=address]
                               [,MLEN=number][,PASSWD=address][,STRNO=number]
                               [,COPIES=number][,LENGTH=number][,WAREA=address]
```

## Parameters

**AM=VSAM**
Indicates that you are using VSAM to process the file associated with the ACB in question.

**BLK=ACB**
Indicates that you wish to generate an access method control block.

This parameter is required to distinguish it from the other two GENCB macro instructions. If necessary, review the entries titled "GENCB" on page 323 and "GENCB" on page 327.

**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, much less process it.

Carefully check the format box above. Note that the processing options are arranged in groups, each with a value that will be assumed by default should you fail to specify from that group. Since they are not positional parameters, they can be specified in any order. Carefully note whether a set of brackets surrounds each individual member of a group, or a set of braces surrounds the entire group. In the former case, you may select more than one of the parameters from the group. In the latter case, you may select only one of them.

**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.

<u>**KEY**</u>

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.

<u>**NDF**</u>

Indicates that any WRITE instruction will not be deferred for a direct PUT instruction.

**DIR**

Indicates direct access to a key-sequenced, entry-sequenced, or relative record file.

<u>**SEQ**</u>

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.

<u>**IN**</u>

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 via 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. If necessary, review the entry titled "OPEN" on page 354. 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 performs 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.

**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 instructions 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 instruction. If necessary, review the entries titled "RPL" on page 361 and "GENCB" on page 327.

**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 largest of the following, by default:

- The buffer size specified in the catalog.

  This buffer size was specified via 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 accommodate 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 via the BUFSP parameter on the DLBL command that identifies your file. This value overrides the BUFSP parameter in the ACB macro instruction. Likewise, 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. Then, any left-over space is used for data buffers. Finally, if any left-over space remains that is insufficient to accommodate 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 instruction determine the size of the user buffer area. (If necessary, review the entry titled "RPL" on page 361.)

**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 via the MODCB macro instruction. If necessary, review the entry titled "MODCB" on page 336.

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 via the EXLST or GENCB macro instruction. If necessary, review the entries titled "EXLST" on page 311 or "GENCB" on page 323.

If you used the EXLST instruction to create this list, then you can write this parameter as the label on that instruction. If you used the GENCB instruction, 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 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 zero, 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 instruction to tailor each ACB to the specific file and type of processing you wish. Review the entry titled "MODCB" on page 336. 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 accommodate the ACB(s) you want to generate. Express this figure in bytes.

**WAREA**

Specifies the address of the area you are supplying in virtual storage to accommodate the ACB(s) 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. Furthermore, 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 Notes

- The GENCB instruction generates an ACB at execution time. Contrast this with the ACB instruction, which generates an ACB at assembly time. If necessary, review the entry titled "ACB" on page 295.
- Each time you issue the GENCB instruction, you must provide the system with a 72-byte save area. Be certain that before you issue the instruction you place the address of this save area in register 13.
- Be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

## Completion Codes, Return Codes and Abend Codes

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 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.

| **Return Code** | **Meaning** |
|---|---|
| 1 | The type of your request is invalid. |
| 2 | The block type is invalid. |
| 3 | One of the keyword codes in the parameter list is invalid. |
| 8 | There is not enough virtual storage to generate the ACB. |
| 9 | The area you specified in the WAREA parameter is not large enough to accommodate the ACB. |
| 14 | You have specified an invalid combination of options in the MACRF parameter. |
| 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. |

## GENCB

### Generate an Exit List at Execution Time

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. Review the entry titled "ACB" on page 295.

This discussion of the GENCB macro instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

Use the GENCB macro instruction to create a list of the addresses of your exit routines.

The format of the GENCB macro instruction is:

| | | |
|---|---|---|
| [label] | GENCB | [AM=VSAM,]BLK=EXLST |
| | | [,EODAD=(address [,A ,N] [,L])] |
| | | [,JRNAD=(address [,A ,N] [,L])] |
| | | [,LERAD=(address [,A ,N] [,L])] |
| | | [,SYNAD=(address [,A ,N] [,L])] |
| | | [,COPIES=number] [,LENGTH=number] [,WAREA=address] |

### Parameters

**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 instruction from the other two GENCB instructions. If necessary, review the entries titled "GENCB" on page 315 and "GENCB" on page 327.

**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 the address of an eight-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 instruction to modify the addresses in any of the exit lists. Review the entry titled "MODCB" on page 344. 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 accommodate 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 accommodate 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. Furthermore, 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 Notes

- Note that the GENCB instruction generates an exit list at execution time. Contrast this with the EXLST instruction which generates an exit list at assembly time. If necessary, review the entry titled "EXLST" on page 311.
- Each time you issue the GENCB instruction, you must provide the system with a 72-byte save area. Be certain that before you issue the instruction you place the address of this save area in register 13.
- Be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

## Completion Codes, Return Codes and Abend Codes

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 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.

| Return Code | Meaning |
|---|---|
| 1 | The type of your request is invalid. |
| 2 | You selected an access method control block. This is invalid. |
| 3 | One of the keyword codes in the parameter list is invalid. |
| 8 | There is not enough virtual storage to generate the exit list. |
| 9 | The area you specified in the WAREA parameter is not large enough to generate the exit list. |
| 10 | You specified an exit without giving an address. |
| 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. |

## GENCB

### Generate a Request Parameter List (RPL) at Execution Time

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. If necessary, review the entry titled "CHECK" on page 302, "ENDREQ" on page 307, "ERASE" on page 309, "GET" on page 334, "POINT" on page 357, or "PUT" on page 359.

Use the GENCB macro instruction to create a request parameter list at execution time describing the characteristics of your VSAM request.

This discussion of the GENCB macro instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

The format of the GENCB macro instruction is:

```
[label]     GENCB       [AM=VSAM,]BLK=RPL[,ACB=address]
                        [,AREA=address][,AREALEN=number][,ARG=address]
                        [,ECB=address][,KEYLEN=number][,NXTRPL=address]
                        ┌                 ┐
                        │,OPTCD=( ┌ADR┐   │
                        │         │CNV│   │
                        │         │KEY│   │
                        │         └   ┘   │
                        │         ┌    ┐  │
                        │         │,DIR│  │
                        │         │,SEQ│  │
                        │         │    │  │
                        │         │,SKP│  │
                        │         └    ┘  │
                        │         ┌    ┐  │
                        │         │,ARD│  │
                        │         │    │  │
                        │         │,LRD│  │
                        │         └    ┘  │
                        │         ┌    ┐  │
                        │         │,FWD│  │
                        │         │    │  │
                        │         │,BWD│  │
                        │         ┌    ┐  │
                        │         │,ASY│  │
                        │         └    ┘  │
                        │         ┌    ┐  │
                        │         │,NSP│  │
                        │         │,NUP│  │
                        │         │    │  │
                        │         │,UPD│  │
                        │         └    ┘  │
                        │         ┌    ┐  │
                        │         │,KEQ│  │
                        │         │    │  │
                        │         │,KGE│  │
                        │         └    ┘  │
                        │         ┌    ┐  │
                        │         │,FKS│  │
                        │         │    │  │
                        │         │,GEN│  │
                        │         └    ┘  │
                        │         ┌    ┐  │
                        │         │,LOC│  │
                        │         │,MVE│) │
                        └         └    ┘  ┘
                        [,RECLEN=number][,COPIES=number]
                        [,LENGTH=number][,WAREA=address]
```

## Parameters

**BLK = RPL**
Indicates that you wish to generate a request parameter list.

This parameter is required to distinguish it from the other two GENCB macro instructions. If necessary, review the entries titled "GENCB" on page 315 and "GENCB" on page 323.

**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 via the ACB macro instruction, 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 via another GENCB macro instruction before issuing this GENCB instruction. If necessary, review the entry titled "GENCB" on page 315.

**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 four bytes to accommodate the address of the I/O buffer in 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). In this case, you must also specify its size via 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 issue the SHOWCB macro instruction. If necessary, review the entry titled "SHOWCB" on page 375.

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 issue 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 format box above. Note that the parameters are arranged in groups, each with a value that will be assumed by default should you fail to specify from that group. Since they are not positional parameters, they can be specified in any order.

**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.

**<u>KEY</u>**
Indicates access to a key-sequenced or relative record file.

**DIR**
Indicates direct processing.

**<u>SEQ</u>**
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 proceed through the file in a forward direction.

**BWD**
Indicates that processing is to proceed through the file in a backward direction for keyed or addressed access, and for sequential or direct processing.

**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 issue the CHECK instruction to obtain the results of the operation. If necessary, review the entry titled "CHECK" on page 302.

**NSP**
Indicates that, for direct processing only, your request is not for update. Further, 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 that of 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.

**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.

**RECLEN**
Specifies the length, in bytes, of a record that is to be stored.

If you intend to issue the PUT instruction, then this parameter is required. If you issue 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.

**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 instruction to tailor each RPL to the specific file and type of processing you wish. Review the entry titled "MODCB" on page 347.

Unless you specify otherwise, GCS will generate only one copy.

**LENGTH**
Specifies the length of the area you are supplying in virtual storage to accommodate the RPL(s) you want to generate. Express this figure in bytes.

**WAREA**
Specifies the address of the area you are supplying in virtual storage to accommodate the RPL(s) 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. Furthermore, 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.

## Usage Notes

- The GENCB macro generates an RPL at execution time. Contrast this with the RPL macro, which generates an RPL at assembly time. If necessary, review the entry titled "RPL" on page 361.
- Each time you issue the GENCB instruction, you must provide the system with a 72-byte save area. Be certain that before you issue the instruction you place the address of this save area in register 13.
- Be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

## Completion Codes, Return Codes and Abend Codes

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 were not specified, then register 0 contains the length of the area in 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.

| Return Code | Meaning |
|---|---|
| 1 | The type of your request is invalid. |
| 2 | You selected a request parameter list. This is invalid. |
| 3 | One of the keyword codes in the parameter list is invalid. |
| 8 | There is not enough virtual storage to generate the RPL. |
| 9 | The area you specified in the WAREA parameter is not large enough to generate the RPL. |
| 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

## Retrieve a Record from a VSAM File

Use the GET macro instruction 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 instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

The format of the GET macro instruction is:

| | | |
|---|---|---|
| [label] | GET | RPL=address |

## Parameter

**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 via the RPL macro instruction. (If necessary, review the entry titled "RPL" on page 361.)

You can write this parameter as an assembler program label or as register (1) through (12).

## Usage Notes

- Each time you issue the GET instruction, you must provide the system with a 72-byte save area. Be certain that before you issue the instruction you place the address of this save area in register 13.
- Be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

## 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 via the SHOWCB or TESTCB macro instructions. If necessary, review the entries titled "SHOWCB" on page 375 or "TESTCB" on page 390.

| Return Code | Meaning |
|---|---|
| 0 | Your request was accepted. |
| 4 | Your request was not accepted because the RPL you specified in the GET instruction is already active for another request. |
| 8 | A logical error occurred. |
| 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. |

# MODCB

## Modify Certain Fields in an Access Method Control Block (ACB)

An access method control block (ACB) defines certain characteristics of a file that you intend to process via VSAM. When the file is opened, other characteristics of the file that you defined via the DLBL command are merged with the ACB to complete the picture. For more information on the DLBL command see "ACB" on page 295 and "GCS Commands" on page 20.

Use the MODCB macro instruction at execution time to add or modify certain fields in an ACB that you previously created.

This discussion of the MODCB macro instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

The format of the MODCB macro instruction is shown on the following page.

```
[label]     MODCB      ACB=address  ┌                ┐
                                    │ ,MACRF=( ┌ADR ┐│
                                    │          │CNV ││
                                    │          │KEY ││
                                    │          └    ┘│
                                    │         [,NDF] │
                                    │          ┌,DIR┐│
                                    │          │,SEQ││
                                    │          │,SKP││
                                    │          └    ┘│
                                    │          ┌,IN ┐│
                                    │          │,OUT││
                                    │          └    ┘│
                                    │         {,NRM} │
                                    │         {,AIX} │
                                    │         {,NRS} │
                                    │         {,RST} │
                                    │         [,NSR] │
                                    │         {,NUB} │
                                    │         {,UBF})│
                                    └                ┘
                       [,BUFND=number][,BUFNI=number][,BUFSP=number]
                       [,DDNAME=ddname][,EXLST=address][,MAREA=address]
                       [,MLEN=number][,PASSWD=address][,STRNO=number]
```

Defaults (underlined in the diagram): KEY, NDF, SEQ, IN, NRM, NRS, NSR, NUB.

## Parameters

**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, much less process it.

Carefully check the format box above. Note that the processing options are arranged in groups, each with a value that will be assumed by default should you fail to specify from that group. Since they are not positional parameters, they can be specified in any order. Carefully note whether a set of brackets surrounds each individual member of a group, or a set of braces surrounds the entire group. In the former case, you may select more than one of the parameters from the group. In the latter case, you may select only one of them.

**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 instruction will not be deferred for a direct PUT instruction.

**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 via the alternate index.

**<u>NRS</u>**
: 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. That is, it resets it to the status it had before the file was open the first time. If necessary, review the entry titled "OPEN" on page 354. 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 performs 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.

**<u>NSR</u>**
: Indicates that the resources are not shared.

**<u>NUB</u>**
: 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 instructions 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 instruction. If necessary, review the entries titled "RPL" on page 361 and "GENCB" on page 327.

**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 largest of the following, by default:

- The buffer size specified in the catalog.

  This buffer size was specified via 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 accommodate 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 via the BUFSP parameter on the DLBL command that identifies your file. This value overrides the BUFSP parameter in the ACB macro instruction. Likewise, 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. Then, any left-over space is used for data buffers. Finally, if any left-over space remains that is insufficient to accommodate 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 instruction determine the size of the user buffer area. (If necessary, review the entry titled “RPL” on page 361.)

**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 via the MODCB macro instruction.

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 via the EXLST or GENCB macro instruction. If necessary, review the entries titled "EXLST" on page 311 or "GENCB" on page 323.

If you used the EXLST instruction to create this list, then you can write this parameter as the label on that instruction. If you used the GENCB instruction, 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 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 zero, 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.

If you omit this parameter, then its value is 1, by default.

## Usage Notes

- You can add or modify any parameter listed above. 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 that of another value, then the new value supplants the old. For example, if the ACB presently specifies the MACRF=UBF parameter, and you specify the MACRF=NUB parameter in the MODCB instruction, then NUB replaces UBF.
- 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.
- Each time you issue the MODCB instruction, you must provide the system with a 72-byte save area. Be certain that before you issue the instruction you place the address of this save area in register 13.
- Be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

## Completion Codes, Return Codes and Abend Codes

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 instruction 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.

| Return Code | Meaning |
|---|---|
| 1 | The type of request was invalid. |
| 2 | The block type was invalid. |
| 3 | One of the keywords in the parameter list is invalid. |
| 4 | The block at the address you specified was not of the type you indicated. |
| 12 | The file associated with the ACB in question is open. Hence, it cannot be modified. |
| 14 | You specified an incompatible set of parameters for MACRF. |

| Return Code | Meaning |
|---|---|
| 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. |

## MODCB

### Modify an Exit List at Execution Time

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. Review the entry titled "MODCB" on page 336.

This discussion of the MODCB macro instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

Use the MODCB macro instruction at execution time to modify a previously created list that contains the addresses of your exit routines.

The format of the MODCB EXLST macro instruction is:

| | | |
|---|---|---|
| [label] | MODCB | EXLST=address<br>[,EODAD=(address [,A \| ,N] [,L])]<br>[,JRNAD=(address [,A \| ,N] [,L])]<br>[,LERAD=(address [,A \| ,N] [,L])]<br>[,SYNAD=(address [,A \| ,N] [,L])] |

### Parameters

**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 to 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 eight-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 Notes

- It does not matter whether the file whose exit list you are trying to modify is opened or closed. You can issue the MODCB instruction in either case.
- The exit list you want to modify is of 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.
- Each time you issue the MODCB instruction, you must provide the system with a 72-byte save area. Be certain that before you issue the instruction you place the address of this save area in register 13.
- Be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

## Completion Codes, Return Codes and Abend Codes

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 instruction 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.

| Return Code | Meaning |
|---|---|
| 1 | The type of request was invalid. |
| 2 | The block type was invalid. |
| 3 | One of the keywords in the parameter list is invalid. |
| 4 | The block at the address you specified was not of the type you indicated. |
| 7 | Either the exit list is not large enough to accommodate your modification or the exit entry you tried to modify was not in the list at all. |
| 10 | You failed to specify an address for one of your exit routines. Also, you must specify either the A or N parameter. |
| 13 | You attempted to activate an exit, but did not provide an address for it. |
| 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. |

## MODCB

### Modify Certain Fields in a Request Parameter List (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. If necessary, review the entry titled "CHECK" on page 302, "ENDREQ" on page 307, "ERASE" on page 309, "GET" on page 334, "POINT" on page 357, or "PUT" on page 359.

This discussion of the MODCB macro instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

Use the MODCB macro instruction to modify specified fields within a certain request parameter list.

The format of the MODIFY RPL macro instruction is on the following page.

```
[label]   MODCB   RPL=address[,ACB=address][,AREA=address]
                  [,AREALEN=number][,ARG=address]
                  [,ECB=address][,KEYLEN=number][,NXTRPL=address]
                  ┌          ┌   ┐  ┐
                  │,OPTCD=(  │ADR│  │
                  │          │CNV│  │
                  │          │KEY│  │
                  │          └   ┘  │
                  │                 │
                  │          ┌    ┐ │
                  │          │,DIR│ │
                  │          │,SEQ│ │
                  │          │    │ │
                  │          │,SKP│ │
                  │          └    ┘ │
                  │                 │
                  │          ┌    ┐ │
                  │          │,ARD│ │
                  │          │    │ │
                  │          │,LRD│ │
                  │          └    ┘ │
                  │                 │
                  │          ┌    ┐ │
                  │          │,FWD│ │
                  │          │    │ │
                  │          │,BWD│ │
                  │          └    ┘ │
                  │                 │
                  │          ┌    ┐ │
                  │          │,ASY│ │
                  │          └    ┘ │
                  │                 │
                  │          ┌    ┐ │
                  │          │,NSP│ │
                  │          │,NUP│ │
                  │          │    │ │
                  │          │,UPD│ │
                  │          └    ┘ │
                  │                 │
                  │          ┌    ┐ │
                  │          │,KEQ│ │
                  │          │    │ │
                  │          │,KGE│ │
                  │          └    ┘ │
                  │                 │
                  │          ┌    ┐ │
                  │          │,FKS│ │
                  │          │    │ │
                  │          │,GEN│ │
                  │          └    ┘ │
                  │                 │
                  │          ┌    ┐ │
                  │          │,LOC│ │
                  │          │,MVE│)│
                  │          └    ┘ │
                  └                 ┘
                  [,RECLEN=number]
```

## Parameters

**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. That is, an RPL that defines a request that has been issued, but is not yet completed. To confirm whether an active request is complete, issue the CHECK macro instruction. To cancel an active request, issue the ENDREQ macro instruction. If necessary, review the entries "CHECK" on page 302 and "ENDREQ" on page 307.

**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 via the ACB macro instruction, 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 via the ACB or GENCB macro instruction before issuing the RPL instruction. If necessary, review the entry titled "ACB" on page 295 or "GENCB" on page 315.

**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 four bytes to accommodate the address of the I/O buffer in 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). In this case, you must also specify its size via 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 issue the SHOWCB macro instruction. If necessary, review the entry titled "SHOWCB" on page 375.

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 instruction that generates the last RPL in the chain. When you issue 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.

**FWD**
Indicates that processing is to proceed through the file in a forward direction.

**BWD**
Indicates that processing is to proceed through the file in a backward direction for keyed or addressed access and for sequential or direct processing.

**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 issue the CHECK macro instruction to obtain the results of the operation. If necessary, review the entry titled "CHECK" on page 302.

**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 that of 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 invoke the PUT or ERASE instructions, though it is valid with the GET instruction. 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 issue any PUT instruction. 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 issue the PUT instruction, then this parameter is required. If you issue 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.

## Usage Notes

- 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.
- 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 instruction, then that option overrides any other option from that group that might be specified in the RPL.
- Each time you issue the MODCB instruction, you must provide the system with a 72-byte save area. Be certain that before you issue the instruction you place the address of this save area in register 13.
- Be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

## Completion Codes, Return Codes and Abend Codes

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 instruction 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.

| Return Code | Meaning |
|---|---|
| 1 | The type of request was invalid. |
| 2 | The block type was invalid. |
| 3 | One of the keywords in the parameter list is invalid. |
| 4 | The block at the address you specified was not of the type you indicated. |
| 11 | The MODCB macro is already active on the specified control block. |
| 14 | You specified an incompatible set of parameters. |
| 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. |

# OPEN

## Prepare a VSAM File for Processing

Before your program can access a file, the file must be opened for processing. The process of thus 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 via the ACB or GENCB macro instructions.
- And, verifying any necessary passwords to the file. (If necessary, review the entry titled "ACB" on page 295 or "GENCB" on page 315.)

Use the OPEN macro instruction to prepare a VSAM file for processing.

This discussion of the OPEN macro instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

The format of the OPEN macro instruction is:

| | | |
|---|---|---|
| [label] | OPEN | (acb address,acb address ...) |

## Parameters

**acb address**
Specifies the address of the access method control block (ACB) associated with the file you wish to open.

Note that you can specify the address of more than one, and thereby 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, 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 set of parentheses.

## Usage Notes

- Be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.
- If you have data control blocks (DCBs) that you wish to open, as well as ACBs, you can specify a combination of both in the same OPEN instruction. GCS is able to distinguish the address of one from the address of the other, as long as you separate each with a comma. This is a way of saying that, for all practical purposes, this instruction and the one described in the entry titled "OPEN (BSAM/QSAM)" on page 262 are one and the same.

## 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 using the SHOWCB. instruction. If necessary, review the entry titled "SHOWCB" on page 367.

| Completion Code | Meaning |
|---|---|
| 0 | All the files specified are now opened. |
| 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. |

| Return Code | Meaning |
|---|---|
| 4 | The file indicated by the access method control block is already open. |
| 96 | Catalog recovery for this file failed. Therefore, the file is not usable. |
| 100 | The OPEN macro found an empty alternate index that is part of an upgrade set. |
| 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. |
| 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. |
| 116 | The file was not properly closed. |
| 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. |
| 132 | A permanent I/O error occurred while VSAM was reading label information. |
| 136 | Insufficient virtual storage is available for work areas, control blocks, or buffers. |
| 144 | An I/O error, which cannot be corrected, occurred while VSAM was reading or writing a catalog record. |

| Return Code | Meaning |
|---|---|
| 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. |
| 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. |
| 160 | The parameters specified in the ACB or GENCB instruction are either inconsistent with each other, or inconsistent with the information in the catalog record. |
| 168 | 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. |
| 180 | An error occurred in opening a catalog. |
| 188 | The file specified by the access method control block is not one that can be specified by an ACB. |
| 192 | An unusable file was opened for output. |
| 196 | Access to data was requested via an empty alternate index. |
| 232 | RESET was specified for a non-reusable file, but the file is not empty. |

| Abend Code | Meaning |
|---|---|
| 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

### Position a Pointer for Access to a Specific Record in a VSAM File

To access a certain record within a VSAM file, you must "position yourself" within that file and "point" to the record in question.

Use the POINT macro instruction to position yourself forward or backward within the file to a specific record.

This discussion of the POINT macro instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

The format of the POINT macro instruction is:

| | | |
|---|---|---|
| [label] | POINT | RPL=address |

### Parameter

**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 via the RPL macro instruction. (If necessary, review the entry titled "RPL" on page 361.) 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 Notes

- 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 instruction to position either forward or backward within the file.
- 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 instruction to position either forward or backward within the file.
- VSAM also can be positioned for sequential processing by either a direct GET or PUT instruction.
- Each time you issue the POINT instruction, you must provide the system with a 72-byte save area. Be certain that before you issue the instruction you place the address of this save area in register 13.
- Be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

## 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 via the SHOWCB or TESTCB macro instructions. If necessary, review the entries titled "SHOWCB" on page 375 or "TESTCB" on page 390.

| Return Code | Meaning |
|---|---|
| 0 | Your request was accepted. |
| 4 | Your request was not accepted because the RPL you specified in the POINT instruction is already active for another request. |
| 8 | A logical error occurred. |
| 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

### Store a Record in a VSAM file

Use the PUT macro instruction to move a record from an I/O buffer or work area into a VSAM file.

This discussion of the PUT macro instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

The format of the PUT macro instruction is:

| | | |
|---|---|---|
| [label] | PUT | RPL=address |

### Parameter

**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 via the RPL macro instruction. (If necessary, review the entry titled "RPL" on page 361.)

You can write this parameter as an assembler program label or as register (2) through (12).

### Usage Notes

- Each time you issue the PUT instruction, you must provide the system with a 72-byte save area. Be certain that before you issue the instruction you place the address of this save area in register 13.
- Be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

### 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 via the SHOWCB or TESTCB macro instructions. If necessary, review the entries titled "SHOWCB" on page 375 or "TESTCB" on page 390.

| Return Code | Meaning |
|---|---|
| 0 | Your request was accepted. |
| 4 | Your request was not accepted because the RPL you specified in the PUT instruction is already active for another request. |
| 8 | A logical error occurred. |
| 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. |

## RPL

### Create a Request Parameter List (RPL) at Assembly Time

Certain 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. If necessary, review the entry titled "CHECK" on page 302, "ENDREQ" on page 307, "ERASE" on page 309, "GET" on page 334, "POINT" on page 357, or "PUT" on page 359. Also, be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

This discussion of the RPL macro instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

Use the RPL macro instruction to create a request parameter list at assembly time describing the characteristics of your VSAM request.

The format of the RPL macro instruction is on the following page.

```
[label]   RPL   [ACB=address][,AREA=address]
                [,AREALEN=number][,ARG=address][,ECB=address]
                [,KEYLEN=number][,NXTRPL=address]
                ┌                ┐
                │ ,OPTCD=(┌ADR┐ │
                │         │CNV│ │
                │         │KEY│ │
                │         └   ┘ │
                │         ┌    ┐ │
                │         │,DIR│ │
                │         │,SEQ│ │
                │         │,SKP│ │
                │         └    ┘ │
                │         ┌    ┐ │
                │         │,ARD│ │
                │         │,LRD│ │
                │         └    ┘ │
                │         ┌    ┐ │
                │         │,FWD│ │
                │         │,BWD│ │
                │         └    ┘ │
                │         [,ASY] │
                │         ┌    ┐ │
                │         │,NSP│ │
                │         │,NUP│ │
                │         │,UPD│ │
                │         └    ┘ │
                │         ┌    ┐ │
                │         │,KEQ│ │
                │         │,KGE│ │
                │         └    ┘ │
                │         ┌    ┐ │
                │         │,FKS│ │
                │         │,GEN│ │
                │         └    ┘ │
                │         ┌    ┐ │
                │         │,LOC│ │
                │         │,MVE│)│
                │         └    ┘ │
                └                ┘
                [,RECLEN=number]
```

## 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 via the ACB macro instruction, 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 via the GENCB macro instruction. If necessary, review the entry titled "GENCB" on page 315.

**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.

**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 four bytes to accommodate the address of the I/O buffer in 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). In this case, you must also specify its size via 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 issue the SHOWCB macro instruction. If necessary, review the entry titled "SHOWCB" on page 375.

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 issue 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 format box above. Note that the parameters are arranged in groups, each with a value that will be assumed by default should you fail to specify from that group. Since they are not positional parameters, they can be specified in any order.

**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.

**FWD**
Indicates that processing is to proceed through the file in a forward direction.

**BWD**
Indicates that processing is to proceed through the file in a backward direction for keyed or addressed access, and for sequential or direct processing.

**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 issue the CHECK instruction to obtain the results of the operation.

**NSP**
Indicates that, for direct processing only, you request is not for update. Further, 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 that of 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.

**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.

**RECLEN**

Specifies the length, in bytes, of a record that is to be stored.

If you intend to issue the PUT instruction, then this parameter is required. If you issue 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.

## SHOWCB

### Display the Fields of an Access Method Control Block (ACB)

An access method control block (ACB) defines certain characteristics of a file that you intend to process via VSE/VSAM. When the file is opened, other characteristics of the file that you defined via the DLBL command are merged with the ACB to complete the picture.

Use the SHOWCB macro instruction to display certain fields of an ACB. The display appears in a virtual storage work area that you set aside for this purpose.

Note that the contents of each field (except the ACBLEN field) is determined by the corresponding parameter in the ACB macro instruction, the GENCB macro instruction, and/or the DLBL command. If necessary, review the entries titled "ACB" on page 295 and "GENCB" on page 315. For more information on the DLBL command see "GCS Commands" on page 20. Also, be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

This discussion of the SHOWCB macro instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

The format of the SHOWCB ACB macro instruction is:

```
[label]   SHOWCB   ACB=address,AREA=address,LENGTH=number [,OBJECT= { DATA  } ]
                                                                  { INDEX }

                   ,FIELDS=([ACBLEN][,AVSPAC][,BUFND]
                           [,BUFNI][,BUFNO][,BUFSP]
                           [,CINV][,DDNAME][,ERROR]
                           [,EXLST][,FS][,KEYLEN]
                           [,LRECL][,MAREA][,MLEN]
                           [,NCIS][,NDELR][,NEXCP]
                           [,NEXT][,NINSR][,NIXL]
                           [,NLOGR][,NRETR][,NSSS]
                           [,NUPDR][,PASSWD][,RKP]
                           [,STMST][,STRNO])
```

## Parameters

**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 issued the ACB macro instruction 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 accommodate 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 instruction.

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 accommodate the ACB fields to be displayed.

Check the field parameters listed below to determine the necessary length for this work area. If the area is not large enough to accommodate 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. Moreover, 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 accommodate 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.

**BUFNO (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 via 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 upon whether you access the base cluster via 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.

## Usage Note

- Each time you issue the SHOWCB instruction, you must provide the system with a 72-byte save area. Be certain that before you issue the instruction you place the address of this save area in register 13.

## Completion Codes, Return Codes and Abend Codes

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 instruction 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.

| Return Code | Meaning |
|---|---|
| 1 | The type of request was invalid. |
| 2 | The block type was invalid. |
| 3 | One of the keywords in the parameter list is invalid. |
| 4 | The block at the address you specified was not of the type you indicated. |
| 5 | Either the file associated with the ACB in question is not open or is not a VSAM file. |
| 6 | Index information was requested, but no index was opened for the file in question. |
| 9 | The work area you provided to accommodate the fields to be displayed is too small. |
| 15 | The work area you provided to accommodate the fields to be displayed is not on a fullword boundary. |
| 16 | You specified an invalid control block address in the ACB parameter. |
| 20 | You specified certain parameters that can apply only if MACRF=LSR or MACRF=GSR. MACRF=LSR and MACRF=GSR are not supported by GCS. |

| 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. |

## SHOWCB

### Display the Fields of an Exit List

During VSAM processing, unusual conditions sometimes occur. If you wish, you can supply one or more exit routines to handle such conditions. If necessary, review the entries titled "EXLST" on page 311 or "GENCB" on page 323. 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. Review the entry titled "ACB" on page 295 or "MODCB" on page 336.

This discussion of the SHOWCB macro instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

Use the SHOWCB macro instruction to display certain fields of an exit list. This display appears in a virtual storage work area that you set aside for this purpose.

The format of the SHOWCB EXLST macro instruction is:

| | | |
|---|---|---|
| [label] | SHOWCB | EXLST=address,AREA=address,LENGTH=number,<br>FIELDS=([EODAD][,EXLLEN][,JRNAD][,LERAD][,SYNAD]) |

### Parameters

**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 instruction 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. Note that the fields are displayed in the order which you specify them in the SHOWCB instruction.

**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 four bytes for each field you specify in the FIELD parameter. If the work area is not large enough to accommodate 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 Notes

- Use the SHOWCB macro instruction to display a certain field in an exit list only if that field exists.

  GCS will display the fields in the order in which you request them.
- Each time you issue the SHOWCB instruction, you must provide the system with a 72-byte save area. Be certain that before you issue the instruction you place the address of this save area in register 13.
- Be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

## Completion Codes, Return Codes and Abend Codes

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 instruction 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.

| Return Code | Meaning |
|---|---|
| 1 | The type of request was invalid. |
| 2 | The block type was invalid. |
| 3 | One of the keywords in the parameter list is invalid. |
| 4 | The block at the address you specified was not of the type you indicated. |
| 7 | The type of exit you specified is not in the exit list. |
| 9 | The work area you provided to accommodate the fields to be displayed is too small. No fields were displayed. |
| 15 | The work area you provided to accommodate the fields to be displayed is not on a fullword boundary. No fields were displayed. |
| 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. |

## SHOWCB

### Display the Fields of a Request Parameter List

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. If necessary, review the entry titled "CHECK" on page 302, "ENDREQ" on page 307, "ERASE" on page 309, "GET" on page 334, "POINT" on page 357, or "PUT" on page 359. Also, be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

You create a request parameter list via the RPL or GENCB macro instructions. If necessary, review the entry titled "RPL" on page 361 or "GENCB" on page 327.

This discussion of the SHOWCB macro instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

Use the SHOWCB macro instruction to display certain fields of a request parameter list. This display appears in a work area that you have set aside for this purpose.

The format of the SHOWCB RPL macro instruction is:

| | | |
|---|---|---|
| [label] | SHOWCB | RPL=address,AREA=address,LENGTH=number,<br>FIELDS=([ACB][,AIXPC][,AREA]<br>[,AREALEN][,ARG][,ECB]<br>[,FDBK][,FTNCD][,KEYLEN]<br>[,NXTRPL][,RBA][,RECLEN]<br>[,RPLLEN]) |

### Parameters

**RPL**

Specifies the address of the request parameter list whose fields you want to display.

Since 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 instruction 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. Moreover, the fields are displayed in this work area in the order in which you list them in the SHOWCB instruction.

**LENGTH**
Specifies the length, in bytes, of the work area in virtual storage you have set aside to accommodate the RPL fields you want to display.

Each RPL field requires one fullword. Therefore, allow four 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 issue the CHECK macro instruction to place the return code in this field. (If necessary, review the entry titled "CHECK" on page 302.) 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 in 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.

## Usage Note

- Each time you issue the SHOWCB instruction, you must provide the system with a 72-byte save area. Be certain that before you issue the instruction you place the address of this save area in register 13.

## Completion Codes, Return Codes and Abend Codes

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 instruction 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.

| Return Code | Meaning |
|---|---|
| 1 | The type of request was invalid. |
| 2 | The block type was invalid. |
| 3 | One of the keywords in the parameter list is invalid. |
| 4 | The block at the address you specified was not of the type you indicated. |
| 9 | The work area you provided to accommodate the fields to be displayed is too small. No fields were displayed. |
| 15 | The work area you provided to accommodate the fields to be displayed is not on a fullword boundary. No fields were displayed. |
| 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. |

# TESTCB

## Test a Certain Field in an Access Method Control Block (ACB)

An access method control block (ACB) defines certain characteristics of a file that you intend to process via VSAM. When the file is opened, other characteristics of the file that you defined via the DLBL command are merged with the ACB to complete the picture.

Use the TESTCB macro instruction to test the value of a certain field in an ACB.

Note that the contents of each field (except the ACBLEN field) is determined by the corresponding parameter in the ACB macro instruction, the GENCB macro instruction, and/or the DLBL command. If necessary, review the entries titled "ACB" on page 295 and "GENCB" on page 315. For more information on the DLBL command see "GCS Commands" on page 20.

This discussion of the TESTCB macro instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

The format of the TESTCB ACB is on the following page.

```
[label]    TESTCB    ACB=address[,ERET=address][,OBJECT= {DATA }]
                                                          {INDEX}

                      [,ATRB= ([ESDS][,KSDS][,REPL][,RRDS][,SPAN][,SSWD][,WCK])
                              UNQ

                      ,MACRF=([ADR][,AIX][,CNV]
                              [,DIR][,IN][,KEY][,NDF]
                              [,NRM][,NSR][,NUB][,OUT]
                              [,RST][,SEQ][,SKP][,UBF])

                       [ACBLEN ]
                       [AVSPAC ]
                       [BUFND  ]
                       [BUFNI  ]
                       [BUFNO  ]
                       [BUFSP  ]
                       [CINV   ]
                       [DDNAME ]
                       [ERROR  ]
                       [EXLST  ]
                       [FS     ]
                       [KEYLEN ]
                       [LRECL  ]
                       [MAREA  ]
                       [MLEN   ] =value
                       [NCIS   ]
                       [NDELR  ]
                       [NEXCP  ]
                       [NEXT   ]
                       [NINSR  ]
                       [NIXL   ]
                       [NLOGR  ]
                       [NRETR  ]
                       [NSSS   ]
                       [NUPDR  ]
                       [PASSWD ]
                       [RKP    ]
                       [STMST  ]
                       [STRNO  ]

                       OFLAGS=OPEN

                       OPENOBJ= {PATH}
                                {BASE}
                                {AIX }                                      ]
```

## Parameters

**ACB**

Specifies the address of the ACB that contains the information you want to test.

Since 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 instruction, since 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.

**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.

**WCK**

Whether write operations for the file are being verified.

**UNQ**

Whether the alternate index requires unique keys.

**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.

**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 instruction will not be deferred for a direct PUT instruction.

**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 via 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 first opened. If necessary, review the entry titled "OPEN" on page 354. 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 performs 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.

**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 instructions 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 instruction. If necessary, review the entries titled "RPL" on page 361 and "GENCB" on page 327.

**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 via 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 upon whether you access the base cluster via 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 equivalent 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.

**OFLAGS**
Indicates that a test will be made to determine whether a file for which the OPEN macro instruction has been issued is indeed 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 Notes

- You can use the TESTCB instruction 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.
- Each time you issue the TESTCB instruction, you must provide the system with a 72-byte save area. Be certain that before you issue the instruction you place the address of this save area in register 13.
- Be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

## Completion Codes, Return Codes and Abend Codes

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 instruction 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.

| Return Code | Meaning |
|---|---|
| 1 | The type of request was invalid. |
| 2 | The block type was invalid. |
| 3 | One of the keywords in the parameter list is invalid. |
| 4 | The block at the address you specified was not of the type you indicated. |
| 5 | Either the file associated with the ACB in question is not open or is not a VSAM file. |
| 6 | Index information was requested, but no index was opened for the file in question. |
| 14 | The MACRF and/or ATRB parameters contain incompatible options. |
| 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. |

## TESTCB

### Test a Certain Field in an Exit List

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. Review the entry titled "MODCB" on page 336.

This discussion of the TESTCB macro instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

Use the TESTCB macro instruction to test the value of a certain field in an exit list.

The format of the TESTCB EXLST macro instruction follows.

```
[label]     TESTCB     EXLST=address[,ERET=address]
                       ┌                                ┐
                       │,EODAD=  0                      │
                       │                  ┌  ┐          │
                       │         (address │,A│[,L])    │
                       │                  │,N│          │
                       │                  └  ┘          │
                       │,JRNAD=  0                      │
                       │                  ┌  ┐          │
                       │         (address │,A│[,L])    │
                       │                  │,N│          │
                       │                  └  ┘          │
                       │,LERAD=  0                      │
                       │                  ┌  ┐          │
                       │         (address │,A│[,L])    │
                       │                  │,N│          │
                       │                  └  ┘          │
                       │,SYNAD=  0                      │
                       │                  ┌  ┐          │
                       │         (address │,A│[,L])    │
                       │                  │,N│          │
                       │                  └  ┘          │
                       └                                ┘
                       [,EXLLEN=number]
```

## Parameters

**EXLST**
Specifies the address of the exit list whose information you want to test.

Since the same maximum length applies to every exit list, you can omit this parameter if you want to test the EXLLEN 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 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 instruction, since 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.

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 tests described below.

**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 eight-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 EXLST parameter, 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 Notes

- You can use the TESTCB instruction 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.

- Each time you issue the TESTCB instruction, you must provide the system with a 72-byte save area. Be certain that before you issue the instruction you place the address of this save area in register 13.

- Be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

## Completion Codes, Return Codes and Abend Codes

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 instruction 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.

| Return Code | Meaning |
|---|---|
| 1 | The type of request was invalid. |
| 2 | The block type was invalid. |
| 3 | One of the keywords in the parameter list is invalid. |
| 4 | The block at the address you specified was not of the type you indicated. |
| 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. |

# TESTCB

## Test a Certain Field in a Request Parameter List (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. If necessary, review the entry titled "CHECK" on page 302, "ENDREQ" on page 307, "ERASE" on page 309, "GET" on page 334, "POINT" on page 357, or "PUT" on page 359.

This discussion of the TESTCB macro instruction deals only with those matters that involve GCS and is not intended to be an exhaustive instruction on this or any other VSE/VSAM topic. Presumably you are already familiar with VSE/VSAM through past experience and through study of the VSE/VSAM manuals.

Use the TESTCB macro instruction to test a certain field in a request parameter list.

The format of the TESTCB RPL macro instruction follows.

| | | |
|---|---|---|
| [label] | TESTCB | RPL=address[,ERET=address] |

```
[label]    TESTCB    RPL=address[,ERET=address]
                       ⎧ AIXFLAG=AIXPKP         ⎫
                       ⎪ AIXPC=number           ⎪
                       ⎪ FTNCD=number           ⎪
                       ⎪ IO=COMPLETE            ⎪
                       ⎪ OPTCD=(⎧,ADR⎫          ⎪
                       ⎪        ⎪,CNV⎪          ⎪
                       ⎪        ⎪,KEY⎪          ⎪
                       ⎪        ⎪,DIR⎪          ⎪
                       ⎪        ⎪,SEQ⎪          ⎪
                       ⎪        ⎪,SKP⎪          ⎪
                       ⎪        ⎪,ARD⎪          ⎪
                       ⎪        ⎪,LRD⎪          ⎪
                       ⎪        ⎪,FWD⎪          ⎪
                       ⎪        ⎪,BWD⎪          ⎪
                       ⎪        ⎨,ASY⎬          ⎪
                       ⎪        ⎪,SYN⎪          ⎪
                       ⎪        ⎪,NSP⎪          ⎪
                       ⎨        ⎪,NUP⎪          ⎬
                       ⎪        ⎪,UPD⎪          ⎪
                       ⎪        ⎪,KEQ⎪          ⎪
                       ⎪        ⎪,KGE⎪          ⎪
                       ⎪        ⎪,FKS⎪          ⎪
                       ⎪        ⎪,GEN⎪          ⎪
                       ⎪        ⎪,LOC⎪          ⎪
                       ⎪        ⎩,MVE⎭ )        ⎪
                       ⎪ ACB=address            ⎪
                       ⎪ AREA=address           ⎪
                       ⎪ AREALEN=number         ⎪
                       ⎪ ARG=address            ⎪
                       ⎪ ECB=address            ⎪
                       ⎪ FDBK=number            ⎪
                       ⎪ KEYLEN=number          ⎪
                       ⎪ NXTRPL=address         ⎪
                       ⎪ RBA=number             ⎪
                       ⎪ RECLEN=number          ⎪
                       ⎩ RPLLEN=number          ⎭
```

## Parameters

**RPL**

Specifies the address of the RPL whose field you want to test.

Since all RPLs are the same length, you can omit this parameter if you are testing the RPLLEN 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 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 instruction, since 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.

**AIXFLAG = AIXPKP**
Indicates whether the alternate index just processed contains prime key pointers.

**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.

**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.

**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.

**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.

**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 proceed through the file in a forward direction.

**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.

**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.

**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 GCS is to remember the current position within the file for subsequent, sequential access.

Only the ENDREQ macro instruction 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.

**UPD**
Indicates that any record retrieved can be updated or deleted.

On direct and sequential requests, GCS will remember the record's position.

**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 issue the POINT instruction.

**KGE**
Indicates that if the key you specify as a search argument does not equal that of 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 subsequent PUT instruction will insert the record at this position.

**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 the I/O buffer to be processed.

This parameter is not valid if you intend to invoke the PUT or ERASE instructions, though it is valid with the GET instruction. 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 issue any PUT instruction. For keyed-sequential retrieval, modifying key fields in the I/O buffer may cause erroneous results in 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.

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.

**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**
Specifies the return code from the request associated with this RPL.

For asynchronous requests, you must issue the CHECK macro instruction to place the return code in this field. (If necessary, review the entry titled "CHECK" on page 302.) 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 in which a logical or physical error occurred. It indicates whether the upgrade set may have been modified incorrectly by the request.

**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.

## Usage Notes

- You can use the TESTCB instruction 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.
- Each time you issue the TESTCB instruction, you must provide the system with a 72-byte save area. Be certain that before you issue the instruction you place the address of this save area in register 13.
- Be certain that you are familiar with the material covered in the entry titled "Using VSAM under GCS" on page 280.

## Completion Codes, Return Codes and Abend Codes

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 instruction 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.

| Return Code | Meaning |
|---|---|
| 1 | The type of request was invalid. |
| 2 | The block type was invalid. |
| 3 | One of the keywords in the parameter list is invalid. |
| 4 | The block at the address you specified was not of the type you indicated. |
| 14 | The MACRF and/or ATRB parameters contain incompatible options. |
| 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. |

# Chapter 14. IUCV Service Macros

# IUCVCOM

## Communicate within the IUCV 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. By using the GCS IUCV support, communications can take place among several users operating within several tasks operating within several virtual machines.

When the word "user" appears, you should take it to mean any supervisor or problem program.

Use the IUCVCOM macro instruction to coordinate communication among users within the IUCV environment.

This treatment of the IUCVCOM macro instruction assumes that you are already familiar with the section dealing with IUCV in the *VM/XA SP CP Programming Services*. For more information on IUCV, see "IUCVINI" on page 413.

The IUCVCOM macro instruction is available in standard, list, list address, and execute formats. The ***standard*** format follows.

```
[label]   IUCVCOM    QUERY[,OPTION B]
                     CONNECT,NAME=addr,PRMLIST=addr[,OPTION A][,OPTION B]
                     ACCEPT,NAME=addr,PRMLIST=addr[,OPTION A][,OPTION B]
                     SEVER,NAME=addr,PRMLIST=addr[,OPTION B][,OPTION C]
                     QUIESCE,NAME=addr,PRMLIST=addr[,OPTION B][,OPTION C]
                     RESUME,NAME=addr,PRMLIST=addr[,OPTION B][,OPTION C]
                     SEND,NAME=addr,PRMLIST=addr[,OPTION B]
                     RECEIVE,NAME=addr,PRMLIST=addr[,OPTION B]
                     REPLY,NAME=addr,PRMLIST=addr[,OPTION B]
                     REJECT,NAME=addr,PRMLIST=addr[,OPTION B]
                     PURGE,NAME=addr,PRMLIST=addr[,OPTION B]
                     REP,NAME=addr,OPTION E[,OPTION B][,OPTION C][,OPTION D]

                   OPTION A:                   OPTION B:   OPTION C:
                   [EXIT=addr][,UWORD=addr]    ERROR=addr  CODE= { ALL }
                                                                 { ONE }
                   OPTION D:                       OPTION E:
                                                   ┌                      ┐
                   PATH=addr                       │[EXIT=addr,]UWORD=addr│
                                                   │EXIT=addr[,UWORD=addr]│
                                                   └                      ┘
```

## 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.

A user can invoke the QUERY function without first having issued the IUCVINI SET instruction.

**CONNECT**

Indicates that the user requests a communication path be established between it and the virtual machine or CP system service with whom the user is trying to communicate.

The user must identify the virtual machine or Cp system service with which it wishes to communicate using the CP IUCV 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 eight bytes of the IPUSER field, it should identify the particular user it wishes to contact in that machine.

Remember that all IUCV users, including privileged ones, must use the IUCVCOM CONNECT instruction to establish an IUCV path.

**ACCEPT**

Indicates that the user wants to complete the communication path initiated by another virtual machine or CP system service trying to communicate with it.

Remember that all IUCV users,including privileged ones, must use the IUCVCOM ACCEPT instruction to complete an IUCV 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 parameter list.

However, if the user issues an IUCVCOM SEVER instruction specifying the CODE=ALL parameter, then GCS issues an IUCV SEVER instruction for the user's paths, (the IUCV path). 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. Which path it is must be specified in the CP IUCV parameter list.

Remember that all IUCV users, including privileged ones, must use the IUCVCOM SEVER instruction to sever an IUCV 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 at this time. 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 issues 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 in 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 using an IUCVCOM ACCEPT instruction.

If the user specifies CODE=ONE (or allows it to default) then only one specific path shall be quiesced. Which path it is 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 issues 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 in 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 using an IUCVCOM ACCEPT instruction.

If the user specifies CODE=ONE (or allows it to default) then only one specific path shall be resumed. Which path it is must be specified in the CP IUCV parameter list.

**SEND**
Causes the user's message to be sent to the party at the other end of the specified path.

Presumably the virtual machine or CP system service at the other end of the path has consented to communicate using the IUCVCOM ACCEPT instruction.

**RECEIVE**
Indicates that the user accepts the data that was passed using an IUCV SEND function.

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 parameter list.

**REPLY**
Conveys the user's response to a message sent to it by another virtual machine or CP system service through the SEND function.

**REJECT**
Indicates that the user refuses to receive a specific message that some party sent to it via the SEND function.

The path in question must be identified in the IPPATHID field of the CP IUCV parameter list. Moreover, 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 virtual machine or CP system service. Whether the other virtual machine or CP system service received the message or not, the message is cancelled.

The path in question must be identified in the IPPATHID field of the CP IUCV parameter list. Moreover, 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 and/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 and/or UWORD are to be changed, then the REP function must be requested from the same task that established the path in the first place.

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 issued 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.

When an external interrupt occurs involving an unauthorized user, the exit routine gains control in the same state and key as the user. Furthermore, the exit runs enabled for all interrupts.

Upon entry to the exit routine, the registers contain the following:

| **Register** | **Contents** |
|---|---|
| Register 0 | The UWORD. |
| Register 1 | Unpredictable. |
| Register 2 | The address of the external interrupt buffer. |
| Registers 3 - 12 | Unpredictable |
| Register 13 | The address of a user save area. |
| Register 14 | The address to which control must be returned once the exit routine completes execution. |
| Register 15 | The address of the exit routine. |

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. Furthermore, the exit routine is disabled and cannot issue any SVC calls.

Upon entry to the exit routine, the registers contain the following:

| Register | Contents |
|---|---|
| Register 0 | The UWORD. |
| Register 1 | Unpredictable. |
| Register 2 | The address of the external interrupt buffer. |
| Registers 3 - 12 | Unpredictable |
| Register 13 | The address of the 72-byte register save area. |
| Register 14 | The address to which control must be returned once the exit routine completes execution. |
| Register 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). (In the case of the latter, GCS will sever the path involved in the error.) Moreover, 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 parameter list associated with the function the user wishes to perform.

Remember that every function in the IUCVCOM instruction (except QUERY and REP) requires such a parameter list. You are expected to provide one yourself. The list format of the IUCV macro instruction is a convenient way to create it.

**CODE**
Indicates the scope of the IUCVCOM function that the user wishes to perform.

The IUCVCOM functions SEVER, QUIESCE, RESUME, and REP require that their scope either be confined 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.[13]

**NAME**
Specifies the address of the name by which the user is known within the IUCV 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 and/or UWORD changed via 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).

## Usage Notes

- A CP IUCV parameter list must be created for each IUCVCOM function that requires the PRMLIST parameter. The *VM/XA SP CP Programming Services* can provide you with more information on this.
- No user can issue the IUCVCOM macro instruction before it is admitted to the IUCV environment through the IUCVINI SET instruction. The IUCVCOM QUERY function is the only exception to this.

[13] For the SEVER function, this path is the one specified in the CP IUCV 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 instruction.

- To ensure that no user tries to perform an IUCV 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 issue the IUCVCOM instruction 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.
- 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.

| Path State | Description |
|---|---|
| CONNECT issued | A user issued 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 issue two types of instructions:<br>• IUCVCOM RECEIVE if there was a connection parameter list extension specified by the CONNECT on an IUCV path. The user would remain in a CONNECT pending state.<br>• 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 issued 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 issued the IUCVCOM QUIESCE instruction. Therefore, that user will not receive incoming communication over the path, though he can communicate out. |
| SEVER in progress | One of the users using an active or QUIESCED path has issued 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 issue the same instruction to sever "his half of the path." |
| 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) via 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 issues 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.

- If a user invokes the SEVER, QUIESCE, or RESUME function with CODE=ALL specified, then all paths associated with that user are affected. When the function terminates 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.
- 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 parameter list.

## Return Codes and Abend Codes

When this macro completes processing, it passes to the caller a return code in register 15.

| Return Code | Meaning |
|---|---|
| 000 | Function completed successfully. |
| 008 | The user issued an IUCVCOM instruction before admitting itself to the IUCV environment using the IUCVINI SET instruction. |
| 012 | The user does not own the path in question. |
| 016 | Either the NAME parameter was not specified or its address is zero. |
| 024 | Either the PRMLIST parameter was not specified or its address is zero. |
| 028 | The user cannot process the SEVER, QUIESCE, or RESUME function with the IPALL bit of the CP IUCV parameter list set to 1. |
| 032 | The path identifier was not specified in the CP IUCV or parameter list. |
| 040 | 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. |
| 044 | Invalid parameter list. |

| Return Code | Meaning |
|---|---|
| 048 | The state of the path is inconsistent with the function the user requested. For example, the user may have issued an IUCVCOM SEND, RECEIVE, REPLY, REJECT, or PURGE instruction for a path before a connection complete interrupt occurred on it. That is, a path upon which a pending CONNECT interrupt has occurred. In such a case, the user should issue an IUCVCOM ACCEPT or SEVER instruction instead. |
| 052 | Either the task that issued the IUCVCOM REP instruction is not the same task that established the path in question, or the IUCVCOM REP instruction was issued by a privileged user. |
| 056 | WAIT=YES can only be specified by privileged IUCV users. |
| 204 | An error occurred in obtaining storage to satisfy the IUCV request. 204 is the return code from the GETMAIN macro. |
| 1xxx | An error occurred. 'xxx' is the value in the IPRCODE field in the IUCV parameter list that describes the error. Consult the section of the *VM/XA SP CP Programming Services* that defines the fields in the IUCV parameter lists. **Note:** 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 issued directly via IUCV. These functions are PURGE, RECEIVE, REJECT, and REPLY. |

| Abend Code | Reason Code | Meaning |
|---|---|---|
| FCA | 1101 | A GETMAIN macro instruction failed when GCS tried to obtain storage on behalf of the task that terminated abnormally. |

## The List Format

```
[label]   IUCVCOM   MF=L
                     [,NAME=label][,PRMLIST=label][,EXIT=label][,UWORD=label]
                                    ┌                  ┐
                                    │,CODE= ( ONE )   │[,PATH=label]
                                    │       <       >  │
                                    │       ( ALL )   │
                                    └                  ┘

                    ┌                                                            ┐
                    │,QUERY                                                      │
                    │                                                            │
                    │┌                                                     ┐     │
                    ││,CONNECT[,EXIT=label][,UWORD=label]                  │     │
                    ││                                                     │     │
                    ││,ACCEPT[,EXIT=label][,UWORD=label]                   │     │
                    ││                                                     │     │
                    ││,SEVER ┌,CODE= ( ONE ) ┐                            │     │
                    ││       │       <       > │                            │     │
                    ││       └       ( ALL ) ┘                            │     │
                    ││                                                     │     │
                    ││,QUIESCE ┌,CODE= ( ONE ) ┐                          │     │
                    ││         │       <       > │                          │     │
                    ││         └       ( ALL ) ┘                          │     │
                    ││                                                     │     │
                    ││,RESUME ┌,CODE= ( ONE ) ┐              [OPTIONS]    │     │
                    ││        │       <       > │                           │     │
                    ││        └       ( ALL ) ┘                           │     │
                    ││                                                     │     │
                    ││,SEND                                                │     │
                    ││                                                     │     │
                    ││,RECEIVE                                             │     │
                    ││,REPLY                                               │     │
                    ││,REJECT                                              │     │
                    ││,PURGE                                               │     │
                    │└                                                     ┘     │
                    │,REP[,EXIT=label][,UWORD=label]┌,CODE= ( ONE ) ┐[,PATH=label]│
                    │                               │       <       > │          │
                    │                               └       ( ALL ) ┘          │
                    │    [,NAME=label]                                           │
                    └                                                            ┘
                    OPTIONS:  [,NAME=addr][,PRMLIST=addr]
```

This format of the macro instruction 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 parameters listed above are valid in the list format of this instruction.

## Added Parameter

**MF=L**
Specifies the list format of this macro instruction.

## The List Address Format

```
[label]    IUCVCOM    MF=(L,address[,label])
                       [,NAME=addr][,PRMLIST=addr][,EXIT=addr][,UWORD=addr]
                                      ⎡,CODE= ⎧ONE⎫⎤[,PATH=addr]
                                      ⎢       ⎨   ⎬⎥
                                      ⎣       ⎩ALL⎭⎦

                      ⎡,QUERY                                                   ⎤
                      ⎢⎡,CONNECT[,EXIT=addr][,UWORD=addr]             ⎤         ⎥
                      ⎢⎢,ACCEPT[,EXIT=addr][,UWORD=addr]              ⎥         ⎥
                      ⎢⎢,SEVER⎡,CODE= ⎧ONE⎫⎤                          ⎥         ⎥
                      ⎢⎢      ⎣       ⎩ALL⎭⎦                          ⎥         ⎥
                      ⎢⎢,QUIESCE⎡,CODE= ⎧ONE⎫⎤                        ⎥         ⎥
                      ⎢⎢        ⎣       ⎩ALL⎭⎦                        ⎥         ⎥
                      ⎢⎢,RESUME⎡,CODE= ⎧ONE⎫⎤               [OPTIONS] ⎥         ⎥
                      ⎢⎢       ⎣       ⎩ALL⎭⎦                         ⎥         ⎥
                      ⎢⎢,SEND                                         ⎥         ⎥
                      ⎢⎢,RECEIVE                                      ⎥         ⎥
                      ⎢⎢,REPLY                                        ⎥         ⎥
                      ⎢⎢,REJECT                                       ⎥         ⎥
                      ⎢⎣,PURGE                                        ⎦         ⎥
                      ⎢,REP[,EXIT=addr][,UWORD=addr]⎡,CODE= ⎧ONE⎫⎤[,PATH=addr]⎥
                      ⎢                             ⎣       ⎩ALL⎭⎦            ⎥
                      ⎣    [,NAME=address]                                      ⎦
                      OPTIONS:  [,NAME=addr][,PRMLIST=addr]
```

This format of the macro instruction 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 issue the instruction using this format, then you must do so before any related invocation of the instruction using the execute format.

Note that only the parameters listed above are valid in the list address format of this instruction.

## Added Parameter

**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 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.

## The Execute Format

```
[label]    IUCVCOM    MF=(E,address)
                       [,NAME=addr][,PRMLIST=addr][,EXIT=addr][,UWORD=addr]
                                   ┌                ┐
                                   │,CODE= { ONE }  │[,PATH=addr][,ERROR=addr]
                                   │       { ALL }  │
                                   └                ┘
                      ┌                                                        ┐
                      │,QUERY[,ERROR=address]                                  │
                      │┌                                                ┐      │
                      ││,CONNECT[,EXIT=addr][,UWORD=addr]                │      │
                      ││                                                 │      │
                      ││,ACCEPT[,EXIT=addr][,UWORD=addr]                 │      │
                      ││                                                 │      │
                      ││       ┌                ┐                        │      │
                      ││,SEVER │,CODE= { ONE }  │                        │      │
                      ││       │       { ALL }  │                        │      │
                      ││       └                ┘                        │      │
                      ││         ┌                ┐                      │      │
                      ││,QUIESCE │,CODE= { ONE }  │                      │      │
                      ││         │       { ALL }  │                      │      │
                      ││         └                ┘                      │      │
                      ││        ┌                ┐                       │      │
                      ││,RESUME │,CODE= { ONE }  │             [OPTIONS] │      │
                      ││        │       { ALL }  │                       │      │
                      ││        └                ┘                       │      │
                      ││,SEND                                            │      │
                      ││                                                 │      │
                      ││,RECEIVE                                         │      │
                      ││,REPLY                                           │      │
                      ││,REJECT                                          │      │
                      ││,PURGE                                           │      │
                      │└                                                ┘      │
                      │                                  ┌              ┐      │
                      │,REP[,EXIT=addr][,UWORD=addr]     │,CODE= { ONE }│[,PATH=addr]
                      │                                  │       { ALL }│      │
                      │                                  └              ┘      │
                      │    [,NAME=address]                                     │
                      └                                                        ┘
                      OPTIONS:  [,NAME=addr][,PRMLIST=addr][,ERROR=address]
```

(ONE is underlined as the default in each CODE= choice.)

This format of the macro instruction generates code that executes the function using a parameter list whose address you specify.

Note that only the parameters listed above are valid in the execute format of this instruction.

## Added Parameter

**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.

## IUCVINI

### Establish or Terminate a Program as an IUCV User

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. By using the GCS IUCV support, communications can take place among several users operating within several tasks operating within several virtual machines.

When the word "user" appears, it should be taken to mean any supervisor or problem program.

Use the IUCVINI macro instruction to either admit a user to or withdraw a user from the IUCV environment.

This treatment of the IUCVINI macro instruction assumes that you are already familiar with the section dealing with IUCV in the *VM/XA SP CP Programming Services*. For more information on IUCV, see "IUCVCOM" on page 398.

The IUCVINI macro instruction is available in standard, list, list address, and execute formats. The ***standard*** format follows.

```
[label]     IUCVINI    { SET,NAME=name,EXIT=addr[,OPTION A][,OPTION B] }
                       { REP,NAME=name,OPTION C[,OPTION B]             }
                       { CLR,NAME=name[,OPTION B]                      }
                       OPTION A:                          OPTION B:

                       [ [UWORD=addr,] PRIV= { NO  }  ]     ERROR=addr
                       [                     { YES }  ]
                       [                              ]
                       [ UWORD=addr [,PRIV= { NO  } ] ]
                       [            [       { YES } ] ]

                       OPTION C:

                       [ [EXIT=addr],UWORD=addr ]
                       [ EXIT=addr[,UWORD=addr] ]
```

### Parameters

**SET**
Indicates that you want the user admitted to the IUCV 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. (All these parameters are described below.) Finally, the user receives permission to establish ownership of the IUCV path over which it will send and receive information.

The user must issue the IUCVINI SET instruction once before it attempts to send or receive information through the IUCV facility. If the SET function completes successfully, register 0 contains the number of possible IUCV 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 using the IUCVCOM instruction, but to communicate on that path using IUCV directly, rather than using the GCS IUCV Support. If necessary, review the entry titled "IUCVCOM" on page 398.

**REP**
Indicates that you want to change the address of the user's general exit routine and/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 upon the situation at the moment. The general exit routine and UWORD specify the manner in 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 issued 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 environment.

When you select this parameter, the ID BLOCK is released and the user's IUCV paths are severed.

**NAME**
Specifies the address of the symbolic name by which the user shall be known within the IUCV 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.

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 exit routine.

This general exit routine receives control each time an IUCV pending connect interrupt occurs on a path associated with this name. An IUCV PENDING

CONNECT interrupt occurs on a path when some other user issues 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 issued 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.

Upon entry to the exit routine, the registers contain the following:

| Register | Contents |
|---|---|
| Register 0 | The UWORD. |
| Register 1 | Unpredictable. |
| Register 2 | The address of the external interrupt buffer. |
| Registers 3 - 12 | Unpredictable |
| Register 13 | The address of a user save area. |
| Register 14 | The address to which control must be returned once the exit routine completes execution. |
| Register 15 | The address of the exit routine. |

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.

Upon entry to the exit routine, the registers contain the following:

| Register | Contents |
|---|---|
| Register 0 | The UWORD. |
| Register 1 | Unpredictable. |
| Register 2 | The address of the external interrupt buffer. |
| Registers 3 - 12 | Unpredictable |
| Register 13 | The address of the 72-byte register save area. |
| Register 14 | The address to which control must be returned once the exit routine completes execution. |

| Register | Contents |
|---|---|
| Register 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). (In the case of the latter, GCS will sever the path involved in the error.) Moreover, 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.

**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 non-privileged. If you omit this parameter, then the user is considered non-privileged, by default. This parameter is valid only when the user issuing this instruction is in supervisor state.

**NO**

Indicates that the user will be non-privileged.

This means that you must use the GCS support macro instructions for all IUCV activities.

**YES**

Indicates that the user will be privileged.

This means that the user has the authority to communicate over a path using IUCV directly, rather than using the IUCVCOM instruction. However, the user must establish and terminate the path using the IUCVCOM instruction. This ensures a proper match between the GCS IUCV path table and the CP IUCV path table.

**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 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). 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.

| Return Code | Meaning |
|---|---|
| 000 | Function completed successfully. |
| 004 | This name is already being used by another IUCV user. |
| 008 | The IUCV facility cannot be used unless the IUCVINI instruction, with the SET parameter specified, is issued first. |
| 016 | Either the NAME parameter was not specified or its address was specified as zero. |
| 020 | Either the EXIT parameter was not specified or its address was specified as zero. |
| 040 | The function requested was unrecognizable by GCS. Specify SET, REP, or CLR. |
| 044 | Invalid parameter list. |
| 052 | Either the user did not issue 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. |
| 204 | An error occurred in obtaining storage to satisfy the IUCV request. 204 is the return code from the GETMAIN macro. |
| 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. Consult the section of *VM/XA SP CP Programming Services* that defines the fields in the IUCV parameter lists. |

| Abend Code | Reason Code | Meaning |
|---|---|---|
| FCA | 1101 | A GETMAIN macro instruction failed when GCS tried to obtain storage on behalf of the task that terminated abnormally. |

## The List Format

```
[label]   IUCVINI   MF=L

                    [[,NAME=label][,EXIT=label][,UWORD=label]     ][,PRIV= { NO  }]
                    [,SET[,NAME=label][,EXIT=label][,UWORD=label] ][       { YES }]
                    [,REP[,NAME=label][,EXIT=label][,UWORD=label] ]
                    [,CLR[,NAME=label]                            ]
```

This format of the macro instruction 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 parameters listed above are valid in the list format of this instruction.

### Added Parameter

**MF = L**
Specifies the list format of this macro instruction.

## The List Address Format

```
[label]   IUCVINI   MF=(L,address[,label])

                    [[,NAME=addr][,EXIT=addr][,UWORD=addr]       ][,PRIV= { NO  }]
                    [ ,SET[,NAME=addr][,EXIT=addr][,UWORD=addr]  ][       { YES }]
                    [ ,REP[,NAME=addr][,EXIT=addr][,UWORD=addr]  ]
                    [ ,CLR[,NAME=addr]                           ]
```

This format of the macro instruction 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 issue the instruction using this format, then you must do so before any related invocation of the instruction using the execute format.

Note that only the parameters listed above are valid in the list address format of this instruction.

### Added Parameter

**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 a user specified label, indicating that the user want to determine the length of the parameter list. The macro expansion equates the label you specify with the length of the parameter list.

## The Execute Format

```
[label]    IUCVINI    MF=(E,address)

                      ┌                                           ┐
                      │[,NAME=addr][,EXIT=addr][,UWORD=addr]      │  ┌            ┐
                      │[,ERROR=addr]                              │  │,PRIV= {NO } │
                      │                                           │  │       {YES} │
                      │                                           │  └            ┘
                      │SET[,NAME=addr][,EXIT=addr]                │
                      │[,UWORD=addr][,ERROR=addr]                 │
                      │                                           │
                      │REP[,NAME=addr][,EXIT=addr]                │
                      │[,UWORD=addr][,ERROR=addr]                 │
                      │                                           │
                      │CLR[,NAME=addr][,ERROR=addr]               │
                      └                                           ┘
```

This format of the macro instruction generates code that executes the function using a parameter list whose address you specify.

Note that only the parameters listed above are valid in the execute format of this instruction.

### Added Parameter

**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.

# Chapter 15. Build Macros

## AUTHUSER

### Specify the Authorized Users in a Virtual Machine Group

The GROUP CONFIGURATION file describes a GCS virtual machine group. This file is divided into three data blocks.

| | |
|---|---|
| **The Configuration Block** | Defines the virtual machine group's configuration so that it conforms to the needs of your installation. Review the entry titled "CONFIG" on page 424. |
| **The Segment Block** | Identifies which saved segments will be automatically linked to each member of the group at IPL time. Review the entry titled "SEGMENT" on page 430. |
| **The Authorized User Block** | Identifies which members of the group are "authorized." That is, which members are permitted to perform authorized GCS functions. This book describes the macros associated with these functions in the section titled Chapter 9, "Authorized GCS Service Macros" on page 191. |

Use the AUTHUSER macro instruction to create an authorized user block for the GROUP CONFIGURATION file.

The format of the AUTHUSER macro instruction is:

```
AUTHUSER     { START       }
             { NAME=userid }
             { END         }
```

### Parameters

**START**
Indicates that this AUTHUSER instruction marks the beginning of the authorized user block.

The authorized user block must begin with an AUTHUSER instruction with this parameter specified.

**NAME**
Specifies the userid of the virtual machine that is to have authorized status.

**END**
Indicates that this AUTHUSER instruction marks the end of the authorized user block.

The authorized user block must end with an AUTHUSER instruction with this parameter specified.

## Usage Notes

- Most installations will not explicitly use the CONFIG, SEGMENT, and AUTHUSER macro instructions 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, invoked by the GROUP command, eliminate the need to build the file by explicitly coding these macros. When you invoke the GROUP command without the use of a full-screen terminal, your file will have to be built using the editor and coding the macro instructions manually.
- The GROUP CONFIGURATION file adopts the system name as its filename. This name corresponds exactly with that specified in the SYSNAME parameter of the CONFIG instruction. The filetype of the GROUP CONFIGURATION file is always GROUP.
- Remember that in using the CONFIG, SEGMENT, and AUTHUSER instructions you are creating blocks of information. Thus, all occurrences of the AUTHUSER instruction must be physically grouped together in the GROUP CONFIGURATION file. The same is true of the CONFIG and SEGMENT macro instructions.

## Example

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 instruction 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 instruction with the END parameter specified.

## Return Codes and Abend Codes

The AUTHUSER macro generates no return codes and no abend codes.

## CONFIG

### Define the Configuration of a GCS System (Virtual Machine Group)

The GROUP CONFIGURATION file describes a GCS virtual machine group. This file is divided into three data blocks.

| | |
|---|---|
| **The Configuration Block** | Defines the virtual machine group's configuration so that it conforms to the needs of your installation. |
| **The Segment Block** | Identifies which saved segments will be automatically linked to each member of the group at IPL time. Review the entry titled "SEGMENT" on page 430. |
| **The Authorized User Block** | Identifies which members of the group are "authorized." That is, which members are permitted to perform authorized GCS functions. This book describes the macros associated with these functions in the section titled Chapter 9, "Authorized GCS Service Macros" on page 191. Also, review the entry titled "AUTHUSER" on page 422. |

Use the CONFIG macro instruction to define the configuration of your GCS virtual machine group.

The format of the CONFIG macro instruction is:[14]

```
CONFIG    SYSNAME=name,MAXVM=number,RECVM=userid[,SDISK= {address}]
                                                         {595    }

          [,YDISK={address}] [,TABSIZE={number}] [,DUMPVM={userid}]
                  {59E    }            {16    }           {*     }

          [,SYSID='character string']
```

### Parameters

**SYSNAME**

The name under which this system (group) will be saved.

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 filename of the GROUP CONFIGURATION file associated with the group. Likewise, this name must correspond with the name of the saved system as entered in the system name table.

[14] Note that 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.

You can write this as any string of alphameric 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 the group whose configuration is being defined.

You can write this parameter as any number from 1 to 65535.

**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 on behalf of any virtual machine that is reset. A virtual machine that is "reset" is one that has logged-off, been re-IPLed, etc. "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 by IPLing the GCS named saved system. Otherwise, an error will occur and the system will be reset.

Write this parameter as the user ID of the recovery machine.

**SDISK**
Specifies the virtual address of the system disk (S-DISK) that all machines in the group will attempt to access.

If you omit this parameter, the virtual address of the S-DISK will be 595, by default.

**YDISK**
Specifies the virtual address of the Y-DISK that is an extension of the S-DISK and that all machines in the group will attempt to access.

If you omit this parameter, the virtual address of the Y-DISK will be 59E, by default.

**TABSIZE**
Specifies the size of the GCS trace table that the virtual machine group will use. The tabsize is allocated from GCS common storage.

The GCS supervisor records all supervisor events in this trace table. Moreover, users have the option to record user virtual machine events in the same table via the ITRACE command and GTRACE macro instruction. For more information on the ITRACE and GTRACE commands, see "GTRACE" on page 234 and "GCS Commands" on page 20.

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 userid of the virtual machine that is to receive all dumps requested by any member of the group.

It is strongly recommended that the userid specified by this parameter refer to an authorized user. Remember that dumps often contain fetch-protected, non-key-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 userid of the virtual machine you wish to designate as the recipient of all dumps.

**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 that the character string must be surrounded by single quotation marks.[15] If you omit this parameter, then the text of the system id will be a blank line, by default.

## Usage Notes

- Most installations will not explicitly use the CONFIG, SEGMENT, and AUTHUSER macro instructions 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, invoked 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 macro instructions manually.
- The GROUP CONFIGURATION file adopts the system name (virtual machine group name) as its filename. Its filetype is always GROUP.
- Remember that in using the CONFIG, SEGMENT, and AUTHUSER instructions you are creating blocks of information. Thus, all occurrences of the CONFIG instruction must be physically grouped together in the GROUP CONFIGURATION file. The same is true of the AUTHUSER and SEGMENT macro instructions.

## Example

In the following example, the name of the system being described is MAIN. It is under this name that the system or virtual machine group is to be saved.

```
CONFIG SYSNAME=MAIN,MAXVM=5,RECVM=VM1,DUMPVM=VM1,SYSID='WELCOME!'
```

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 to be displayed to each user at IPL time.

## Return Codes and Abend Codes

The CONFIG macro generates no return codes and no abend codes.

[15] 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.

# CONTENTS

## 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 instruction to create such a directory (also called a CONTENTS MODULE) for a saved segment.

The format of the CONTENTS macro instruction is:

```
CONTENTS     START

             NAME=name[,EP=entry point]  ⎡,OL= ⎧ YES ⎫⎤
                                         ⎣     ⎩ NO  ⎭⎦

             END
```

## Parameters

**START**
Indicates that this CONTENTS instruction marks the beginning of the CONTENTS MODULE. In addition, 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 instruction or the OSRUN command is issued. For more information see "ATTACH" on page 86, "LINK" on page 137, "XCTL" on page 152, "LOAD" on page 142, or "GCS Commands" on page 20.

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.
- The 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.

**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 instruction is the only macro instruction that can be used on such code. Instructions 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.

**END**
Indicates that this CONTENTS instruction marks the end of the CONTENTS MODULE. What follows, then, is the first module in the saved segment.

## Usage Notes

- Each saved segment must begin with a CONTENTS MODULE. The first example below illustrates the contents of such a module.
- 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.
- All entry point names in a particular saved segment must be unique. Moreover, 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.

## Example

The code in the example below 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
         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 instruction with the START parameter specified. Then the entry points are defined.

Of particular interest is the fact that the second through fourth CONTENTS instructions have both the NAME and EP parameters specified. PROG1, PROG2, and PROG3 are defined as aliases for PROGRAMA. If an external program invokes any of these names, then control will pass to the code at the entry point named PROGRAMA.

Note that neither of the entry points named PROGRAMB and PROGRAMC has an alias defined for it. Thus, to invoke either of these, an external program would have to use either the name PROGRAMB or PROGRAMC.

Further, the entry point DATA is defined as containing code that is ONLY LOADABLE.

Finally, the CONTENTS MODULE is concluded with a CONTENTS instruction with the END parameter specified.

## Return Codes and Abend Codes

The CONTENTS macro generates no return codes and no abend codes.

# SEGMENT

## 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 three data blocks.

| | |
|---|---|
| **The Configuration Block** | Defines the virtual machine group's configuration so that it conforms to the needs of your installation. Review the entry titled "CONFIG" on page 424. |
| **The Segment Block** | Identifies which saved segments will be automatically linked to each member of the group at IPL time. |
| **The Authorized User Block** | Identifies which members of the group are "authorized." That is, which members are permitted to perform authorized GCS functions. This book describes the macros associated with these functions in the section titled Chapter 9, "Authorized GCS Service Macros" on page 191. Also, review the entry titled "AUTHUSER" on page 422. |

Use the SEGMENT macro instruction to create a segment block for the GROUP CONFIGURATION file.

The format of the SEGMENT macro instruction is:

```
SEGMENT    { START             }
           { NAME=segment name }
           { END               }
```

## Parameters

**START**
Indicates that this SEGMENT instruction marks the beginning of the segment block.

The segment block must begin with a SEGMENT instruction with this parameter specified.

**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 instruction marks the end of the segment block.

The segment block must end with a SEGMENT instruction with this parameter specified.

## Usage Notes

- Most installations will not explicitly use the CONFIG, SEGMENT, and AUTHUSER macro instructions 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, invoked 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 macro instructions manually.
- The GROUP CONFIGURATION file adopts the system name (virtual machine group name) as its filename. Its filetype is always GROUP.
- Remember that in using the CONFIG, SEGMENT, and AUTHUSER instructions you are creating blocks of information. Thus, all occurrences of the SEGMENT instruction must be physically grouped together in the GROUP CONFIGURATION file. The same is true of the CONFIG and AUTHUSER macro instructions.

## Example

The following 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 instruction with the START parameter specified. 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 instruction with the END parameter specified.

## Return Codes and Abend Codes

The SEGMENT macro generates no return codes and no abend codes.

# Chapter 16. Data Areas Macros

## CVT

### Simulate the OS Communication Vector Table

When using VTAM under GCS, a communication vector table is required.

Use the CVT macro instruction to simulate the communication vector table in your virtual machine group's common storage.

The format of the CVT macro instruction is:

| | |
|---|---|
| CVT | |

### Parameters

The CVT macro instruction accepts no parameters.

### Usage Notes

- The simulated CVT is in common storage following the GCS supervisor code. Only certain CVT fields are required to support the interface between GCS and VTAM.
- Within each member of your virtual machine group, the address of the CVT resides at location X'10'.
- The following table illustrates the format of the CVT, as simulated by GCS.

| Address | Field |
|---|---|
| -8 (X'-8') | RESERVED |
| 116 (X'74') | CVTDCB |
| 120 (X'78') | RESERVED |
| 256 (X'100') | CVTCBSP |
| 260 (X'104') | RESERVED |
| 328 (X'148') | CVTEXT2 |
| 332 (X'14C') | RESERVED |
| 348 (X'15C') | END |

- Bit 1 of the CVTDCB field contains 0 when under the GCS supervisor. Under CMS, it contains 1.

- The CVTCBSP field contains the address of a control block that contains the addresses of the VSAM/VTAM manipulative macro routines. The address of this control block is loaded via code generated by the manipulative macros themselves. Thus, it is necessary to maintain its address for object level VSAM and VTAM API compatibility.
- The CVTEXT2 field contains the address of the common extension. CVTEXT is used strictly by VTAM.

## Return Codes and Abend Codes

The CVT macro instruction generates no return codes and no abend codes.

## FLS

### 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.

Use the FLS macro instruction to gain access to these fields.

The format of the FLS macro instruction is:

| | |
|---|---|
| FLS | |

### Parameters

The FLS macro instruction accepts no parameters.

### Usage Notes

- The FLS macro instruction gives you access to the following fields residing in page 0 of your virtual storage:

| | |
|---|---|
| **FLSVTAM** | A fullword made available to various application programs, such as VTAM. |
| **FLSVMID** | The userid associated with your virtual machine. |
| **FLSLVL** | A fullword that contains the release number and service level of your GCS system. If you wish, you can use the GCSLEVEL macro instruction to map this field. Review the entry titled "GCSLEVEL" on page 438. |
| **FLSSIGID** | The signal services machine id used to communicate with CP signal services. This field is a halfword, stored in binary format. |
| **FLSATID** | The task id of the active task. This field is a halfword, stored in binary format. |
| **FLSATB** | The address of the active task block. |
| **FLSPOST** | The branch entry address for the POST macro. If necessary, review the entry titled "POST" on page 116. |
| **FLSCTB** | The address of the trace block. |

- The following table illustrates the format of the FLS fields:

| Address | Field | | |
|---|---|---|---|
| 0 (X'0') | RESERVED | | |
| 512 (X'200') | FLSVTAM | | |
| 516 (X'204') | FLSVMID | | |
| 524 (X'20C') | FLSLVL | | |
| 528 (X'210') | FLSSIGID | -- | FLSATID |
| 532 (X'214') | FLSATB | | |
| 536 (X'218') | FLSPOST | | |
| 540 (X'21C') | FLSCTB | | |
| 544 (X'220') | RESERVED | | |
| 552 (X'228') | END | | |

## Return Codes and Abend Codes

The FLS macro instruction generates no return codes and no abend codes.

# GCSLEVEL

## Map the Contents of 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. For more information on gaining access to these fields see "FLS" on page 436.

One of these fields—the FLSLVL field—accommodates the release number and service level of the GCS system you are using.

Use the GCSLEVEL macro instruction to map the release number and service level of your GCS system from the FLSLVL field.

The format of the GCSLEVEL macro instruction is:

| | |
|---|---|
| GCSLEVEL | |

## Parameters

The GCSLEVEL macro instruction accepts no parameters.

## Usage Notes

- The number of bytes for each subfield are in parentheses. The format of the FLSLVL field is:

| | | |
|---|---|---|
| RESERVED FOR FUTURE USE (1) | RELEASE NUMBER (1) | SERVICE LEVEL (2) |

- The subfield containing the release number actually contains a code that represents the release number. In the case of GCS, the RELEASE NUMBER field contains 02, which stands for VM/XA SP RELEASE 2.
- The SERVICE LEVEL information is a halfword, stored in binary format.

## Return Codes and Abend Codes

The GCSLEVEL macro instruction generates no return codes and no abend codes.

# Appendix. The System Product Interpreter

## The System Product Interpreter in the GCS Environment

Most REXX capabilities available in the CMS environment are also available in the GCS environment. You can use such REXX capabilities as instructions, functions, expressions, and operators. There are, however, some differences between writing REXX programs for the GCS environment and writing REXX programs for the CMS environment.

The differences in the GCS environment are as follows:

1. EXECs normally reside in CMS-formatted disk files and have a filetype of GCS. The GCS filetype can be overridden by using the FILEBLK parameter on the CMDSI macro. If necessary, consult *VM/XA SP System Product Interpreter User's Guide* or *VM/XA SP System Product Interpreter Reference* for more information on FILEBLK parameter.
2. GCS does not support the following immediate commands: TS, TE, and HI.
3. An EXEC written for the GCS environment should not have the same name as an immediate command. Immediate commands are higher in the search order; therefore, an immediate command would be executed before an EXEC. An EXEC written for the GCS environment with the same name as an immediate command would never get executed.
4. GCS does not support the external function libraries: RXSYSFN, RXLOCFN, and RXUSERFN. However, GCS does support external function calls. These functions and subroutines must be written in the REXX language.
5. The GCS CMDSI macro ("CMDSI" on page 175) can be used to invoke REXX programs from Assembler language programs. The FILEBLK parameter on the CMDSI macro contains the address of the file block. FILEBLK is useful for executing in-storage EXECs, executing EXECs with filetypes other than GCS, and establishing an initial subcommand environment.
6. The default ADDRESS environment of REXX is GCS.

   ADDRESS GCS specifies that full command resolution is in effect. With full command resolution, first search for an EXEC with the given name. If such an EXEC does not exist, then invoke the given name using SVC 202. If the above fails, search for a CP command with the given name.

   ADDRESS COMMAND searches for host commands (GCS commands).
7. GCS does not have a terminal input buffer. If you issue a PULL instruction and the program stack is empty, the WTOR macro generates a read to the console.
8. Each task has its own program stack. Therefore, data in a program stack can be shared among EXECs running in the same task.
9. To specify other subcommand environments in GCS, you must use LOADCMD. LOADCMD defines a command name to the requested module of a CMS load library and loads this command module into storage. Therefore, GCS can call the requested command module when a command is entered at the console or submitted by a program with the CMDSI macro.

   GCS does not support non-SVC fast path subcommand invocation.
10. The SIGNAL ON HALT instruction has no effect in GCS.

## Processing EXECs in GCS (CSIREX module)

All EXEC processing in GCS is routed to the GCS module, CSIREX. CSIREX is the external interface for the System Product Interpreter (CSIRIN).

SVC 202 calls CSIREX with the contents of the registers as follows:

| Register | Contents |
|---|---|
| R0 | Address of the extended parameter list |
| R1 | Address of the standard tokenized parameter list |
| R12 | Address of the entry point |
| R13 | Address of a register save area |
| R14 | Return address |
| R15 | Address of the entry point (same as R12) |

### The Extended Plist

The extended plist has the following format:

```
EPLIST   DSECT
EPLCMD   DS    A      Address of command token
EPLARGBG DS    A      Address of beginning of arguments
EPLARGND DS    A      Address of byte following the end
*                     of arguments
EPFBL    DS    A      Address of the file block
EPARGLST DS    A      Address of function argument list
*                     for EXEC
EPFUNRET DS    A      Address for return of function data
*                     for EXEC

EPLIND   DS    X      Indicator
EPLPGM   EQU   X'00' Program issued command
EPLACMD  EQU   X'01' Call from System Product Interpreter
*                     when ADDRESS COMMAND is specified
EPLFNC   EQU   X'05' Subroutine/function call
EPLCONS  EQU   X'0B' Console command
EPLRESVD DS    3X     Reserved
```

### The Standard Tokenized Plist

The standard tokenized plist has the following format:

```
         DC    CL8'EXEC'
         DC    CL8'execname'
         DC    XL8'FF'
```

## The File Block

The file block has the following format:

```
FBLOCK   DSECT
FBLNAME  DS    CL8   Program name (usually EXEC filename)
FBLTYPE  DS    CL8   Program type/default prefix
*                    (usually GCS filetype)
FBLMODE  DS    CL2   Program filemode
FBLEXTL  DS    H     Extension block length in fullwords
FBLEXT   EQU   *     Extension block starts here
*  The next 2 words represent the start
*  and end of in-storage EXECs
FBLDLS   DS    AL4   Descriptor list starts here
FBLDLE   DS    AL4   Descriptor length
FBLPREF  DS    CL8   Explicit initial prefix
```

## EXECCOMM Processing (Sharing Variables)

The EXECCOMM macro allows programs to access and manipulate the current generation of REXX variables. These variables may be inspected, set, or dropped. To use the EXECCOMM capability, a REXX program must be active on the current task.

The standard format of the EXECCOMM macro instruction is:

| | | |
|---|---|---|
| [label] | EXECCOMM | REQLIST=addr |

## Parameters

**REQLIST**
is an RX-type address or register. addr specifies the address of the shared variable request block chain. Each caller is responsible for setting up its variable request block chain.

**Note:** The internal REXX work areas are manipulated by the System Product Interpreter's own routines. Therefore, the user's program does not need to know the structure of the variable's access method.

The EXECCOMM macro generates an SVC 203, and the register input for EXECCOMM processing is as follows:

| Register | Contents |
|---|---|
| R0 | Shared variable request block chain pointer |
| R12 | Entry point address |
| R13 | Save area address |
| R14 | Return address |
| R15 | Entry point address |

On return from the SVC 203, register 15 contains the return codes. The possible return codes are:

| Return Code | Meaning |
|---|---|
| 0 or positive | The entire request list was processed. |
| -1 | Invalid entry condition (no REXX program active on this task) has occurred. |
| -2 | Insufficient storage is available to process the request. |

## Shared Variable Request Block

If the address of the shared variable request block passed in register 0 is invalid, the task is terminated with abend code FCB and reason code 0D01. Each request block in the chain must be structured as follows:

```
************************************************************
SHVBLOCK DSECT
SHVNEXT  DS    A     Chain pointer to next element or 0
SHVUSER  DS    F     Used during "Fetch Next"
SHVCODE  DS    CL1   Individual function code
SHVRET   DS    XL1   Individual return code flags
         DS    H'0'  Not used
SHVBUFL  DS    F     Length of 'Fetch' value buffer
SHVNAMA  DS    A     Address of variable name
SHVNAML  DS    F     Length of variable name
SHVVALA  DS    A     Address of value buffer
SHVVALL  DS    F     Length of value (set on 'Fetch')
*
*  Function Codes (SHVCODE):
*
SHVSET   EQU   C'S'  Set variable from given value
SHVFETCH EQU   C'F'  Copy value of variable to buffer
SHVDROPV EQU   C'D'  Drop variable
SHVSYSET EQU   C's'  Symbolic name Set variable
SHVSYFET EQU   C'f'  Symbolic name Fetch variable
SHVSYDRO EQU   C'd'  Symbolic name Drop variable
SHVNEXTV EQU   C'N'  Fetch 'Next' variable
SHVPRIV  EQU   C'P'  Fetch private information
*
*  Return Codes (SHVRET)
*
SHVCLEAN EQU   X'00' Execution was OK
SHVNEWV  EQU   X'01' Variable did not exist
SHVLVAR  EQU   X'02' Last variable transferred (for 'N')
SHVTRUNC EQU   X'04' Truncation occurred during 'Fetch'
SHVBADN  EQU   X'08' Invalid variable name
SHVBADV  EQU   X'10' Reserved in REXX
SHVBADF  EQU   X'80' Invalid function code (SHVCODE)
************************************************************
```

A typical calling sequence using the EXECCOMM macro is:

```
EXECCOMM REQLIST=(5)
```

where register 5 points to the first of a chain of one or more request blocks.

## Function codes (SHVCODE)

Function codes may be given in lowercase or uppercase.

**Lowercase** (The **symbolic** interface). The names must be valid REXX symbols (in mixed case if desired), and normal REXX substitution will occur in compound variables.

**Uppercase** (The **direct** interface). No substitution or case translation takes place. Simple symbols must be valid REXX variable names (that is, in uppercase, and not starting with a digit or a period). Compound symbols must contain a valid REXX stem. However, **any** characters are permitted (including blanks and lowercase characters, for example) following this valid stem.

**Note:** The **direct** interface should be used in preference to the **symbolic** interface whenever generality is desired.

The function codes S, F, and D may be given in lowercase or uppercase. The specific actions for each function code are as follows:

| **Function Code** | **Action** |
|---|---|
| S and s | Set variable. The SHVNAMA/SHVNAML adlen describes the name of the variable to be set, and SHVVALA/SHVVALL describes the value that is to be assigned to it. The name is validated to ensure that it does not contain invalid characters. The variable is then set from the value given. If the name is a stem, all variables with that stem are set, just as though this were a REXX assignment. SHVNEWV is set if the variable did not exist before the operation. |
| F and f | Fetch variable. The SHVNAMA/SHVNAML adlen describes the name of the variable to be fetched. SHVVALA specifies the address of a buffer into which the data is to be copied, and SHVBUFL contains the length of the buffer. The name is validated to ensure that it does not contain invalid characters, and the variable is then located and copied to the buffer. The total length of the variable is put into SHVVALL, and, if the value was truncated (because the buffer was not big enough), the SHVTRUNC bit is set. If the variable is shorter than the length of the buffer, no padding takes place. If the name is a stem, the initial value of that stem (if any) is returned.<br><br>SHVNEWV is set if the variable did not exist before the operation, and in this case the value copied to the buffer is the derived name of the variable (after substitution, for example). |

| Function Code | Action |
|---|---|
| **D and d** | Drop variable. The SHVNAMA/SHVNAML adlen describes the name of the variable to be dropped. SHVVALA/SHVVALL are not used. The name is validated to ensure that it does not contain invalid characters, and the variable is then dropped, if it exists. If the name given is a stem, all variables starting with that stem are dropped. SHVNEWV is set if no variables were affected by the operation. |

The function codes N and P should always be given in uppercase. The specific actions for each function code are as follows:

| Function Code | Action |
|---|---|
| N | Fetch Next variable. This function may be used to search through all the variables known to the interpreter (that is, all those of the current generation, excluding those "hidden" by PROCEDURE instructions). The order in which the variables are revealed is not specified.<br><br>The interpreter maintains a pointer to its list of variables. The pointer is reset to point to the first variable in the list when one of the following takes place:<br><br>• A host command is issued using<br>• Any function other than N function is executed using EXECCOMM.<br><br>Whenever an N (Next) function is executed, the name and value of the next variable available are copied to two buffers supplied by the caller.<br><br>SHVNAMA specifies the address of a buffer into which the name is to be copied, and SHVUSER contains the length of that buffer. The total length of the name is put into SHVNAML, and, if the name was truncated because the buffer was not big enough, the SHVTRUNC bit is set. If the name is shorter than the length of the buffer, no padding takes place. The value of the variable is copied to the user's buffer area using exactly the same protocol as for the fetch operation.<br><br>If SHVLVAR is set in SHVRET, the end of the list of known variables has been found, the internal pointers have been reset, and no valid data has been copied to the user buffers. If SHVTRUNC is set, either the name or the value has been truncated.<br><br>By repeatedly executing the N function until the SHVLVAR flag is set, a user program can locate all the REXX variables of the current generation. |

| Function Code | Action |
|---|---|
| **P** | Fetch private information. This function is identical to the F fetch function, except that the name refers to certain fixed information items that are available. Only the first letter of each name is checked (though callers should supply the whole name). The following names are recognized:<br><br>**ARG** Fetch primary argument string. The first argument string that would be parsed by the ARG instruction is copied to the user's buffer.<br><br>**SOURCE** Fetch source string. The source string, as described for PARSE SOURCE on page /REF37/, is copied to the user's buffer.<br><br>**VERSION** Fetch version string. The source string, as described for PARSE VERSION on page /REF38/, is copied to the user's buffer. |

# Glossary

## A

**automatic software re-IPL**. The process by which the control program attempts to restart the system after abnormal termination. This process does not involve the hardware IPL process. See also virtual = real machine recovery.

## C

**CCS**. Console communication services.

**CCW**. Channel command word.

**channel command word (CCW)**. A doubleword structure that directs an I/O operation on a device or channel and includes pointers to any storage areas associated with the operation. One or more CCWs make up a channel program.

**CMS**. Conversational monitor system.

**console communication services (CCS)**. A group of CP routines that interface with the VTAM service machine, providing full VM/XA™ SP console capabilities for SNA/CCS terminal users.

**control program (CP)**. The component of VM/XA SP that manages the resources of a single System/370-Extended Architecture system so that multiple computing systems appear to exist. Each virtual machine is the functional equivalent of either a System/370 computing system or a System/370-Extended Architecture computing system.

**conversational monitor system (CMS)**. The component of VM/XA SP that, as a virtual machine operating system, provides interactive time-sharing. CMS allows users to communicate with the system and with each other, to create and edit files, and to develop and run application programs. It operates in either System/370 mode or 370-XA mode under the control of CP.

**CP**. Control program.

## D

**DCSS**. Discontiguous saved segment.

**directory**. A CP disk file that includes an entry for each user in the system. The entry defines the characteristics of the user's initial virtual machine configuration. These characteristics include the userid, the password, normal and maximum allowable virtual storage, virtual device definitions, the privilege class, the dispatching priority, logical line editing characters, and the account number.

**discontiguous saved segment (DCSS)**. A saved segment that occupies one or more architecturally-defined segments. It begins and ends on segment boundaries. It is accessed by its own name. Contrast with member saved segment. See also saved segment, segment, segment space.

**dump viewing facility**. A VM/XA SP component that allows users to display, format, and print data interactively from CP hard and soft abend, stand-alone, and virtual machine dumps, and to process CP trace table data stored on tape in a system trace file or data I/O and guest trace in a system trace file.

**dynamic paging area**. The area of real storage allocated by CP for V = V machine paging. This area also contains CP nonresident modules, CP control blocks, CP trace tables, free storage pages, and the alternate processor's prefix storage areas.

## E

**Expanded Storage**. Optional integrated high-speed storage. In VM/XA SP, Expanded Storage may be shared by CP and one or more virtual machines. It may also be dedicated to CP or to a particular virtual machine.

VM/XA is a trademark of the International Business Machines Corporation.

## F

**full-pack minidisk**. A virtual disk that contains all of the addressable cylinders of a real DASD volume.

**full-screen mode**. In VM/XA SP, the environment in which an entire 3270 display screen is under the control of a program running in a virtual machine.

## G

GCS. Group control system.

**group control system (GCS)**. The component of VM/XA SP that, as a virtual machine supervisor, executes in a group of System/370 virtual machines under CP control to provide an interface that helps support a native Systems Network Architecture (SNA) network.

**guest**. An operating system running in a virtual machine managed by the VM/XA SP control program. Contrast with host.

**guest real storage**. The storage that appears real to the operating system running in a virtual machine. Contrast with guest virtual storage, host real storage, and host virtual storage.

**guest virtual storage**. The storage that appears virtual to the operating system running in a virtual machine. Contrast with guest real storage, host real storage, and host virtual storage.

## H

**host**. The VM/XA SP control program in its capacity as manager of a virtual machine in which another operating system is running. Contrast with guest.

**host real storage**. The storage that appears real to the control program. If VM/XA SP is running native, this is real storage; if VM/XA SP is running in a virtual machine, this is virtual storage. Contrast with guest real storage, guest virtual storage, and host virtual storage.

**host virtual storage**. The storage that appears virtual to the control program. Contrast with guest real storage, guest virtual storage, and host real storage.

## I

**image library**. A set of modules, contained in a system data file, that define the spacing, characters, and copy modification data that a 3800 printer uses to print a spool file or that define the spacing and character set that an impact printer uses to print a spool file. See also system data file.

**inter-user communication vehicle (IUCV)**. A generalized CP interface that facilitates the transfer of data among virtual machines.

**IUCV**. Inter-user communication vehicle.

## M

**member saved segment**. A saved segment that begins and ends on a page boundary. It belongs to up to 64 segment spaces and is accessed either by the segment space name or by its own name. Contrast with discontiguous saved segment. See also saved segment, segment, segment space.

**message repository file**. A type of system data file that contains a set of VM/XA SP messages translated into a national language.

**missing interrupt handler**. A CP function for detecting and dealing with real I/O operations that do not complete within a specified time.

**multiple preferred guests**. A VM/XA SP facility that supports up to six preferred virtual machines when the Processor Resource/Systems Manager™ (PR/SM™) feature is installed in the real machine. See also preferred virtual machine.

Processor Resource/Systems Manager and PR/SM are trademarks of the International Business Machines Corporation.

## N

**named saved system (NSS)**. A copy of an operating system that a user has named and retained in a system data file. The user can load the operating system by its name, which is more efficient than loading it by device number. See also discontiguous saved segment, member saved segment, saved segment, segment space, system data file.

**NSS**. Named saved system.

## P

**pageable virtual machine**. Synonymous with virtual = virtual machine.

**preferred virtual machine**. A virtual machine that runs in the V = R area. CP gives this virtual machine preferred treatment in the areas of performance, processor assignment, and I/O interrupt handling. See also multiple preferred guests, virtual = fixed machine, virtual = real area, virtual = real machine.

**Processor Resource/Systems Manager (PR/SM)**. A separately orderable feature available with 3090E processors that provides for logical partitioning of the real machine and support of multiple preferred guests. See also multiple preferred guests.

**PR/SM**. Processor Resource/Systems Manager.

## R

**real system operator**. Any user who loads and runs VM/XA SP in the real machine. Contrast with virtual machine operator.

## S

**saved segment**. One or more pages of storage that have been named and retained in a system data file. See also discontiguous saved segment, member saved segment, segment, segment space, system data file.

**segment**. In System/370 architecture, 64 kilobytes of storage. In 370-XA architecture, 1 megabyte of storage. See also saved segment.

**segment space**. A saved segment composed of up to 64 member saved segments accessed by a single name. A segment space occupies one or more architecturally-defined segments; it begins and ends on segment boundaries. A user with access to a segment space has access to all of its members. See also discontiguous saved segment, member saved segment, saved segment, segment.

**service virtual machine**. A virtual machine that provides system services. These services include accounting, error recording, monitoring, and those provided by supported licensed programs.

**SMSG function**. A CP function that allows a virtual machine to send a special message to another virtual machine programmed to accept and process the message. See also special message.

**SNA**. Systems Network Architecture.

**SNA/CCS terminal**. Any terminal accessing VM/XA SP that is managed by a VTAM service machine.

**special message**. A data transmission, made up of instructions or commands, sent from one virtual machine to another via the SMSG function. A special message is processed by the receiving virtual machine and does not appear on the receiver's console. See also SMSG function.

**spool file**. A collection of data along with CCWs for processing on a unit record device. Contrast with system data file.

**SVC 76**. In VM/XA SP, a supervisor call instruction that records the error incidents encountered by certain operating systems running in virtual machines. When a virtual machine operating system issues an SVC 76, VM/XA SP translates the virtual storage and I/O device addresses to real addresses, records the information on the VM/XA SP error recording virtual machine, and returns control to the issuing virtual machine. This interface bypasses the virtual machine's own error recording routine, and avoids duplicate error recording.

**System/370 mode**. A virtual machine operating mode in which System/370 functions are simulated. Contrast with 370-XA mode.

**system data file**. A collection of data associated with a particular function. Types of system data files include saved segments, NSSs, UCR files, image libraries, message repository files, and system trace files. Because a system data file contains no CCWs, it cannot be processed on a unit record device. Contrast with spool file.

**system hold status**. A spool file status that prevents a file from being printed, punched, or read until the real system operator releases it. Contrast with user hold status.

**system trace file**. A type of system data file that contains CP or virtual machine trace data.

**Systems Network Architecture (SNA)**. The description of the logical structure, formats, protocols, and operational sequences for transmitting information units through, and controlling the configuration and operation of, networks.

## U

**UCR file**. User class restructure file.

**unit record device**. A reader, a printer, or a punch.

**user class restructure file (UCR file)**. A type of system data file that contains information used to override the IBM-defined privilege class structure of CP commands, DIAGNOSE instruction codes, and certain CP system functions.

**user directory**. See directory.

**user hold status**. A spool file status that prevents a file from being printed, punched, or read until the file owner releases it. Contrast with system hold status.

## V

**Vector Facility (VF)**. A hardware feature that provides synchronous instruction processing for high-speed manipulation of fixed-point and floating-point data.

**VF**. Vector Facility.

**V=F machine**. Virtual=fixed machine.

**virtual=fixed machine (V=F machine)**. A preferred virtual machine with a fixed, contiguous area of host real storage that does not start at page 0. CP provides performance enhancements for this virtual machine. See also multiple preferred guests, preferred virtual machine, virtual=real area, virtual=real machine, virtual=virtual machine.

**virtual machine**. In VM/XA SP, a functional equivalent of either a System/370 computing system or a System/370-Extended Architecture computing system. Each virtual machine is controlled by an operating system. VM/XA SP controls the concurrent execution of multiple virtual machines on an actual System/370-Extended Architecture system.

**Virtual Machine/Extended Architecture™ System Product (VM/XA SP)**. An operating system that allows multiple IBM System/370 and 370-XA operating systems to run simultaneously on a single 370-XA processor. The multiple systems may be used for production, testing, developing application programs, maintenance, and migration. VM/XA SP also provides a high-capacity interactive environment. There are four components: the control program (CP), the conversational monitor system (CMS), the dump viewing facility, and the group control system (GCS).

**virtual machine operator**. Any user who loads and runs an operating system in a virtual machine. Contrast with real system operator.

**virtual=real area (V=R area)**. A fixed, contiguous section of real storage, starting at page 0, in which preferred virtual machines execute. CP does not page this storage. See also preferred virtual machine, virtual=fixed machine, virtual=real machine.

**virtual=real machine (V=R machine)**. A preferred virtual machine with a fixed, contiguous area of host real storage that starts at page 0. CP provides performance enhancements and an automatic recovery facility for this virtual machine. See also multiple preferred guests, preferred virtual machine, virtual=real area, virtual=real machine recovery, virtual=virtual machine.

**virtual=real machine recovery (V=R machine recovery)**. A CP function that allows the V=R machine to resume operation after most CP

Virtual Machine/Extended Architecture is a trademark of the International Business Machines Corporation.

abnormal terminations. When possible, the facility reestablishes the V = R machine environment, allowing the operating system running in that virtual machine to perform its own recovery processes. See also automatic software re-IPL.

**virtual = virtual machine (V = V machine).** A virtual machine that runs in the dynamic paging area. CP pages this virtual machine's guest real storage in and out of host real storage. See also dynamic paging area, virtual = fixed machine, virtual = real machine.

**virtual supervisor state.** A condition, controlled by a virtual machine's current PSW, during which the control program allows the virtual machine to issue input/output and other privileged instructions. When these instructions are not emulated, the control program intercepts these instructions and simulates their functions for the virtual machine.

**virtual wait time.** The period during which the control program suspends the processing of a program while a required resource is unavailable.

**VM/XA SP.** Virtual Machine/Extended Architecture System Product.

**VTAM service machine.** A collection of networking programs running in a virtual machine that, together with the CP console communication services (CCS) routines, provide full VM/XA SP console capabilities for SNA/CCS terminal users. A VTAM service machine contains either (1) VM/VTAM with VSCS running as an application under control of GCS, or (2) VM/VCNA running as a VTAM application under control of the VSE or VS1 operating system.

**V = R area.** Virtual = real area.

**V = R machine.** Virtual = real machine.

**V = R machine recovery.** Virtual = real machine recovery.

**V = V machine.** Virtual = virtual machine.

## Numerics

**370 mode.** Synonym for System/370 mode.

**370-XA mode.** A virtual machine operating mode in which System/370-Extended Architecture functions are simulated. Contrast with System/370 mode.

# Bibliography

This bibliography gives the names and order numbers of microfiche and publications about VM/XA System Product and related products.

## VM/XA System Product Microfiche

You can order microfiche listings that contain code. The order numbers for the microfiche are:

| Order No. | Description |
|---|---|
| LYC7-0330 | VM/XA System Product: CP listings |
| LYC7-0331 | VM/XA System Product: CMS listings |
| LYC7-0332 | VM/XA System Product: GCS listings |
| LYC7-0334 | VM/XA System Product: dump viewing facility listings. |

## VM/XA System Product Publications

The publications are shown in Figure 12 on page 454. You can order any of them by their individual order numbers or you can order most of them as a group by using a single order number, SBOF-0260. SBOF-0260 provides:

- All unlicensed publications (order numbers that do not begin with LY)
- Enough three-ring binders to hold the publications
- Spine and cover inserts for the binders.

## Non-VM/XA System Product Publications

For information about the System/370 and 370-XA architectures, see the following:

- *IBM System/370 Principles of Operation*, GA22-7000

For information about some of the separately orderable licensed programs that VM/XA SP supports, see the following:

- *VM/SP Remote Spooling Communications Subsystem (RSCS) Networking Version 2:*

  *Diagnosis Reference*, LY24-5228
  *General Information*, GH24-5055
  *Operation and Use*, SH24-5058
  *Planning and Installation*, SH24-5057
  *Program Reference and Operations Manual*, SH24-5005

- *Virtual Machine/System (VM/SP) Product Pass-Through Facility:*

  *Guide and Reference*, SC24-5208
  *Logic*, LY24-5208

- Networking products:

  *VTAM Diagnosis Guide*, SC23-0116
  *Network Program Products Planning (MVS, VSE, and VM)*, SC23-0110
  *VTAM Operation*, SC23-0113
  *VSE/VSAM Programmer's Reference*, SC24-5145

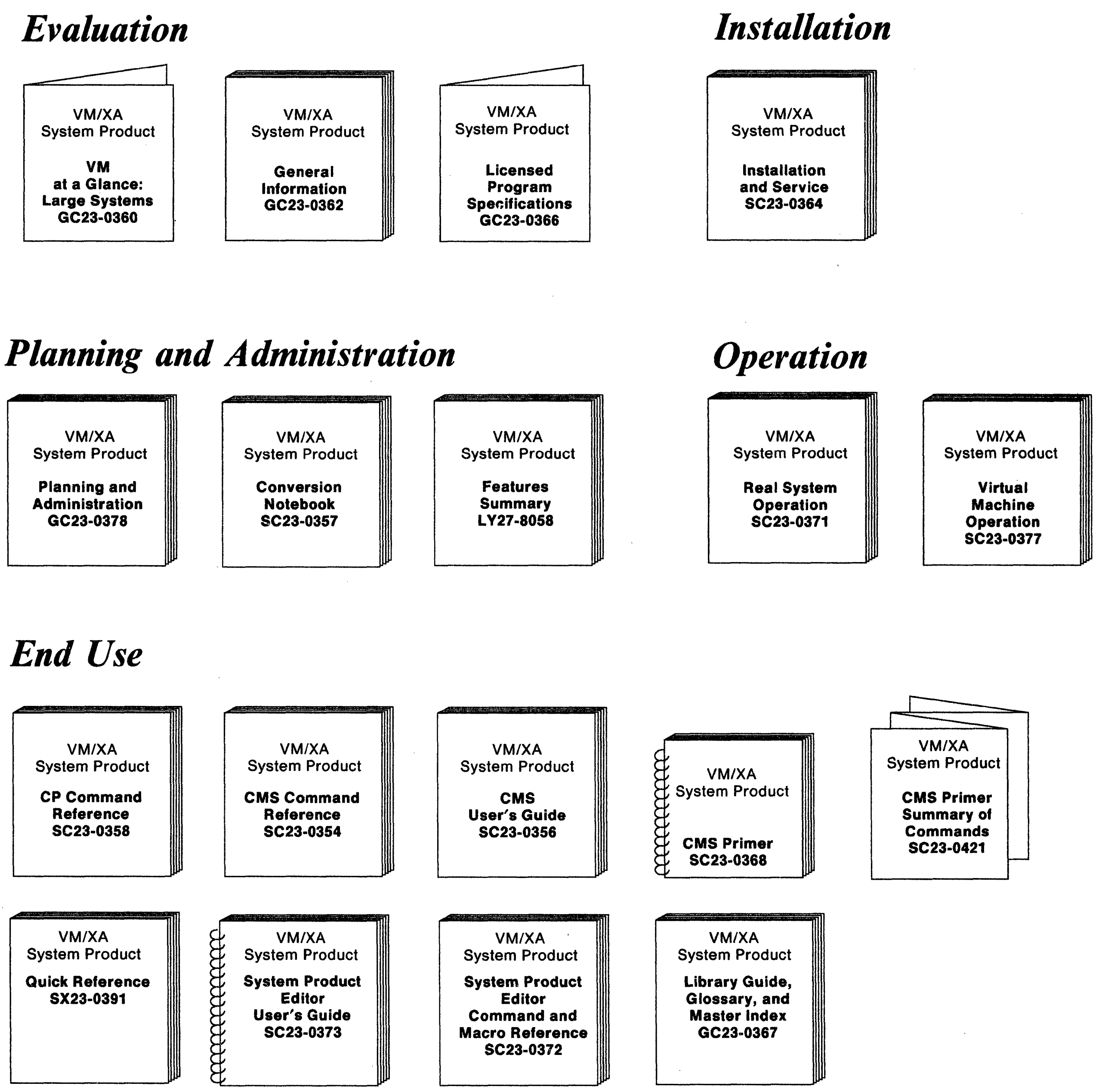
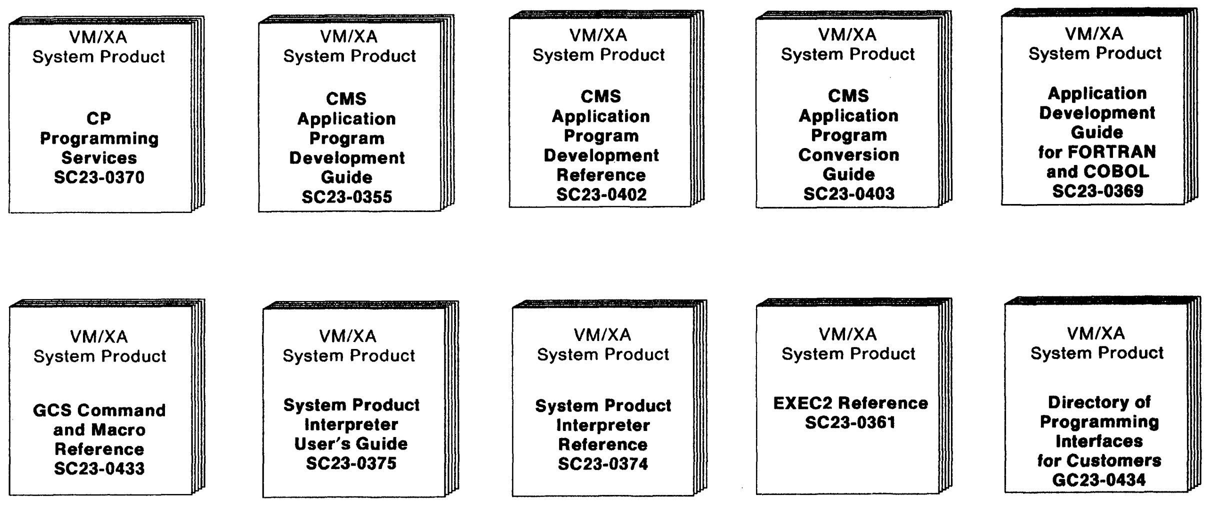
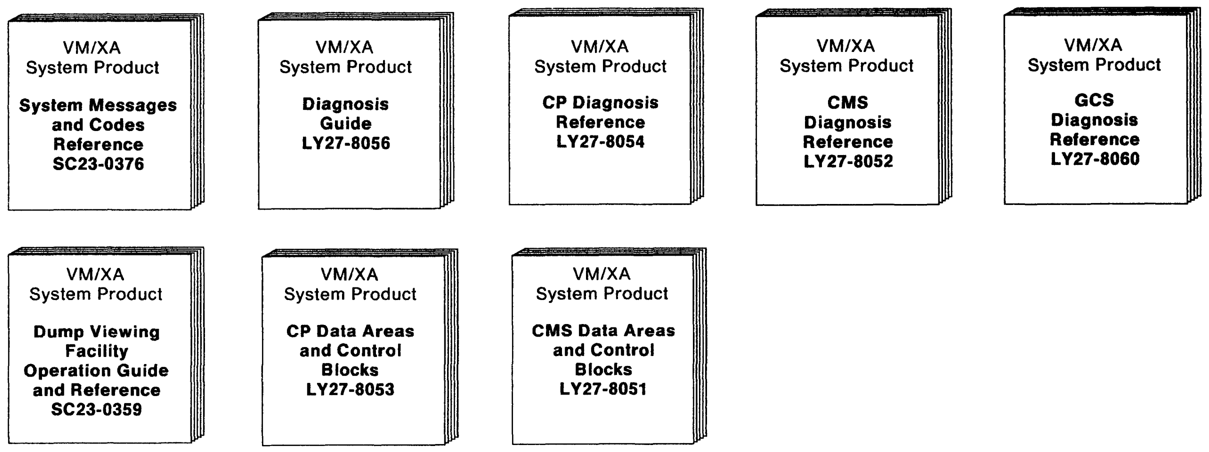

Evaluation
VM/XA System Product VM at a Glance: Large Systems GC23-0360
VM/XA System Product General Information GC23-0362
VM/XA System Product Licensed Program Specifications GC23-0366
Installation
VM/XA System Product Installation and Service SC23-0364
Planning and Administration
VM/XA System Product Planning and Administration GC23-0378
VM/XA System Product Conversion Notebook SC23-0357
VM/XA System Product Features Summary LY27-8058
Operation
VM/XA System Product Real System Operation SC23-0371
VM/XA System Product Virtual Machine Operation SC23-0377
End Use
VM/XA System Product CP Command Reference SC23-0358
VM/XA System Product CMS Command Reference SC23-0354
VM/XA System Product CMS User's Guide SC23-0356
VM/XA System Product CMS Primer SC23-0368
VM/XA System Product CMS Primer Summary of Commands SC23-0421
VM/XA System Product Quick Reference SX23-0391
VM/XA System Product System Product Editor User's Guide SC23-0373
VM/XA System Product System Product Editor Command and Macro Reference SC23-0372
VM/XA System Product Library Guide, Glossary, and Master Index GC23-0367

Figure 12 (Part 1 of 2). VM/XA System Product Publications

## *Application Programming*


VM/XA System Product CP Programming Services SC23-0370
VM/XA System Product CMS Application Program Development Guide SC23-0355
VM/XA System Product CMS Application Program Development Reference SC23-0402
VM/XA System Product CMS Application Program Conversion Guide SC23-0403
VM/XA System Product Application Development Guide for FORTRAN and COBOL SC23-0369
VM/XA System Product GCS Command and Macro Reference SC23-0433
VM/XA System Product System Product Interpreter User's Guide SC23-0375
VM/XA System Product System Product Interpreter Reference SC23-0374
VM/XA System Product EXEC2 Reference SC23-0361
VM/XA System Product Directory of Programming Interfaces for Customers GC23-0434

## *Diagnosis*


VM/XA System Product System Messages and Codes Reference SC23-0376
VM/XA System Product Diagnosis Guide LY27-8056
VM/XA System Product CP Diagnosis Reference LY27-8054
VM/XA System Product CMS Diagnosis Reference LY27-8052
VM/XA System Product GCS Diagnosis Reference LY27-8060
VM/XA System Product Dump Viewing Facility Operation Guide and Reference SC23-0359
VM/XA System Product CP Data Areas and Control Blocks LY27-8053
VM/XA System Product CMS Data Areas and Control Blocks LY27-8051

## *Binders and Inserts*

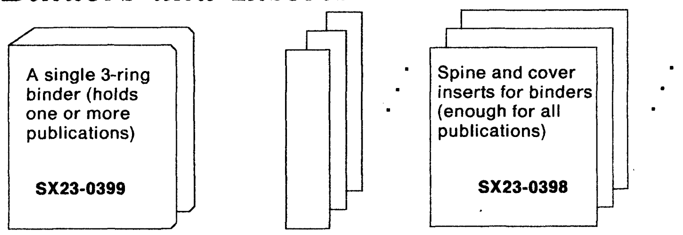

A single 3-ring binder (holds one or more publications) SX23-0399
Spine and cover inserts for binders (enough for all publications) SX23-0398

Figure 12 (Part 2 of 2). VM/XA System Product Publications

# Index

Note: Staples can cause problems with automatic mail-sorting equipment.
Please use pressure-sensitive or other gummed tape to seal this form.

**Virtual Machine/Extended Architecture ™**
**System Product and Macro**
**Group Control System**
**Command and Macro Reference**

**SYSTEM USABILITY COMMENTS**

**VM/XA™ SP Release 2**

Please use this form to communicate your comments about the usability of the VM system, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the Product Usability Department for appropriate review and action, if any. Comments may be written in your own language; English is not required.

**System Information**

If you answer **No**, please explain.

| | Yes | No |
|---|---|---|
| • Does the VM system meet your needs? | ☐ | ☐ |
| • Is it easy to use and understand? | ☐ | ☐ |
| • Are the commands/messages easy to understand and use? | ☐ | ☐ |
| • Are the HELP facilities appropriate? | ☐ | ☐ |

**Customer Information**

- What is your occupation? ______
- How long have you been in this occupation? ______
- How long have you been using VM? ______
- Indicate the tasks your job involves:

| | | | |
|---|---|---|---|
| Evaluation | ☐ | Planning | ☐ |
| Installation | ☐ | Administration | ☐ |
| Customization | ☐ | Operations | ☐ |
| Diagnosis | ☐ | End Use | ☐ |
| Other | ☐ | | |

**Your Comments:**

We appreciate your comments.

If you would like a reply, please supply your name and address on the reverse side of this form. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

**System Usability Comments**

Fold and Tape | Please Do Not Staple | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

**International Business Machines Corporation**
**Department 47U MS 914**
**Neighborhood Road**
**Kingston, New York 12401**

Fold and Tape | Please Do Not Staple | Fold and Tape

If you would like a reply, *please print:*

*Your Name* ____________________

*Company Name* ____________________ *Department* __________

*Street Address* ____________________

*City* ____________________

*State* ____________________ *Zip Code* __________

*IBM Branch Office serving you* ____________________

IBM®

PRINTED IN U.S.A.

Note: Staples can cause problems with automatic mail-sorting equipment.
Please use pressure-sensitive or other gummed tape to seal this form.

**Virtual Machine/Extended Architecture ™**
**System Product and Macro**
**Group Control System**
**Command and Macro Reference**

**READER'S COMMENT FORM**

**VM/XA™ SP Release 2**

**Order No. SC23-0433-0**

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

**Note:** *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

How did you use this publication?

| | |
|---|---|
| [ ] As an introduction | [ ] As a text (student) |
| [ ] As a reference manual | [ ] As a text (instructor) |

[ ] For another purpose (explain) ____________________

____________________

Is there anything you especially like or dislike about the organization, presentation, or writing in this manual? Helpful comments include general usefulness of the book; possible additions, deletions, and clarifications; specific errors and omissions.

Page Number: Comment:

What is your occupation? ____________________

Newsletter number of latest Technical Newsletter (if any) concerning this publication: __________

If you wish a reply, give your name and address: ____________________

____________________

____________________

IBM branch office serving you ____________________

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

SC23-0433-0

**Reader's Comment Form**

**Fold and Tape** **Please Do Not Staple** **Fold and Tape**

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

**International Business Machines Corporation**
**Department 52Q MS 458**
**Neighborhood Road**
**Kingston, New York 12401**

**Fold and Tape** **Please Do Not Staple** **Fold and Tape**

IBM®

PRINTED IN U.S.A.

Program Number
5664-308

File Number
S370-39

SC23-0433-0

Printed in U.S.A.