File No. S360-50
Form C28-2008-0

TSS

IBM Systems Reference Library

IBM System/360 Time Sharing System

System Programmer's Guide

IBM System/360 Time Sharing System (TSS/360) makes a distinction between user and system programmers. This publication is specifically intended for persons responsible for maintaining, modifying, or extending the system and discusses:

- Operating environment
- Program structure
- Coding practices and conventions
- Privileged supervisor call instructions
- Serviceability aids
- System macro definitions
- Changing TSS/360
- Privilege Class E

PREFACE

This publication will aid you -- as a system programmer -- extend and modify IBM System/360 Time Sharing System. We'll discuss the programming capabilities available to you and the conventions followed in developing the programs that are already part of TSS/360. We'll also cover a number of examples designed to give you a feeling for what is involved in changing the system.

There are four sections to this publication. The Introduction, Section 1, contains a reader's guide to help you find your way through the publications devoted to system programming. In Section 2, Resident Programs, we'll discuss some of the factors that go into writing resident TSS/360 programs. The same topics are discussed for Nonresident Programs in Section 3. Section 4, Defining Macro Instructions, discusses the types of macro instrunctions used in TSS/360 and the techniques you can use in writing them. Section 5, Generating and Maintaining TSS/360, shows some sample changes to the system. Finally, in Section 6, Programming with Privilege Class E, the additional facilities that are available to the System Monitor are discussed.

You may be reading this publication just for interest; you may have no intention to modify TSS/360. If this is the case, be sure and see the reader's guide -- it will help you in selecting those publications most beneficial to you. If you plan to change TSS/360, you should be an experienced programmer who knows the overall design of the system. You should have read -- or have handy -- most of the user-programmer publications. Finally, you should thoroughly understand IBM System/360 Principles of Operation, Form A22-6821, and IBM System/360 Model 67 Functional Characteristics, Form A27-2719.

First Edition (October 1967)

Significant changes or additions to the specifications contained in this publication will be reported in subsequent revisions or Technical Newsletters.

This publication was prepared for production using an IBM computer to update the text and to control the page and line format. Page impressions for photo-offset printing were obtained from an IBM 1403 Printer using a special print chain.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form is provided at the back of this publication for reader's comments. If the form has been removed, comments may be addressed to IBM Corporation, Time Sharing System/360 Programming Publications, Department 561, 2651 Strang Blvd., Yorktown Heights, N.Y. 10598

ILLUSTRATIONS

FIGURES

TABLES

READER'S GUIDE

Several publications concerned with System/360 Time Sharing System are devoted to system programming; they describe how TSS/360 was designed and constructed, how it can be modified, and how you can construct programs that will become part of it. The following descriptions will familiarize you with the purpose of the system programming publications that are available to you; they stress the role of each publication in presenting a complete picture of TSS/360, from the system programmer's viewpoint.

- System Programmer's Guide, Form C28-2008, describes the facilities available to system programmers in designing programs to be a part of TSS/360; also, it discusses the conventions used throughout the system.

- System Generation and Maintenance, Form C28-2010, describes the procedure for creating and maintaining the object and source forms of TSS/360; specifically, the macro instructions and commands you may use to add, delete, or modify system object program modules.

Program Logic Manuals

These TSS/360 publications describe the detailed design and implementation of specific groups of programs within the system:

- System Logic Summary PLM, Form Y28-2009
- Resident Supervisor PLM, Form Y28-2012
- Command Language Subsystem PLM, Form Y28-2013
- Program Checkout Subsystem PLM, Form Y28-2014
- Access Methods PLM, Form Y28-2016
- System Service Routines PLM, Form Y28-2018
- Assembler PLM, Form Y28-2021
- FORTRAN IV PLM, Form Y28-2019
- Linkage Editor PLM, Form Y28-2030
- Dynamic Loader PLM, Form Y28-2031
- System Control Blocks PLM, Form Y28-2011

When possible, you should use the program logic manuals in conjunction with a current assembly listing of the program modules of interest.

Assembler Language Manuals

The manuals described herein represent the TSS/360 documentation specifically intended for system programmers, who should, of course, be familiar with other IBM System/360 Time Sharing System publications, such as:

- Assembler Programmer's Guide, Form C28-2032
- Assembler User Macro Instructions, Form C28-2004

System programmer publications need not be read in sequence since, usually, each deals with a variety of TSS/360 considerations. As a recommendation, however, depending on your experience with the design and implementation of TSS/360, you probably will find it easier to read System Programmer's Guide and System Generation and Maintenance after you have read System Logic Summary. The program logic manuals should be read only after you have acquired a thorough familiarity with System Logic Summary. Finally, don't forget the program listings -- these should always be carefully checked before you attempt a local modification to TSS/360.

SYSTEM PROGRAMMING WITH TSS/360

Time Sharing System/360 is a set of programs. Each program is intended to perform a part of the overall job that the system as a whole was designed and developed to do. System programming with TSS/360 involves adding to, deleting from, or modifying these programs. By changing the function of the parts, the function of the whole is changed. This is the purpose of system programming.

Who

As a system programmer, you are expected to be an experienced programmer charged with the responsibility of modifying, extending, and generally tailoring TSS/360 to suit the needs of your installation. To do this you should be knowledgeable in two areas: the design and construction of TSS/360 and the needs and capacity of your installation. Within TSS/360 you will have greater authority than nonsystem programmers. The power to create, however, is also the power to destroy.

Why

Any large, general-purpose programming system is a compromise of the many conflicting demands of its prospective users. System designers attempt to take these diverse demands and create a cohesive, efficient programming system. All situations can never be anticipated. Generality must sometimes be sacrificed for efficiency. Realizing this, the developers of TSS/360 have produced a modular system; this facilitates change. The rules, suggestions, and operating considerations for making these changes are described in the following pages.

TSS/360 ORGANIZATION

The programs that make up TSS/360 are of two types. The first type, resident programs, are brought into main storage and left there, basically, until the machine is turned off. The second type, virtual memory programs, are brought into main storage as required and are removed from main storage when the space is needed. We call the first kind of program resident; the second kind, nonresident. During the operation of TSS/360, both kinds of programs "talk" to each other by using a well defined interface. We will discuss this interface later when we talk about virtual machines.

Resident programs generally have the responsibility of scheduling the use of the system's resources. For this reason they take care of most of the administration of the multiprogramming and multiprocessing going on in TSS/360. Nonresident programs have the responsibility for providing services to the user, making it easier for him to use the

system. An attempt has been made to separate these responsibilities as much as possible. In this way, resident programs needn't "worry" about providing user services, and nonresident programs needn't worry about scheduling a polymorphic computing system. If you keep this division of function in mind as you read this publication, you'll probably find it easier to understand the material. Sometimes you will think that we are discussing two different machines. In a way, we are.

This publication is organized along the lines of TSS/360, itself. A program that is part of TSS/360 resides in either main or virtual storage. In other words, it runs with the address translator turned off or on. This is the basic subdivision of the remainder of the material in this manual. No matter what area interests you, however, you should read this entire publication.

FORMAT AND NOTATION

The general format of the macro instructions and supervisor calls represented in this publication is:

Name	Operation	Operand

Name Field

The name field may contain a symbol or remain blank. Normally, this symbol is the name associated with the first executable instruction of the macro expansion.

Operation Field

The operation field contains the mnemonic operation code of the macro instruction or supervisor call. This code may be a string of not more than eight alphameric characters, the first of which is alphabetic.

Operand Field

The operand field may contain no operands (in which case the word "none" appears in the format illustration), or one or more operands separated by commas; the two types of operands are positional and keyword.

The user must supply positional operands in the same order as that shown in the format illustration. If a positional operand is omitted (the rules about omission are explained later in this section) and another positional operand is written to the right of the omitted operand, the comma that would have preceded the omitted operand must be retained. For example, assume positional operands A, B, and C. These may be written:

A,B,C	A,,C	A,B	A	,B	,,C

Keyword operands can appear in any order after the positional operands. Commas are not used to show omitted keyword operands. All keyword operands have the general form: KEYWORD=value-mnemonic.

The terms and formats used to illustrate operands are defined below:

OPERAND NAME: This is a single word, usually a mnemonic, that identifies the operand. Unless it is shown in upper-case letters in the command format, operand name represents a variable for which the user must supply specific information. A typical operand name is started; i.e., the address at which you are going to start.

VALUE MNEMONIC: This is a single word or mnemonic that tells how an operand should be written. The value mnemonics used in this publication are:

absexp
An absolute expression may be an absolute term or any arithmetic combination of absolute terms; an absolute term may be an absolute symbol or self-defining term. All arithmetic operations are permitted between absolute terms.

In the following examples, ALAN and JAY are relocatable and defined in the same control section; MARK and ERIC are absolute:

```
331
MARK
MARK+ERIC-2
ALAN-JAY
MARK*4-ERIC
```

addr
A relocatable expression or register notation. A relocatable expression is one whose value would change by n if the program in which it appears is relocated n bytes away from its originally assigned area of storage. A general-purpose register can be written as an absolute expression enclosed in parentheses. When evaluated, the absolute expression must have a value between 0 and 15, corresponding to the designations of the general-purpose registers. If, in the following examples, ALAN, GAIL, and JAY are relocatable and defined in the same control section, and MARK and ERIC are absolute, the following are relocatable expressions:

```
ALAN
ALAN+GAIL-JAY
GAIL-MARK*5
JAY+3
```

and the following are valid uses for register notation:

```
(5)
(ALAN-GAIL)
(ERIC)
(MARK+2)
```

addrx
Register notation, an explicit address, or an implied address. The valid forms of register notation are described above. An explicit address is written in the same form as an assembler language operand, that is , with a base, displacement, and index value. An explicit address might be written as:

```
2(0,5)
0(2,4)
5(3)
```

An implied address is written as a symbol, optionally indexed by a specified index register. For example:

LEE
CARL(2)

addx
An explicit or implied address, as described above.

characters
The character operand is written as a character string. Embedded commas or blanks are not permitted. Two apostrophes or two ampersands must be used to represent one apostrophe or one ampersand in the character string. The character string may not be enclosed in apostrophes. For example:

```
2+SADORE*LANE"6H"
```

A value written exactly as indicated in the description beneath the format illustration. Thus, LIEN would be written exactly that way within the program.

hexinteger
A hexidecimal value, which can be written as one or more hexadecimal characters from 0-F. The limit on the number of hexadecimal characters that are permitted is given under each format illustration in which this value mnemonic appears. The following are examples of hexintegers:

```
01
A2
ABC
```

integer
A decimal value, which can be written as one or more decimal digits from 0-9. The limit on the number of decimal digits that are permitted is given under each format illustration in which this value appears. The following are examples of integers:

```
006452
100
2134
```

relexp
A relocatable expression is one whose value would change by n if the program in which it appears is relocated n bytes away from its originally assigned area of storage. All relocatable expressions must have a positive value. A relocatable expression may be a relocatable term. A relocatable expression may contain relocatable terms -- alone or in combination with absolute terms -- under the following conditions:

1. There must be an odd number of relocatable terms.

2. All relocatable terms but one must be paired.

3. The unpaired term must not be directly preceded by a minus sign.

4. A relocatable term must not enter into a multiply or divide operation.

A relocatable expression reduces to a single relocatable value. This value is the value of the odd relocatable term, adjusted by the values represented by the absolute terms and/or paired relocatable terms associated with it. The relocatability attribute is that of the odd relocatable term. Complex relocatable expressions

are also permitted. Refer to IBM System/360 Time Sharing System: Assembler Language, Form C28-2000.

In the following examples of relocatable expressions, SAM, JOE, and FRANK are in the same control section and are relocatable; PT is absolute.

```
SAM
SAM-JOE+FRANK
JOE-PT*5
SAM+3
```

Note that SAM-JOE is not relocatable, because the difference between two relocatable addresses is constant.

specsym
A special symbol that may consist of any mixture of alphabetic, numeric, and/or special characters. For example,

```
1A370ABCD
OPENHOUSE+PARTY
```

symbol
A symbol may be a symbolic address (i.e., a single relocatable term), such as the name of an instruction in an assembler-language program, or it may merely be a character string used for identification, not location (such as the ddname parameter of a DCB macro instruction).

In TSS/360, the alphabetic characters are the letters A-Z, and $, @, and #. The alphameric characters are the alphabetic characters plus the digits 0-9.

The symbol is written as a string of up to eight alphameric characters, the first of which is alphabetic. Embedded commas and blanks are not permitted. Symbols beginning with the characters CHD may not be used, since symbols beginning with those characters are reserved for system use. Examples of symbols are:

```
DDNAME1
LOOP12
START
#1
```

text
A text operand is written as a string of alphameric characters enclosed in apostrophes. Embedded blanks and special characters are permitted. Two apostrophes or two ampersands must be used to represent one apostrophe or one ampersand in the character string. The text operand may not exceed 255 characters including the enclosing apostrophes. For example:

```
'AREA,PCB,132, ,1256'
```

value
A value may be written as an integer or as register notation. For example of each, see above.

CODED VALUE: This is a string of characters that is to be written exactly as shown. Coded values always appear in the command formats as numbers of upper case letters.

Positional operands are therefore represented with these elements in one of three ways, as shown below. The hyphen and the value mnemonic

are never written in the actual command. They serve only as a convenience in displaying the command format.

Operand	Example From Command Format
operand name-value mnemonic	action-code
operand name-coded value	field - {TOD / YMD / TASKINIT}
coded value	EDIT

What the programmer actually writes for each kind of positional operand is:

Positional Operand Representation	Programmer Writes
action-code	The appropriate action. For example; OFF
field - {TOD / YMD / TASKINIT}	The appropriate field. Either TOD, YMD, or TASKINIT
EDIT	EDIT

The keyword operand consists of a keyword followed by an equal sign and either a value mnemonic or a coded value. The programmer writes the keyword, the equal sign, and, when indicated, the coded value exactly as shown.

Notational Symbols

The symbols listed below are used in command formats to help the user decide how and when to write certain operands. None of these symbols is written by the user.

[ ] Brackets are used to denote options. Anything enclosed within brackets may be entered once or not at all. Stacked items show alternatives within the optional syntactical unit. For example:

[line-integer]

[E]

[VI / VS]

{ } Braces are used to denote grouping. Stacked items within the syntactical unit show alternatives. Examples:

{dsname-name / *ALL}

{7 / 7DC / 9}

... Three dots indicate the preceding syntactical unit may occur one or more times in succession. For example:

userid-alphname, ...

- Underlining of a stacked item means it is the default value of that syntactical unit. (The system will assume the underlined item is desired if nothing is entered for the unit.) For example:

This means the user may enter any one of the four items or, if RW (the underlined item) is his choice, he may default the entire syntactical unit.

In addition to the notational symbols described above, the comma and the parentheses have a special significance in the command formats.

Commas must always be entered to separate operands. They may also be used to show the omission of optional positional operands. Those operands may be omitted (i.e., defaulted) whenever their default values are desired. Mandatory postional operands must always be given. The rules for showing omission are as follows:

1. If another operand will be entered after the omitted operand(s), a comma must be given for each omission. (Technically, this is the comma that would have preceded the omitted operand.) The user will not be prompted for omitted operands in this case. For example, this command shows three omitted operands:

   PRINT MYDATA,,,,ERASE,SKIP,1234

   Note that a comma must follow the last omitted operand, to separate it from the following operand.

   But if initial operand(s) are omitted, only the separator comma(s) are necessary. For example, when a command contains optional operands A, B, and C, default of A is indicated as

   ,B,C

   and default of A and B would be indicated:

   ,,C

2. If no operand will be entered after the omitted operand(s), commas are optional. In nonconfirmation mode, the user will not be prompted for the omitted operands. In confirmation mode, he will be prompted unless he enters commas to show the omissions are intentional. For example, a user in nonconfirmation mode omits the last six operands of a command:

   PRINT MYDATA

   He is not prompted, and default values are assigned for the missing operands. If he made the same entry in confirmation mode, he would be prompted for each missing operand. To avoid this, he can write the command with commas to show that he wants the default values. Thus he enters

   PRINT MYDATA,,,,,,

   There is no prompting, and default values are assigned.

Note: Commas may not be used to indicate the omission of keyword operands.

The following table shows various ways of indicating default values of a command with optional operands A, B, C, D, and E. Assume confirmation mode.

Contents of Operand Field	System Response
A,B,C,D,E	No prompting; operand values are as given.
,B,C,D,E	No prompting. Default value of A assumed; other operand values are as given.
A,,,,E	No prompting. Default values assumed for B, C, and D; A and E values are as given.
,,,,	No prompting; default values assumed for all operands.
A,	Prompting messages issued for C, D, and E. Default value assumed for B; A value is as given.
A	Prompting messages issued for B, C, D, and E. A value is as given.
A,,,,	No prompting. Default values assumed for B, C, D, and E; A value is as given.
A,,,	Prompting messages issued for E. Default values assumed for B, C, and D; A value is as given.
no operand entered	Prompting messages issued for A, B, C, D, E

Parentheses must be written by the user exactly as shown. They are used to mark a group of similar items, such as a series of volume numbers. They must be given even if there is only one item in the group.

SECTION 2: RESIDENT PROGRAMS

This section discusses the characteristics of resident TSS/360 programs. These programs make up the resident supervisor. If you are primarily interested in the scheduling and resource allocation done by TSS/360, you will find this section of special interest. We will discuss the facilities available to you, and the conventions that should be followed in producing programs that are to be parts of TSS/360.

OPERATING ENVIRONMENT

GETTING STARTED

In TSS/360, a program called Startup has, as one of its duties, the job of bringing into main storage all the modules that make up the resident supervisor. Resident programs have the same physical appearance as any other TSS/360 object program module; they have a program module dictionary (PMD) and text. Startup acts as a limited-purpose link-loader. It reads the various resident program modules from a disk pack called the IPL volume, resolves the symbolic references between these modules, assigns them main storage space, and resolves address constants contained in them to appropriate values. Startup also initializes prefixed storage areas (PSAs) and issues an external start to a second processing unit, if one is attached.

Resident program modules are relocatable; however, once Startup has transferred control to the resident supervisor, the relocation of resident programs is complete.

A number of tables, or system control blocks, are also initialized by Startup. Basically, these tables tell the resident supervisor what resources it has to work with. One of these resources is main storage space, which will be reserved for the resident supervisor's use. Space not used for resident programs, or set aside for their use, will be available for allocation to nonresident programs.

NORMAL OPERATION

Extended Control PSW

When Startup transfers control to the resident supervisor, the IBM 2067 processing unit is in the extended control mode. The format of the extended control program status word (XPSW) is shown in Figure 1. Because resident programs operate unrelocated, bit 5 in the XPSW, the relocation bit, is always 0. The XPSW is also 0, allowing resident programs to access all available main storage. The problem state bit is 0, too, since resident programs operate in the supervisor state. Any program interrupt in the supervisor state is considered an error; to allow detection of program interrupts, the four program mask bits are 1s. The second word of the XPSW contains the instruction address. Resident programs are responsible for controlling dynamic relocation of programs; they do not, themselves, run with the address translator on. Addresses used by resident programs are always real addresses, limited by the storage physically available. The maximum allowable amount of main storage is 2,097,151 bytes ($2^{21}-1$) (16 million for 32 bit addressing).

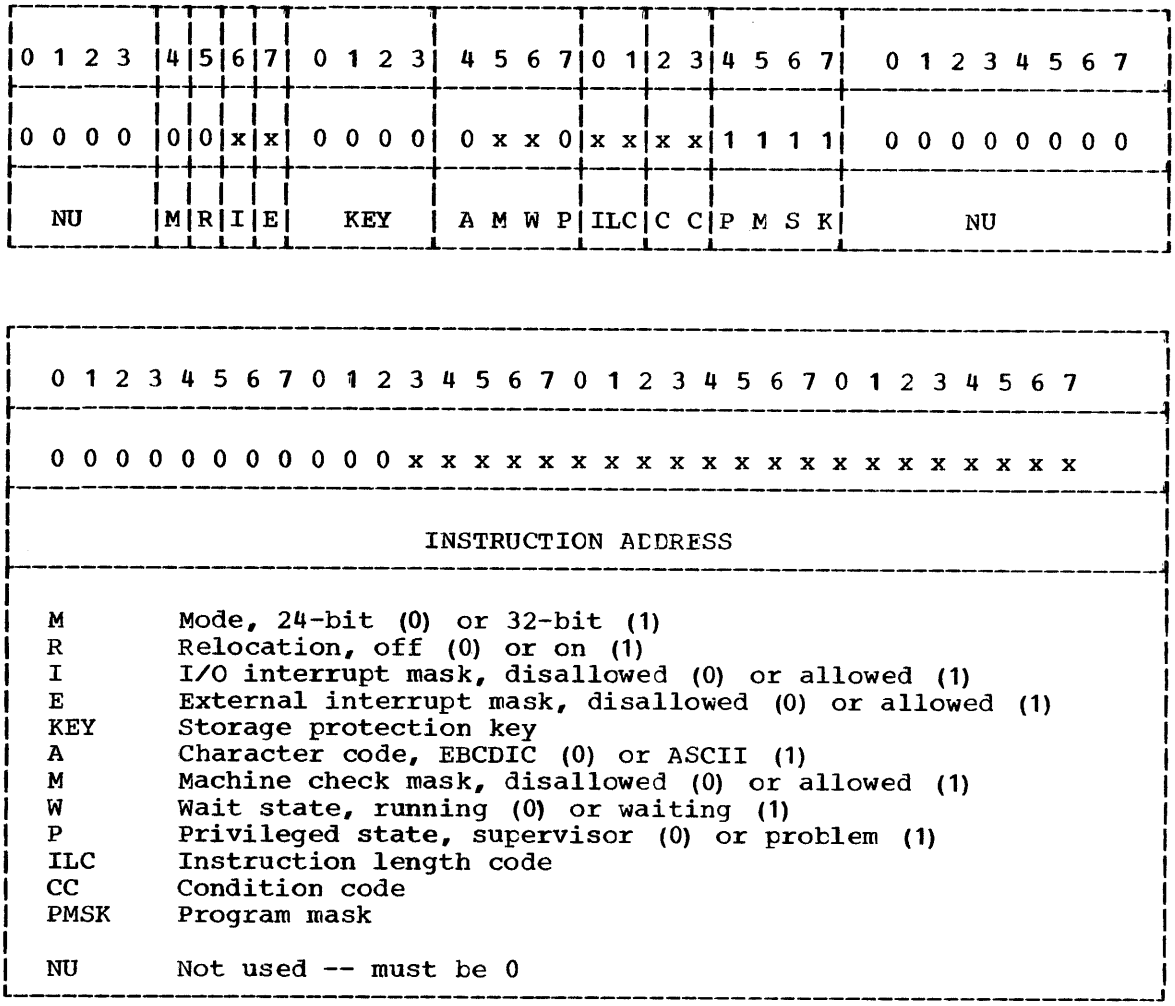


Figure 1. Extended Control Program Status Word

The Prefixed Storage Area

In a simplex or half-duplex System/360 Model 67 configuration, there is only one processing unit. In addition to its general-purpose and floating-point registers, this processor has 16 extended-control registers, eight associative registers, an interval timer, a dynamic address translation (DAT) unit, and other components necessary for fetching and executing instructions. (For a description of the nature and the function of the extended-control registers, refer to Functional Characteristics.) Associated with the CPU, and logically a part of it, is a page (4096 bytes) of main storage called the prefixed storage area (PSA); it is the main storage page referenced by real addresses 0 through 4095. The processing unit uses the prefixed storage area for fetching and storing new and old PSWs, for storing the interval timer value, and for initial program load (IPL).

The PSA is a special page; the high-order 12 bits of its addresses are always zero. Because of this, a PSA address can always be detected by the computing system; any real address whose 12 high-order bits are zero is a PSA address. Prefixing involves substituting an alternate set of bits for the 12 high-order zero bits in a PSA address before accessing main storage. It permits the prefixed storage area to be designated in any place in main storage. The ability to vary the location of the PSA gives us the flexibility of not having to rely on

the operation of a single main storage unit for the successful operation of the system.

There are two prefix quantities available for changing PSA addresses: the primary prefix and the alternate prefix. The selection of the prefix quantity enables us to select which page of which storage unit will be used to contain the PSA.

In a duplex System/360 Model 67 configuration, there are two processing units; each has its own prefixed storage area, just as each has its own set of general-purpose registers, floating-point registers, extended-control registers, associative registers, and interval timer. Each processing unit also has its own primary and alternate prefix quantity.

It does not matter how a PSA address is produced; it may be the result of normal program execution.

A PSA is addressable by two sets of (real) addresses. Clearly, the PSA can be addressed by using 0 through 4095; this is why it is a PSA. If we know what prefix quantity a processing unit is using to address its PSA, we can use that prefix quantity ourselves to access the PSA -- without prefixing. This is like using "before" and "after" addresses. Each PSA has a set of "before" addresses, 0 through 4095, and a set of "after" addresses, the addresses produced by substituting a processing unit's prefix quantity for the 12 high-order bits of the "before" addresses. A processing unit can use either set of addresses to access its PSA; the first set involves prefixing, the second set does not.

The advantage of the second set of addresses, the "after" addresses, is not that a processing unit can address its own PSA. If processor A knows processor B's prefix quantity, then by using that quantity, processor A can access processor B's PSA. Processor A cannot access processor B's PSA in any other way, since the other set of addresses (0-4095) causes processor A to access its own PSA because these addresses are automatically prefixed. By knowing each other's prefix quantity, processors in a multiple processor system can fetch and store from one another's PSAs. This is an important capability for error recovery and reconfiguration. In a dual system, page zero in main storage is not used since its actual and prefix addresses would be the same; this would prohibit one of the processing units from referencing page zero by using its actual (not prefix) addresses.

Even though we keep each processing unit's PSA separate from the other's, the PSAs are organized in the same way. This is somewhat analogous to a processor's private registers. The PSAs, like the registers, are physically separate but logically identical. A PSA may be considered as a "socket" when a processing unit is "plugged" into the main storage of the system.

The format of the prefixed storage area can be seen by copying the DSECT. Bytes 0 through 327 are fixed by the design of the IBM 2067 processing unit. Bytes 328 through 4095 are not required for an explicit 2067 processing unit function, such as interruption handling, but are used as a special storage area by TSS/360. The PSA is used to store data that relates to the processing unit in which the PSA belongs; data that applies to the system as a whole is kept in common storage.

SUMMARY

Resident programs make up the part of TSS/360 known as the resident supervisor. The extended control mode of operation is normal for resident programs. These programs operating in the supervisor state with the address translator turned off, with an XPSW protection key of

zero, may execute any Model 67 instruction, and access all main storage except page zero in a dual system. Although each processing unit in a multiprocessor configuration shares common main storage, each also has a single private page of main storage, the PSA. Each processing unit also has 16 general-purpose registers, four floating-point registers, 16 extended-control registers, eight associative registers, an interval timer, an address translator, and other components for fetching and executing instructions.

DUMMY SECTIONS

PURPOSE

The dummy section (DSECT) is used extensively throughout TSS/360 so that parts of the system can refer to commonly used data items by symbolic names. You can refer to a field in a system control block by the name assigned to that field in the dummy section; this frees you from having to use the field's numeric location. (Actually, the dummy section supplies a number of symbolic field names, lengths, and relative positions which the assembler translates into numeric displacements.) You needn't worry about the specific physical structure of the system control block to which you are referring if you use a DSECT to describe the control block. All you need be concerned with is the field structure (bit, byte, halfword, etc.). Essentially, you don't care where the field is located within the system control block. Thus, if the field position changes, but the field length, boundary alignment, and the meaning of its contents don't change, your program will still run properly after it is reassembled. Reassembly is necessary since displacement values may have changed as a result of using the new dummy section.

In TSS/360, the dummy section is more than a programmer convenience. Dummy sections for system control blocks, obtained from the assembler copy/macro library, ensure that all programs using the same system control block use the identical format. The set of TSS/360 dummy sections can be viewed as a central, current, assembler-oriented description of all system control blocks.

USE

A typical TSS/360 dummy section is illustrated in Figure 3. We use several conventions when working with dummy sections. These conventions are intended to minimize the need for redesigning programs if the dummy sections they use are changed. Dummy section fields that are integral multiples of bytes in length are simply referred to in a program by using the name of the field. Fields that are less than one byte long are referred to by using a mask; we must do this since the IBM 2067 processing unit can directly address no field shorter than a byte. The name of the mask associated with a field that is less than one byte long is obtained by adding the character M to the field name. If we wanted to determine whether the field VPSAI were a 1 we might write,

```
TM    VPSAI,VPSAIM     TEST UNDER MASK
BZ    FIELDOFF         BRANCH IF ZERO
```

Note that we don't have to know where the field VPSAI is located within the system control block or what bit pattern defines the mask VPSAIM. This information is supplied by the dummy section which we incorporate into our program from the assembler copy/macro library. The field we are testing, with the test under mask (TM) instruction, need not be restricted to a single bit. It can be any combination of up to eight bits, as long as all the bits fall within one byte. The

conditional branch instruction can be used to determine if the bits we are testing are all 1s, all 0s, or mixed.

For bit fields, the field name is always the name of the byte in which the bits appear. Since a single byte can have up to 256 different conbinations of bits, a single byte could have up to 256 different bit fields. We frequently find, therefore, that the names of different bit fields are synonomous; that is, they point to the same byte. The bit mask corresponding to the field name must be used to extract the proper bits.

The dummy section itself does not take up program storage space; it is used exclusively to describe a storage area to which it is applied. To properly use a dummy section, we first load a general register with an address constant, pointing to a storage area containing information described by the dummy section. Then we issue a USING pseudo-operation to tell the assembler that the corresponding dummy section format is to be applied to the storage area pointed to by the general register given in the USING statement. It locks like this:

```
L       5,ADCON
USING   CHAVPS,5
```

assuming that ADCON has been defined as,

```
ADCON DC V(WORKAREA)
```

This would apply the format given by CHAVPS to the storage area beginning at the symbolic location work area.

We can, of course, define our own dummy sections and use them as we see fit. In most cases, though, we will get the dummy section from the assembler copy/macro library (see Section 4, "Generating and Maintaining TSS/360"). We can do this simply by issuing a COPY pseudo-operation, with the name of the dummy section we want, as the operand. Here is an example:

```
COPY    CHAVPS
```

The dummy section currently contained on the assembler copy macro-library will be included in our program at the point of the COPY statement. This will enable us to symbolically reference the system control block CHAVPS, as it is currently defined (see IBM System/360 Time Sharing System: System Control Blocks, Form Y28-2011).

MODULE STRUCTURE

Like any other object module, resident object program modules consist of a program module dictionary (PMD) and hexadecimal text. Usually, resident programs contain a single read-only, nonprototype control section. A resident program may contain address constants to be computed and placed into the text by startup. The read-only control sections do not change; they are never modified during program execution.

In addition to read-only control sections, the resident supervisor contains tables or system control blocks. A system control block is nothing more than some data contained in main storage and organized in some way known to the programs that use it. The system table, CHBSYS, is an example of a system control block used by a number of resident programs; it contains such information as the size of the time slice, the operational cycle time, and other parameters affecting the overall operation of the system.

If there is any possibility that one processor can change a system control block at the same time another processor is working on it, the control block must be protected, or interlocked, with a lock byte. A lock byte is a single byte used to control the accessing of variable information. The test and set instruction is used to find out if a lock byte is on or off (and also to turn it on). For example:

```
TEST    TS    LOCK
        BC    1,WAIT
```

tests a lock byte called LOCK; if LOCK is all 1s, control is transferred to WAIT. A lock byte is set (or on) if it is all 1s (actually, only the high-order bit is tested); it is reset (or off) if it is all 0s.

Some programs do not need to test lock bytes, since they are subroutines of programs that do test. Some control blocks are not individually tested (and do not contain a lock byte) but are gathered into queues; the entire queue is interlocked instead of its members.

A lock byte is reset when the program that set it has finished using the protected information. In most cases a second processing unit will only wait a certain length of time for a lock byte to be reset: if the lock byte is not reset within that time period, a minor system error is recognized. When control is returned to the point of interruption by the ERROR routine, accessing the protected data proceeds.

In addition to read-only control sections and interlocked system control blocks, the resident supervisor also contains a number of pages or a "pool" of main storage that may be used, as required, by resident programs. The program controlling the use of this storage pool is the supervisor core allocation module. Since the supervisor core allocation module itself cannot require the allocation of storage space to free working registers it saves the registers in a special area in the PSA.

Relatively few system control blocks continuously require main storage space; most have transient storage needs. Resident programs usually obtain storage space for transient data from the supervisor core allocation module. When the need for this data no longer exists, the space is returned to the supervisor core release subroutine; this dynamic allocation of main storage space ensures that the resident supervisor doesn't tie up more storage space than it actually needs. Most transient data areas cannot be accessed simultaneously by multiple processing units; these control blocks are not interlocked. However, a few transient data areas can be accessed by multiple processing units and are, therefore, interlocked.

Some data areas are known only to one processing unit because one of its registers points to the data area; other data areas are known to all processing units because the address of the data area is kept in common main storage.

SYSTEM CONTROL BLOCKS

The resident supervisor consists of three parts: read-only control sections (resident), nontransient system control blocks, and a pool of dynamically allocatable main storage. The resident supervisor uses the storage pool to create transient system control blocks such as the generalized queue entry (GQE) and the page control block (PCB). During their "lifetimes" transient control blocks are resident in main storage. Transient control blocks exist only as long as they are needed; this may be a few milliseconds or a few minutes. When they are no longer needed, the main storage space they occupy is returned for reallocation.

The resident supervisor also creates nonresident system control blocks. Nonresident system control blocks are brought into main storage only when needed. They exist on some storage device for a relatively long time, for example, for an entire terminal session. However, their time in main storage may represent only a small fraction of their lifetime in the system.

MODULE DESIGN CONSIDERATIONS

Let's assume you have a change to TSS/360 in mind, and that you very clearly understand the logic of the change you wish to make. The next question is: How do you construct the program? Resident programs are different from nonresident programs in one major way: Resident programs do not contain prototype control sections (PSECT). As you know, the purpose of a prototype control section is to contain any part of a program that changes as a result of relocation or execution. As pointed out previously, resident programs never change during execution. The address constants used by resident programs are resolved by startup; they will not be changed after the system is initialized. The only thing left in the resident supervisor that can change are the variables used by resident programs. These variables are kept in general registers, system control blocks, or working storage obtained from the supervisor core allocation subroutine. Your program must be designed to use one of these areas for holding variable information; which one you use depends on what you are attempting to do. The key test of a resident program's correct construction is that it be simultaneously executable by multiple processing units. If the general registers are used as working storage, multiple processors may simultaneously execute the program, since each processor supplies its own registers. If a system control block is used, the lock byte controls the modification of variable information. If storage obtained from the supervisor core allocation subroutine is used, each allocation of storage is kept separate from the others to ensure the protection of variables.

ENABLING AND DISABLING INTERRUPTS

Because they operate in the supervisor state, resident programs can enable and disable interrupts by setting and resetting the system mask or by altering the contents of extended-control registers 4, 5, and 6. The instruction

```
SSM  =X'00'
```

will set bits 0 through 7 of the extended program status word to zero. The processing unit affected will interpret these bits as: 24-bit relocation mode, address translator off, and I/C and external interrupts disabled. To restore interrupts,

```
SSM  =X'0B'
```

will be interpreted by the processor as: 32-bit relocation mode, address translator off, and I/O and external interrupts enabled. When the address translator is turned off, the setting of the relocation mode bit is academic; it is set to 32-bit mode here only for the sake of illustration. If you wish to modify the extended-control registers, the instructions

```
STMC 4,4,SAVE
L    6,SAVE
N    6,=X'BFFFFFFF'
ST   6,SCRATCH
LMC  4,4,SCRATCH
```

will save the contents of control register 4, and disable interrupts from channel 1 (as viewed by the processing unit issuing the LMC). The instruction

```
LMC  4,4,SAVE
```

will restore the original contents of control register 4. The work areas for SAVE and SCRATCH would be obtained from the supervisor core allocation subroutine.

NAMING CONVENTIONS

PROGRAM MODULE NAMES

All TSS/360 programs have standardized program module names, control section names, and entry point names. When a program module becomes part of the system, any reference to it must use the module name, an entry point name, or a control section name. TSS/360 program module names consist of five characters; resident program module names have the form:

```
CEAXX
```

where XX are alphameric characters that uniquely identify the module within the resident supervisor. All TSS/360 program module names begin with C; the characters EA identify resident supervisor modules.

All entry point and control section names begin with the program module name, like this:

```
CEAXXN
```

where N is a character that uniquely identifies the entry point or control section within the program module. Note that special characters are not used in TSS/360 names.

As an example, the pathfinder module has the name, CEAA5; its entry points are CEAA5P, CEAA5R, and CEAA5S.

SYSTEM CONTROL BLOCK NAMES

In addition to program modules, the resident supervisor is made up of system control blocks. A system control block usually requires two names: the name of the dummy section (DSECT) that describes its format, and the symbolic address that points to the information described by the dummy section. All TSS/360 programs, resident and nonresident, use the same rules to name system control blocks. A dummy section name looks like this,

```
CHAXXX
```

The characters XXX are assigned to uniquely identify the dummy section. All fields used within the dummy section look like this:

```
XXXFFF
```

The characters XXX are the same as the last three characters of the dummy section name. The characters FFF are any three characters that uniquely identify the field within the dummy section. For example, these are the assembler statements for a typical dummy section:

```
CHAABC    DSECT            CONTROL BLOCK NAME
ABCFCA    DS     1F        FIELD NAME
ABCRJG    DS     4F        FIELD NAME
ABCXYZ    DS     3C        FIELD NAME
ABCFLG    EQU    ABCXYZ    FLAG NAME
ABCFLGM   EQU    X'80'     FLAG MASK
```

Note that the field name, ABCFLG, is the name of a byte containing a flag bit. The field ABCFLGM can be used as a mask byte in a test under mask instruction (TM) to test the condition of the flag. Mask names are of the form,

```
XXXFFFM
```

where XXXFFF is the name of the dummy section field to which the mask is applied. For more information on dummy section usage, see "Dummy Sections" above.

The dummy section is, of course, only a description of information; it does not supply anything more than the format of the information it describes. Symbolic addresses that point to areas of storage described by dummy sections are named like this:

```
CHBXXX
```

where the characters XXX are the same as the last three characters of the dummy section name. For example,

```
DATA      DC     V(CHBABC)
```

is an address constant pointing to an area of storage organized as described by the dummy section CHAABC.

Remember, CHAXXX says what the information looks like; CHBXXX says where the information is located. The DSECT can only be used for nontransient system control blocks. Symbols generated by macro instructions always begin with CHD. For example,

```
CHD103    MNOTE 3,'ERROR'
```

might be found in a macro definition.

SECONDARY ENTRY POINTS

Resident modules with more than one entry point are designed sc that their base register always points to the primary entry point, even if control of the modules has been transferred to secondary point coding to set up the module's base register to point to the primary entry point. Sometimes its done like this:

```
BASE      EQU      15
ENTRY1    USING    *,BASE
          .
          .
          .
ENTRY2    BASR     BASE,0
          USING    *,BASE
          L        BASE,ADCON
          USING    ENTRY1,BASE
          .
          .
          .
ADCON     DC       A(ENTRY1)
```

where ENTRY2, the secondary entry point, establishes its own addressability and sets up BASE with the address of ENTRY1 (ADCON must be within 4096 bytes of BASE). If we don't do this, and try to use an address "above" the secondary entry point, that address can't be reached; this is why ADCON is "below" ENTRY2. Although we could have assumed that register BASE contained the address of the secondary entry point and eliminated the BASR instruction at the secondary entry point this wouldn't allow us to proceed sequentially from the instruction immediately preceeding the BASR instruction. The secondary entry point, ENTRY2, coded this way, will operate properly if control is transferred to it, or if control passes to it sequentially; that is, without a branch.

SUPERVISOR LINKAGE CONVENTIONS

Resident programs link to subroutines by using a V-type address constant (V-con) or an A-type address constant and an EXTRN statement. Startup will resolve all symbolic references among resident programs, and supply the correct values for the address constants. Resident programs never use R-type address constants (they do not contain prototype control sections). One resident program transfers control to another by a branch-and-store (BASR) instruction. Any understanding between the calling and the called programs concerning the contents of the general registers is arbitrary and depends on the particular programs involved. The calling program must know what register the called program is using as a base register. To transfer control, the calling program loads the address of the called program's entry point into the called program's base register, like this,

```
L       BASE,=V(ENTRYPOINT)
```

Then the calling program branches to that entry point,

```
BASR    14,BASE
```

At the other end of the line, the called program expects its base register to contain the address of its entry points. The called program usually begins like this:

```
USING   *,BASE
```

to tell the assembler that register BASE will contain the address of the called program's entry point when control is transferred.

GETTING RESIDENT WORKING SPACE

We do not assemble working space into resident programs for two reasons: it is inefficient to assemble space that may not be used into a program; it would require the resident program to schedule the use of that space if the program were to be simultaneously executed by multiple processing units. Resident programs use four modules to obtain working space for their execution. These routines are supervisor core allocation (CEAL01), supervisor core release (CEAL02), user allocation (CEANB), and user core release (CEAL04). User core is allocated from one pool and supervisor core from another but the supervisor pool may be replenished from the core released by user core release. The supervisor core allocation routine satisfies requests for resident working space such as control blocks like the generalized queue entry (GQE) and the task status index (TSI). The user core allocation routine satisfies requests for storage for extended task status indexes (XTSI) and for nonresident program pages.

The supervisor core allocation subroutine is a special program since it has private space in the prefixed storage area (PSA), which it uses to store the contents of the general registers. It must use this area since there is no subroutine that it can call to get working space. Programs that call the supervisor core allocation subroutine must save four registers (0, 1, 14, and 15) before transferring control. They can't do this without working space, though, so four words in the prefixed area (PSASCU) are set aside for programs calling the supervisor core allocation subroutine. This lets a called program immediately become a calling program without losing the contents of any of the general registers that were supplied to it. Since a program calling supervisor core allocation must use registers 0, 1, 14, and 15 to transfer control (and parameters), it would lose the original contents of these registers if it had no place to save them. A typical use of the supervisor core allocation subroutine might look like this:

```
SUBR    USING   *,15            REGISTER 15 CONTAINS BASE
        COPY    CHAPSA          GET THE DSECT
        USING   CHAPSA,0        PSA DSECT NEEDS NO BASE
        CSECT                   REESTABLISH CSECT
        STM     14,1,PSASCU     SAVE REGS 0, 1, 14, and 15 IN PSA
        LA      0,128           REQUEST 128 BYTES
        SR      1,1             OPTIONS ALL ZERO
        L       15,ADCON        POINTER TO SUPVR CORE ALLOC
        BASR    14,15           TRANSFER CONTROL
RTRN    LM      14,1,PSASCU     RESTORE REGS 14, 15, 0, AND 1
        .
        .
        .
ADCON   DC      V(CEAL01)       ADDR SUPVR CORE ALLOC
```

In this example, when supervisor core allocation returns control, register 1 points to a 128-byte area of main storage that can be used for any further transient storage needs this program may have. The supervisor core allocation subroutine will not disturb the register contents saved by this program in PSASCU since supervisor core allocation has its own private save area in the PSA (PSACAS).

In order to give this space back to the supervisor core release module, the program might be coded like this:

```
DONE    STM     14,1,PSASCU     SAVE REGS
        L       1,SCAREA        ADDRESS OF SPACE WE'RE RETURNING
        LA      0,128           GIVE BACK 128 BYTES
        L       15,ADCN2        POINTER TO SUPVR CORE RETRN
        BASR    14,15           TRANSFER CONTROL
RTRN    LM      14,1,PSASCU     RESTORE REGS
        BR      14              RETURN TO ORIGINAL CALLER
ADCN2   DC      V(CEAL02)       ADDR SUPVR CORE RELEASE
```

This will return for reallocation the 128 bytes obtained in the previous example.

PROGRAMMING CONVENTION COMMENTS

There is no requirement for resident programs to use particular registers as base registers, return registers, or parameter registers; however, almost all resident programs use these registers,

```
register 0 -- parameter register
register 1 -- parameter register
register 14 -- return address of calling program
register 15 -- entry point of program being called
```

Because of these register assignments, most programs being called begin with,

```
        USING  *,15
```

and end with,

```
        BR     14
```

or the equivalent.

A number of resident programs, such as SVC processing routines, return control to a location pointed to by a V-type address constant instead of branching to the address contained in register 14. For example,

```
THRU    L      14,ADCN3      GET RETURN ADDRESS
        BR     14            TRANSFER CONTROL
ADCN3   DC     V(CEAHND)     ADDR SVC Q PROC RETURN
```

is the way that a SVC processing routine can transfer control back to the supervisor call queue processor. This is done because most SVC processors require two common functions to be performed and this portion of the SVC queue processor provides them with the functions.

SECTION 3: NONRESIDENT PROGRAMS

Nonresident, or virtual storage, programs are programs that operate with the address translation unit turned on; they do not permanently reside in main storage. There are two kinds of virtual storage programs: privileged and nonprivileged. For a conceptual understanding of virtual storage, you should read System/360 Concepts and Facilities.

VIRTUAL MACHINE STRUCTURE

A virtual machine is a one level store achieved through the use of the address translator. A virtual machine running in the problem state is analogous to, though not identical with, one computer being emulated by another. A virtual machine has storage and an instruction set. Just as System/360 subdivides its instruction set into two states, supervisor and problem, a virtual machine has a privileged and a nonprivileged supervisor call set.

The resident supervisor is responsible for scheduling and allocating the computing system's resources to satisfy the collective demands placed upon these resources by virtual machines. The resident supervisor does this by parceling processing unit time, main storage space, and data channel time to the virtual machines it is supporting.

A virtual machine has a large virtual storage whose size is essentially independent of the physical main storage available to the resident supervisor. Virtual storage is thought of as being organized into pages of 4096 bytes which are further collected into segments of 256 pages. Depending on the type of address translator installed on the IBM 2067 processing unit, virtual storage can consist of a maximum of 16 segments (16,777,216 bytes) or 4096 segments (4,294,967,296 bytes).

A virtual machine appears to be like any other machine to a user. Its implementation, however, is different; it is implemented with programming as well as hard-wired components. By "juggling" the system's resources, the resident supervisor can support a number of virtual machines at the same time. Each of these machines appears to be independent of the others. As we discuss virtual machines, we generally will talk about a single one for simplicity of discussion; what we say about a single virtual machine applies to all. The fact that there may be many virtual machines supported at the same time by the resident supervisor - although it may cause problems of contention for resources - does not affect the logical appearance of any of the virtual machines.

VIRTUAL PROGRAM STATUS WORD

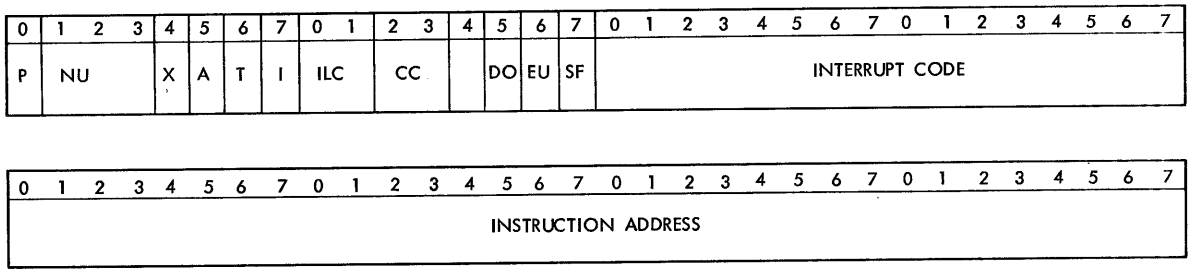
Each virtual machine has 16 general-purpose registers and four floating-point registers. The status of a virtual machine is described by its virtual program status word (VPSW), which is shown in Figure 2.

INTERRUPT STORAGE AREA

Virtual machines can be interrupted with a virtual, or task, interrupt. When this occurs, the information comprising the current VPSW is stored in a predetermined area of virtual storage; a new VPSW, obtained from another location in that area, becomes the current VPSW (it is not apparent to the task that this is done by supervisor programming). The area, called the interrupt storage area (ISA) is

analogous to the prefixed storage area (PSA) in the system's real main storage. The interrupt storage area is bytes 0 through 4095 of virtual storage. There are six different virtual, or task, interrupts: program, supervisor call, external, asynchronous I/O, task-timer, and synchronous I/O. The occurrence of four of these interrupts is controlled by the task mask in the VPSW, analogous to the system mask in the PSW. If the mask bit corresponding to a given interrupt type is 0, or if the interrupt storage area is locked, interrupts for that type will be "stacked," i.e., saved by the resident supervisor, until the mask bit is set to 1.

Each virtual machine has an instruction set consisting of all System/360 problem state instructions and a number of supervisor call instructions. The supervisor call instructions are further divided into SVCs that can be issued only by privileged programs, SVCs that can be issued only by nonprivileged programs, and SVCs that can be issued by privileged or nonprivileged programs depending on the authority code of the programmer. The privileged SVCs are analogous to System/360 supervisor state instructions. Each of these SVCs will be described in detail later.



Legend

P=Privilege. (0) privileged or (1) nonprivileged

NU=Not used

The next four bits are the task mask and are interpreted:

X=External interrupts

A=Asynchronous interrupts

T=Timer interrupts

I=Synchronous interrupts

In all cases, zero indicates interrupts disallowed and one indicates interrupts allowed

ILC=Instruction length code

CC=Condition code

FO=Fixed point overflow mask

DO=Decimal overflow mask

EU=Exponential overflow mask

SF=Loss of significance mask

In each of the above four bits, one permits an interrupt on the occurrance of the condition and zero inhibits the interrupt.

Figure 2. Virtual Program Status Word

The current VPSW includes the address of the instruction following the last instruction executed prior to the interrupt. While the interrupt is being serviced or is waiting to be serviced, this address is significant in that it points to the instruction at which execution is to be resumed. Once execution is resumed, the current VPSW continues

to point to the same instruction; it is not incremented as each instruction is executed and, therefore, loses its significance.

LINKAGE CONVENTIONS

The purpose of a linkage convention is to standardize the method of transferring information and control from one program (the calling program) to another (the called program). Standardization eliminates redundant register usage and allows the use of system macro instructions for the generation of program linkages. Any linkage convention is a compromise between generality and efficiency - the most general linkage convention is not the most efficient, and vice versa. Therefore, TSS/360 uses a number of linkage conventions designed to fit a variety of situations while attempting to keep the conventions as similar as possible.

TSS/360 linkage conventions require the calling program to supply a save area for use by the called program. A save area is an area of virtual storage, accessible to the called program, in which it can save the contents of general-purpose registers, if necessary. Also, a save area contains forward and backward address pointers to other save areas, forming a chain. If one program calls another, and the second program calls a third, the address pointers relate the respective save areas. Thus, if you know one save area is located, you can find the others. The format used in TSS/360 is shown in Figure 3.

The four basic linkage conventions followed by TSS/360 programs residing in virtual storage are summarized in Figure 4. Note that type I has two variations; this yields five distinct linkage conventions; these are the only linkage conventions in use among virtual storage program in TSS/360. All TSS/360 programs are constructed to receive or transfer control, using one of these linkage types. If you wish to add or modify TSS/360 programs you must use these linkage conventions.

In general, TSS/360 system programs use macro instructions for generating program linkages. You should take care, therefore, not to confuse a linkage-producing macro instruction with the program linkage itself. It is preferable to use a macro instruction to generate a program linkage. Some macro instructions generate more than one type of program linkage; for example, GETMAIN can generate either a type-I or a type-II linkage.

The called program frequently does not know which linkage type was used to transfer control to it; generally, it does not need to know. There are exceptions; we shall cover them later. The linkage type must be known by the calling programs, since it is the calling program that supplies the linkage instructions, save area, and proper register contents.

TYPE-I LINKAGE

Type-I linkage is used for transfer of control and information between two programs of the same privilege. A nonprivileged program may never use type-I linkage to call a privileged program; a privileged program may never use type-I linkage to call a nonprivileged program. Type-I Linkage involves these conventions:

1. Use of the standard save area.
2. Use of specific registers for designated functions.
3. Use of the branch-and-store instruction for the transfer of control from the calling program to the called.
4. Preservation of register integrity.

```
CHASAV  DSECT       FORMAT OF STANDARD 19-WORD SAVE AREA
        DS    0F    ALIGN ON WORD BOUNDARY
SAVLEN  DS    1F    LENGTH OF SAVE AREA AND APPENDAGES IN BYTES
SAVBPT  DS    1F    BACKWARD POINTER. ADDRESS OF SAVE AREA, IF ANY,
*                   USED BY CALLING ROUTINE
SAVFPT  DS    1F    FORWARD POINTER. ADDRESS OF SAVE AREA, IF ANY,
*                   SUPPLIED BY USER OF THIS AREA TO PROGRAMS IT CALLS
SAVR14  DS    1F    USED BY CALLED PROGRAM TO SAVE GPR 14
SAVR15  DS    1F    USE CALLED PROGRAM TO SAVE GPR15
SAVR0   DS    1F    USED BY CALLED PROGRAM TO SAVE GPR 0
SAVR1   DS    1F    USED BY CALLED PROGRAM TO SAVE GPR 1
SAVR2   DS    1F    USED BY CALLED PROGRAM TO SAVE GPR 2
SAVR3   DS    1F    USED BY CALLED PROGRAM TO SAVE GPR 3
SAVR4   DS    1F    USED BY CALLED PROGRAM TO SAVE GPR 4
SAVR5   DS    1F    USED BY CALLED PROGRAM TO SAVE GPR 5
SAVR6   DS    1F    USED BY CALLED PROGRAM TO SAVE GPR 6
SAVR7   DS    1F    USED BY CALLED PROGRAM TO SAVE GPR 7
SAVR8   DS    1F    USED BY CALLED PROGRAM TO SAVE GPR 8
SAVR9   DS    1F    USED BY CALLED PROGRAM TO SAVE GPR 9
SAVR10  DS    1F    USED BY CALLED PROGRAM TO SAVE GPR 10
SAVR11  DS    1F    USED BY CALLED PROGRAM TO SAVE GPR 11
SAVR12  DS    1F    USED BY CALLED PROGRAM TO SAVE GPR 12
SAVPCT  DS    1F    R (ENTRY POINT) THIS IS SET BY THE CALLING
*                   PROGRAM BEFORE TRANSFERRING CONTROL AND
*                   POINTS TO THE CONTROL SECTION IN WHICH THE
*                   ENTRY POINT IS DEFINED
```

Notes:

1. If a program is called and, in turn, calls another program, it must, upon receiving control, establish its own save area, save the address of the calling programs save area in the second word of its own save area and save the address of its own save area in the third word of the calling program's save area.

2. Field SAVPCT contains the R-con of the called program's entry point. This R-con may or may not be an address of a PSECT. If the called program was assembled without prototype control sections (PSECTs), the control section containing the ENTRY statement for the entry point being used by the calling program will be the control section pointed to by SAVPCT. See System/360 Assembler Language for more details concerning R- and V-type address constants.

Figure 3. Format of Standard Save Area

Type	Transfer Control via	Save-area Format	Parameter Registers (maximum)	Entry-Point Address in Register *	Return Address in Register	Save-Area Address in Register	PSECT Address in Register **
I (normal)	BASR	Standard	GPR 1	GPR 15	GPR 14	GPR 13	N/A
II	SVC 121 (ENTER)	Standard	GPR 1	GPR 15	GPR 14	GPR 13	N/A
IM/II	BASR or SVC 121 (ENTER)	Standard	GPR 1	GPR 15	GPR 14	GPR 13	N/A
III	SVC 254 (LVPSW)	Standard	GPR 1	GPR 15	GPR 14	GPR 13	N/A
IV (restricted)	BASR	None	GPR 0-6	GPR 15	GPR 14	None	GPR 13

* Also used for return code, if any.

** Very often, but not always, the PSECT and the save-area address are the same.

Figure 4. Virtual Program Linkage Conventions

Use of the Save Area

You will find it helpful to refer to Figure 3 for the following discussion. Whenever a program uses type-I linkage to call another program, the calling program must supply a save area for use by the called program. Prior to transferring control to the called program, the calling program puts certain information, if applicable, into the save area. The calling program is required to preset some fields of the save area; it may preset others. The first word of the save area (SAVLEN) must contain the length, in bytes, of the save area (minimum 76 bytes) and any appendages to it. The last, or 19th, word of the save area (SAVPCT) must contain the R-type address constant (R-con) of the entry point to which the calling progam is transferring control.

The R-value is the address of the control section in which the entry point is defined. It is possible, however, that the called program doesn't have any prototype control sections (PSECTs). In this case, the R-con will be the address of the control section containing the ENTRY statement for the entry point. In addition to naming an entry point, and R-value can use a control-section name or a program-module name. If a control-section name is used, the R-value will point to the beginning of the control section. If a program-module name is used, the R-value will point to the first prototype control section contained in the program module. If the module does not have any prototype control sections, the R-value will point to the first nonprototype control section.

One other field that must be set by the calling program, prior to transferring control to the called program, is the second word (SAVBPT) of the save-area. This field is an address pointer to a save area used by the calling program when it was a called program. The calling program may not be using a save area, though, and this field may contain zero. If this field does not contain zero, the called program may assume that it points to the save area being used by the calling program.

All other fields of the save area may contain anything. The called program should not assume that fields other than the length field, the R-con field, and the backward-pointer field contain meaningful information.

After receiving control, the called program must save the contents of all the general registers, except register 13, in the save area. The called program does not need to use the save area. If it does, it must save the registers in the exact fields designated for those registers.

At the time the called program receives control, register 13 will contain the address of the save area. The other registers are stored in the save area by using register 13 as a base address. For example, STM 14,12,12,(13) will save all the general registers, except 13, in the proper locations of the save area pointed to by register 13. If the called program wishes to use register 13 for its own purposes, it must save register 13 in the backward pointer of its save area. If the called program is going to call another program, and is going to provide a save area for that program, it must store register 13 in the second word of that save area. In this instance, register 13 serves as the backward pointer. Optionally, the user can store register 13 someplace else; not in a save area. Remember, if you save register 13 in a save area that you make available to another program, say program A, you depend on program A not to write over the save area you're letting it use. If program A is unreliable, you might want to save register 13 in an area accessible to your program alone. That precaution will enable you to restore the registers regardless of what the program you call does to the save area you are providing.

The called program does not need to save and restore the floating-point registers. If the contents of the floating-point registers are to be preserved, it is the responsibility of the calling program to save their contents and the contents of its interrupt mask.

Contents of the General Registers

Registers 13, 14, and 15 must be preset by the calling program. Register 13 must contain the address of the first byte of the save area that the calling program is providing for the called program. This address must be on a fullword boundary; i.e., the two low-order bits of the address must be zero. Register 14 must contain the address to which control is to be returned by the called program. Register 15 must be set to contain the address of the entry point in the called program to which control is being transferred.

A number of macro instructions can be used to generate type-I linkage to specific programs. Examples of these macro instructions are GET, PUT, OPEN, and CLOSE. Note, in using CALL, you specify the name of the program to which you wish to transfer control; in using GET, however, the name of the program to which control is to be transferred is supplied by the macro instruction.

The called program always uses register 15 as a return-code register, if applicable. If a parameter list address is passed in register 1, there must be an understanding between the called and calling programs as to its content. In variable length lists, the first word of the parameter list contains a count of the number of parameters in the list. Each following entry is the address of a parameter which has been prestored.

Transfer of Control

A type-I linkage always causes control to be transferred from the calling program to the called program by using the branch-and-store register (BASR) instruction. Specifically, the resident supervisor is never used to assist in transferring control (no interrupt occurs), since the calling and the called programs have the same privilege.

Register 1 may be preset by the calling program with the address of a parameter list. Register 1, 13, 14, and 15 are the only registers used by type-I linkage.

The CALL macro instruction should be used to generate a normal type-I linkage. The use of CALL is discussed in the Assembler User Macro Instructions.

EXAMPLE: A program transferring control to another program via a type-I linkage might use these instructions:

```
DEPART LR     6,13              MOVE SAVE AREA POINTER
       L      13, =A(SAVEREA)   LOCATION OF SPACE FOR STANDARD
*                                SAVE AREA

       USING  CHASAV,13         INDICATE FORMAT
       ST     6,SAVBPT          POINT TO CALLERS (OUR) SAVE AREA
       L      6, =R(SUBR)       GET R=CON OF CALLED PROGRAM
       ST     6, SAVPCT         STORE R-CON IN SAVE AREA
       L      15, =V(SUBR)      GET ADDRESS OF ENTRY POINT
       L      1,A(PARAMLIST)    SET POINTER TO PARAMETER LIST
       BASR   14,15             PUT RETURN ADDRESS IN GPR 14
*                                AND BRANCH
```

The program receiving control might use these instructions,

```
XYZ    PSECT
       ENTRY SUBR        MAKE NAME SUBR EXTERNAL
ABC    CSECT READONLY
SUBR   STM 14,12,12(13)  SAVE ALL REGISTERS
```

to save the general registers and establish definitions for the V-cons and R-cons of the name SUBR. When its processing is finished, the program SUBR might do this,

```
EXIT   LM 14,12,12(13)   RESTORE REGISTERS
       LA 15,4           SET RETURN CODE 4
       BR 14             RETURN TO CALLING PROGRAM
```

TYPE-II LINKAGE

Type-II linkage is used when the calling program is nonprivileged and the called program is privileged. All programs designed to be called via type-II linkage run in the privileged problem state. Type-II linkage involves these conventions:

1. Use of the standard save area.
2. Standardizing the content and usage of the general purpose registers.
3. Standardizing the method of transferring control from the calling to the called program.
4. Preservation of register integrity.

The Save Area

The standard format 19-word save area is used in type-II linkage (see Figure 3). Unlike type-I linkage, the calling program does not provide this save area. Instead, it is provided by the task monitor, which translates type-II linkage into what appears to the called program as modified type-I linkage. The transfer of control from the calling to the called program is through the supervisor when type-II linkage is used. Coding contained in the task monitor is, therefore, an integral part of the linkage process.

Prior to passing control to the called program, the task monitor initializes a save area for the called program's use. The length field (SAVLEN) contains a byte count of 76 (decimal); the backward pointer (SAVBPT) is zero. The last word of the save area (SAVPCT) contains the R-con of the entry point of the called program. All other bytes of the

save area are unpredictable. All programs designed to be called via type-II linkage can assume that the save-area pointed to by register 13 is arranged in this way.

Content and Usage of the General Registers

Type-II linkage conventions assign special functions to registers 0, 1, 13, 14, and 15. The calling program is responsible for presetting registers 0, 1, and 15. The calling program loads registers 0 and 1 with parameters or address pointers to parameter lists; it loads register 15 with a code, called an ENTER code. The purpose of the ENTER code is to identify the program to be called. When control is returned to the calling program, the contents of registers 2 through 14 will be unchanged. Registers 0 and 1 may be used by the called program for returning results; they do not need to be used, however, and may be unchanged. If the called program supplies a return code, it must use register 15; of course, the called program is not required to supply a return code.

The task monitor saves all the general-purpose and floating-point registers in its own save area; the task monitor builds a save area for the called program's use, as described in the previous section. The task monitor sets an address pointer to this save area in register 13. The contents of registers 0 and 1 are set as received from the calling program. Register 15 is set to the address in the called program to which the task monitor will transfer control. This address is determined by the task monitor, based on the ENTER code that was in register 15 when control was received by the task monitor. Register 14 is set to the address in the task monitor to which control is to be transferred by the called program when it has been completed. The contents of registers 2 through 12 are arbitrary; they should not be assumed, by the called program, to be significant.

The called program must save the contents of the general registers, since the task monitor requires the contents of the registers passed to the called program to remain unchanged. The called program must return control to the address in register 14. The called program may put a return code in register 15; it may put results in registers 0 and 1. Registers 0, 1, and 15 will be passed back to the calling program as they are received from the called program when it returns to the task monitor.

Transfer of Control

The calling program transfers control to the called program by issuing this instruction, SVC 121.

An SVC 121 is also generated by issuing the macro instruction ENTER. SVC 121 passes control through the task monitor to the called program. Most of the time ENTER is used as an inner macro instruction. For example, the macro instruction GETMAIN generates an ENTER if the program in which GETMAIN is issued has been declared by the programmer to be nonprivileged (DCLASS USER). All programs that transfer control via SVC 121 must adhere to type-II linkage conventions.

EXAMPLE: Assume you want to get 76 bytes of virtual storage, possibly for use as a save area; you might code it like this:

```
SR      1,1     SET OPTIONS:  NONPRIVILEGED, VARIABLE, BYTE
LA      1,76    BYTE COUNT 76
LA      15,48   ENTER CODE 48 -- GETMAIN (BYTE)
SVC     121     TRANSFER CONTROL
```

Control will be returned to the instruction following the SVC after GETMAIN has been completed. If you had wanted to use the macro instruction GETMAIN, you could have written, GETMAIN R,LV=76 which would have generated equivalent instructions.

TYPE-IM/II LINKAGE

Type-IM/II* linkage applies only to called programs that can be called via both type-I and type-II. Type-II called programs are always privileged programs. The calling program, however, may be privileged, in which case a modified type-I linkage is used (with both registers 0 and 1 usable as parameter registers); or the calling program may be nonprivileged, in which case type-II is used. Since the task monitor makes all type-II linkages appear, to the called program, like type-I, the called program is, generally, not affected by the privilege of the calling program.

If a privileged program is being called via type-IM/II, it may need to determine the privilege of the caller. It can do this by comparing the return address in register 14 to the address of the point in the task monitor to which control is returned when a type-II linkage has been used; for example,

```
CL    14,=V(CZCJER)      COMPARE GPR 14 TO TYPE-II RETURN
BE    NPCLLR             IF EQUAL, CALLER IS NONPRIVILEGED
```

The calling program uses either a type-IM or a type-II linkage as described previously; if the called program can be called by either of these linkage types, it is using type-IM/II. The calling program treats this linkage as described under type-II.

TYPE-III LINKAGE

Type-III linkage is used when the calling program is privileged and the called program is nonprivileged. All programs designed to receive type-III are designed to run in the nonprivileged state. Type-III linkage involves standardizing

1. The save area
2. The content and usage of the general-purpose registers
3. The method of transferring control from calling to called program

The Save Area

Type-III linkage requires the standard format 19-word save area; however, the save area is not supplied by the calling program. It is supplied by the leave-privilege subroutine. The calling program calls the leave-privilege subroutine, which supplies and initializes a save area for use by the called program (see Figure 3). The leave-privilege subroutine establishes a 19-word save area which is not read- or write-protected; the nonprivileged called program can access it. The leave-privilege subroutine sets the first word (SAVLEN) equal to 76. The last word of the save-area (SAVPCT) is loaded with the R-con of the called program's entry point. The calling program supplies this R-con

---

*The use of M with type-I linkage indicates that register 1 may also be used as a parameter register - in addition to register 0. Register 1 may contain a parameter or a pointer to a parameter list.

to the leave-privilege subroutine which inserts it into the save area. The remaining 17 words are left unchanged.

Content and Usage of General-Purpose Registers

Type-III linkage standardizes the use of registers 0, 1, 13, 14, and 15; the contents of the other registers are arbitrary. The contents of the other registers will be returned, intact, to the calling program. The leave-privilege subroutine will not pass to the called program the original contents of registers 2 through 12. Registers 0 and 1 are used for parameters or addresses of parameter lists. These registers are passed to the called program as received by the leave-privilege subroutine from the calling program. The leave-privilege subroutine loads register 13 with a pointer to the save area it is supplying for the called program. It loads register 15 with the address of the entry point in the called program to which it will transfer control. Then it loads register 14 with the address of an SVC 120 (RSPRV) instruction, which is in the part of the interrupt storage area (ISA) that nonprivileged programs can read and write.

Transfer of Control

Control is transferred from the calling program to the leave-privilege subroutine with the standard type-I linkage. Control is transferred from the leave-privilege subroutine to the called program by an SVC 254 (LVPSW). The use of type-III linkage always results in an SVC interrupt.

EXAMPLE: Within a privileged program, if you want to call a nonprivileged subroutine, you might write:

```
            L       13,=A(SAVEAREA)     GET ADDRESS OF SAVE AREA FOR LVPRV
            L       A,=R(CZCJLE)        R-CON OF LEAVE-PRIVILEGE SUBROUTINE
            ST      14,72(13)           PUT R-CON IN 19TH WORD OF SAVE AREA
            LA      1,PARAMTRS          PARAMETER LIST INTO GPR 1
            L       15,=V(CZCJLE)       ENTRY POINT OF LEAVE-PRIVILEGE
*                                       SUBROUTINE
            BASR    14,15               TRANSFER TO LEAVE-PRIVILEGE SUBR
PARAMTRS    DC      A(ADCONS)           POINTER TO V- AND R-CONS
            DC      A(PARAM1)           POINTER TO PARAMETER 1
            DC      A(PARAM2)           POINTER TO PARAMETER 2
ADCONS      DC      V(CALLED)           ENTRY POINT OF CALLED ROUTINE
            DC      R(CALLED)           R-CON OF ENTRY POINT
PARAM1      DC      F'P1'               PARAMETER 1
PARAM2      DC      F'P2'               PARAMETER 2
```

The leave-privilege subroutine will get space to set up a save area for use by the called program. It will load parameters one and two into general registers 0 and 1. It will set up registers 13, 14, and 15, as described previously, and transfer control to the called program via an SVC 254 (LVPSW).

When the called program is completed, it might return like this:

```
            LA      0,RESULT1           RETURN OF RESULTS
            LA      1,RESULT2           TO CALLING PROGRAM
            BR      14                  RETURN TO CALLER
```

General register 14 points to an SVC 120 (RSPRV) which will cause the restore-privilege routine to be entered. The restore-privilege routine will restore the calling routine's original register contents, without disturbing registers 15 (the return code register), 0, and 1 (the result registers).

TYPE-IV (RESTRICTED) LINKAGE CONVENTICNS

Type-IV linkage is used by TSS/360 programs under restricted circumstances for the sake of linkage efficiency. Type-IV is much more restricted general linkage than types I, II, and III. Type-IV linkage is found principally in the coding of the language processors.

Type-IV linkage may be used between two programs if, and cnly if, these conditions are met:

1. Both the called and the calling programs use the save prototype control section (PSECT).

2. The values of address constants required for the linkage have already been supplied by the dynamic loader.

3. The called program is not designed to accept type-I, -II, or -III linkage at the same entry point to be used for type-IV.

4. Both the called and and the calling programs have the privileges.

Type-IV linkage conventions standardize the use of the general registers and the method of transferring control from the calling to the called program. No provision for a standard save area is included in this convention.

Use of the General Registers

Registers 0 through 5 are used by type-IV linkage as parameter registers or as address pointers to parameter lists. These registers may be used by the calling program to supply information to the called program, or by the called program to return information to the calling program. In general, the calling program must not assume that the contents of any of these registers will be returned intact by the called program. It is the responsibility of the calling program to load the address of the common PSECT into register 13 before transferring control to the called program. The calling program must set, in register 15, the address of the entry point to which it will transfer control; the address to which control is to be returned is set in register 14. The called program uses register 15 as a return code register, if applicable. The contents of registers 6 and 7 are immaterial; the called program should not make assumptions about the contents of these registers. Registers 6 and 7 need not be saved by the called program.

The contents of registers 8 through 12 must be saved by the called program if the called program changes them. The calling program may establish any of registers 8 through 12 as common registers; the calling program may do this only if it has not been called via type-IV linkage. A common register is a register whose function is understood similarly by the calling and the called programs. If the function performed by a common register, such as pointing to a control block, is required by the called program, the called program may assume the contents of the common register can be used, as mutually understood between the calling and the called programs. The function of common registers must remain constant in all programs called, in turn, by the called program; their functicns must be returned intact to the calling program. The designation of common registers and the nature of their implied contents is not part of this convention; the nature of common registers is as mutually understood between the calling and the called programs.

Transfer of Control

Control is transferred from the calling to the called program by using the instruction

```
BASR 14,15
```

An interrupt must never take place because of a type-IV linkage.

EXAMPLE: Three macro instructions have been defined to assist your use of type-IV linkage: INVOKE, STORE, and RESUME. In order to emphasize the linkage coding itself, we will not use these macro instructions in this example. To transfer control using type-IV linkage, you might write:

```
LM    0,5,PARAMS          LOAD PARAMETERS INTO GPR 0-5
L     15,=A(ENTPOINT)     ENTRY POINT
BASR  14,15               TRANSFER CONTROL
```

The called program need, at most, save register 8 through 12 in any manner it chooses. It must ensure that the contents of these registers are returned intact to the calling program. The return might be coded:

```
LM    8,12,SCRATCH        RESTORE REGISTERS FOR CALLER
LA    15,RETCOD           RETURN CODE
BR    14                  RETURN
```

LINKAGE CONVENTION COMMENTS

In this discussion, we'll exclude type-IV linkage, which is found principally in the IBM-supplied application programs: Assembler, FORTRAN, and Linkage Editor. Type-IV linkage is used to minimize the overhead associated with program linkage by capitalizing on certain situations that occur in those programs.

We can look at program linkages in two ways: the calling program is the activator; it organizes the linkage information and transfers control. The called program has a more passive role; it receives control and assumes that the linkage information has been organized according to the rules. For some linkage types, a program is inserted between the calling and the called programs; this program performs some of the duties normally associated with the caller. In type-II linkage, the task monitor's ENTER routine is interposed between the calling and the called programs; in type-III, the LVPRV subroutine is between the calling and called programs.

From the viewpoint of the called program, most callers look the same. Type-I linkage doesn't use register 0; the other linkage types may. This is the principal difference from the called program's viewpoint. The called program may return the contents of register 0 to the caller when type-I is used; for the others, the contents of register 0, if not meaningful, can be ignored. Because of this similarity of appearance to the called program, many called programs can be written in much the same way. For instance, the SAVE macro instruction can be used to save the contents of the registers in the standard save area supplied by the calling program, and the RETURN macro instruction can be used to restore the registers, load a return code, and return control to the caller. The macro instructions SAVE and RETURN, therefore, apply not only to type-I linkage, with which they are most often associated, but also to types IM, II, IM/II, and III.

FENCE STRADDLERS

There are a number of programs in TSS/360 that have no built-in privileges; these programs assume the privilege of the calling program. Because these programs have no privileges, they are neither true privileged nor true nonprivileged programs; they are "on the fence," so to speak. We call them fence straddlers.

Fence straddlers must be constructed very carefully. If a nonprivileged program is using a fence straddler and the straddler is interrupted, it is quite possible that a privileged program will use the straddler during the period of interruption. The fence straddler must, therefore, be reenterable. This reenterability applies within the task. Programs may be reentered between tasks or within a task. The use of a prototype control section (PSECT) enables different tasks to use the same read-only control section. Within a task, however, a program is generally made up of one nonprototype control section (which may be shared with other tasks), and one prototype control section (which is never shared with other tasks).

Interruptable service routines, to be reenterable within a task, use multiple save areas and dynamically allocated virtual storage (via GETMAIN).

Fence straddlers can use a number of techniques to prevent destruction of data if they are interrupted and reentered. Some straddlers do not have a PSECT or, if they have one, never modify it. Other straddlers require the calling program to supply working storage; still others use GETMAIN to obtain working storage.

There are times when fence straddlers become calling programs. They must know their current privilege status so they can use the correct linkage type. There is one set way to determine privilege status; that is to check the privileged bit in the VPSW. The status depends on the function a fence straddler is performing. Parameters can be supplied by the calling program to tell the straddler what privilege it has. Sometimes the fence straddler can determine the privilege of the calling program by using information supplied by the calling program, such as the data control block (DCB).

Fence straddlers are called either type-I or type-IM linkage. Since the straddler assumes the privilege of the caller, there is no change in the privilege status. Thus, no interrupt is caused by calling a straddler and no linkage-assisting program is required.

SYSTEM PROGRAMMER AUTHORITY CODES

A programmer becomes known to TSS/360 as a system programmer when he is joined to the system by the system manager or one of the system administrators. The JOIN command contains an authority code which may have the values U (user), P (system programmer), or C (privileged system programmer). When a user logs on, information is taken from the user table specified in the JOIN command processor and inserted into the user's task status index and interrupt storage area. The SVC queue processor controls what programs are allowed to issue privileged SVCs; it uses the information LOGON stores in the task status index (TSIF4) for this purpose. The dynamic loader and program checkout system use information stored by LOGON in the interrupt storage area (ISAUTH) to determine if the task may perform certain privileged operations.

Privileged SVCs

The SVC queue processor controls the execution of SVCs 128 through 255. System programmers (P or O) may issue all the resident supervisor SVCs (128-255). Any program operating in the privileged-program state (VPSW p-bit =0) -- even if being run by a user-programmer -- may issue all the privileged SVCs.

Program Checkout System

The program checkout sub-system (PCS) is not, in general, applicable to system programs. The RUN command always transfers control in the

nonprivilege state and cannot be used to transfer to a privileged program. PCS uses the user save area of the task monitor and cannot, therefore, be used to set or display registers or VPSWs in use by privileged programs. Nevertheless, PCS can be used by privileged system programmers to modify (commands AT and SET) privileged, public control sections; this is the only way that privileged, public control sections can be changed from a terminal. (See the discussion of PCS in the section on serviceability aids.)

Dynamic Loader

Every task has a task dictionary (TDY) which contains, in addition to the PMDs, the hash tables used by the dynamic loader to process symbolic definitions (DEFs) and references (REFs). The hash table is split into three parts: privileged system, nonprivileged system, and user symbols. (See Figure 5.) When the loader encounters a REF in a control section with the attribute of PRVLGD, it searches the privileged system hash table; if the attribute of the control section is nonprivileged SYSTEM, the nonprivileged system hash table is searched; if the attribute of the control section indicates that it is a user's, then the user hash table is searched. One exception to this rule is that case in which the user's authority code is P or O. In this case the loader ignores the user hash table and searches the two system tables. Table 1 summarizes the actions of the loader in processing the REFs and DEFs.

Notice that the loader erases the attributes of PUBLIC, READONLY, SYSTEM, and PRVLGD from any module loaded from any library for a programmer with authority code O. The same attributes are erased from any module loaded from JOBLIB or USERLIB for a programmer with authority code P. If a programmer with authority code P loads a module from SYSLIB, only the PUBLIC and READONLY attributes are erased.

Remember, though, the loader does not load initial virtual memory (IVM); you will always get a public, read/write protected copy of IVM. The loader's action in assigning storage keys to control sections is governed by the attributes of those sections. (See Table 1 and Figure 6.)

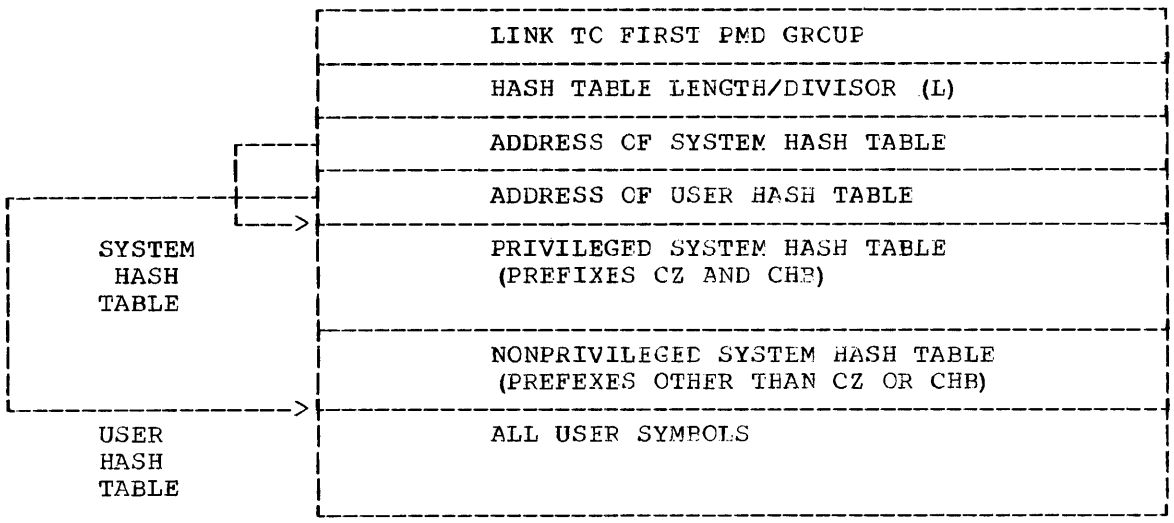


Figure 5. Format of Three-Part Hash Table

Table 1. Effect of Authority Code in Dynamic Loader Processing

ISAUTH set to:	If section containing reference is:	Search this hash table to resolve reference:	If hash table yields no definition, search this library first:	If section containing definition is found in this library:	And attributes of section containing definition are:	Erase these attributes in definition section:	Consider definitions starting with these characters invalid:	Put valid definitions in this hash table:
U	nonsystem	USER*	JOBLIB	JOB or USER	any	S, P	SYS	USER
U	nonsystem	USER*	JOBLIB	SYSLIB	not S or P	none	SYS	USER
U	nonsystem	USER*	JOBLIB	SYSLIB	S	none	CZ or CHB	SYSTEM**
U	nonsystem	USER*	JOBLIB	SYSLIB	P	none	not CZ or CHB	SYSTEM**
U	system	SYSTEM	SYSLIB	SYSLIB	not S or P	none	SYS	USER
U	system	SYSTEM	SYSLIB	SYSLIB	S	none	CZ or CHB	SYSTEM
U	system	SYSTEM	SYSLIB	SYSLIB	P	none	not CZ or CHB	SYSTEM
P	any	SYSTEM	JOBLIB	JOB or USER	any	Pb,RO,S,P	CZ or CHB	SYSTEM
P	any	SYSTEM	JOBLIB	SYSLIB	any	Pb,RO	not CZ or CHB	SYSTEM
O	any	SYSTEM	JOBLIB	any	any	Pb,RO,S,P	none	SYSTEM

Legend

U = User authority code
S = System programmer authority code
O = Privileged system programmer authority code

S = SYSTEM control section attribute
P = PRVLGD control section attribute

Pb = PUBLIC control section attribute
RO = READONLY control section attribute

Notes

* Symbols starting with SYS are always defined in the system hash table; nonsystem programs are not permitted to define SYS symbols. Any program can reference SYS symbols; the loader always looks for SYS symbols in the system hash table.

** A user causing a system program to be loaded will only be able to use the definitions supplied by a program that begins with SYS. These two cases will result in the transmission of a diagnostic message if the REF doesn't begin with SYS indicating an undefined symbol, even though the program supplying the proper definitions has had it's PMD hashed into the system hash table.

PRIVILEGED PROGRAMS

Privileged programs are virtual programs recognized by having their virtual program status word (VPSW) privilege bit (bit 0) set to 0, analogous to the problem bit (bit 15) in the real PSW. Privileged programs differ from nonprivileged user programs in two principal ways: they operate with a PSW protection key of zero and they may issue most supervisor call instructions. Privileged programs can access all virtual storage in their own virtual machine; they cannot access private virtual storage in other virtual machines.

Privileged programs exist to provide services to nonprivileged programs. Privileged service routines that can be called by users are "connected" to the task monitor through a table, called the ENTER table (see ENTER); other privileged service routines are closed subroutines used only by privileged callers.

I/O DEVICE ADDRESSING

Generally, each I/O device appears to have its own data channel. The initiation of a virtual I/O operation does not require the use of the interrupt storage area in the way a real I/O operation requires the use of the prefixed storage area to start an I/O operation. Virtual channel programs are constructed using I/O request control block (IORCB) and channel command words that are similar to real CCWs (see discussion of IOCAL). All I/O operations in a virtual machine needn't use IORCB

channel command sequences, though. Some I/O operations such as Vitual Access Method (VAM) operations are performed by using two special supervisor call instructions which take advantage of the of the characteristics of the address translator (see decriptions of PGCUT and SETXP).

Class of TSS/360 Program Module	CSECT Types For Modules	Attributes	Resultant Segment Assignment And Storage Key	
			Public	Private
SYSTEM PRIVILEGED	REENTERABLE EXECUTABLE CODE	CSECT SYSTEM, PRVLGD FXL,PUB,RDC	C	-
e.g., VAM OPEN	DATA, ADCONS, etc.	PSECT SYS,PRVLGD, FXL	-	C
SYSTEM FENCE SITTER	REENTERABLE EXECUTABLE CODE	CSECT SYS,PUB RDO,FXL	B (USER READ CNLY)	-
e.g., VAM GET	NONMODIFIABLE DATA, etc.	PSECT SYS, RDO,FXL	-	B (USER READ ONLY)
SYSTEM NONPRIVILEGED	REENTERABLE EXECUTABLE CODE	CSECT SYS,PUB RDO,FXL	B (USER READ CNLY)	-
e.g., ASSEMBLER LPC	DATA, ADCONS, etc.	PSECT SYS,FXL	-	A (USER READ WRITE)
A=Key 1 B=Key 2 C=Key 2 with fetch protection				

Figure 6. Relationship of TSS/360 Program Modules, CSECT, CSECT Attributes, Sharability, and Storage Key Assignment

Because most I/O devices attached to the system have more than one path to storage, these devices have multiple real addresses. The supervisor's pathfinding program has the responsibility of selecting the address to be used to access an I/O device. To distinguish an I/O device from the path (i.e., address) used to access it, each device attached to the system is given a unique number, called the symbolic device address. The assignment of symbolic device numbers is unique at each installation.

In addition to the symbolic device address, some I/O operations require the use of a relative page number. The relative page number is a 16-bit quantity allowing a device to have 65,536 pages. For certain I/O operations (for example PGCUT), each device is organized into pages; since each page is 4096 bytes, the position of a given page on all devices of the same type can be determined. Thus, page 136 begins at the same cylinder, track, and record address for all IBM 2311s. In other words, given a relative page number and a device type, it is always possible to figure out where, on that device, the page can be found.

The system symbolic device address and the relative page number, together, make up the external page address; they uniquely identify the location of a page on external storage.

Figure 7 shows the significance of various combinations of PSW and storage keys and the programs to which they may be assigned. Those within the heavy line designate nonprivileged user key combinations. The other combinations are available only to privileged system programs.

Storage protect key 2F is the same as key 2 but with fetch protecticn added.

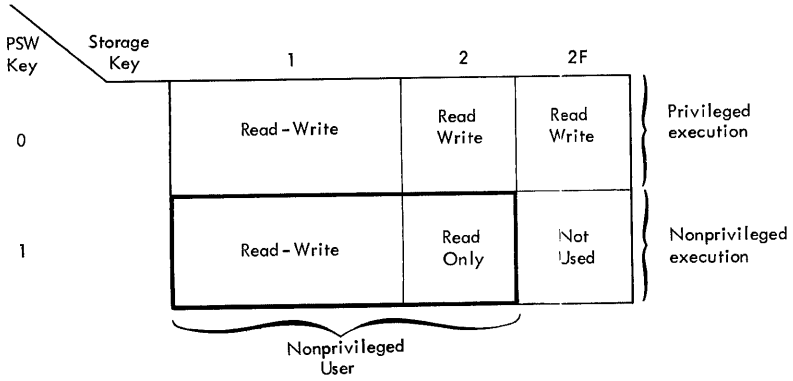


Figure 7. PSW and Storage Protection Keys

STORAGE PROTECTION

Although the virtual program status word doesn't contain a PSW key, storage protection is in effect for virtual programs. The resident supervisor assigns storage keys to virtual programs when it creates external page table entries for them; it sets keys in the main storage pages it allocates (see ADDPG and ADSPG). All main storage pages are assigned a storage protection key; Table 2 illustrates these assignments.

Table 2. Main Storage Page Key Assignments

Type of Page	Key	Fetch Protection Bit
Nonprivileged read/write	1	off
Nonprivileged read-only	1	off
Privileged	2	on
Engaged in paging operation	3	off
Storage obtained from supervisor core allocation	4	off
Resident supervisor	5	on

The ability of a processing unit or a data channel to access storage is controlled by the protection key contained in storage and the key used by the processing unit (PSW) or data channel (CAW). The resident supervisor assigns keys to programs and channel programs before starting them; these assignments are shown in Table 3.

Table 3. Processing Unit and Data Channel Key Assignments

Category	Key
Processing Unit Programs	
Nonprivileged	1
Privileged	0
Resident supervisor	0
Data Channel Programs	
Nonprivileged I/O	1
Privileged I/O	2
Paging I/O	3
IORCB I/O	4
Sense Data I/O	4

TIMEKEEPING

There are a number of different cells used by TSS/360 to store information about elapsed time, estimated time, and related data. Data concerning calender time is kept in several places.

The prefixed storage area of each processing unit includes a double-word, called PSAETM, which contains the number of 13-microsecond "ticks" that the processor's interval timer has been running since PSAETM was last cleared to zero (see RSTTIM). Every time an interrupt occurs, and every time a virtual task is restarted, the elapsed time since the timer was previously loaded is added to the contents of this cell.

The system table contains two double-words that record time for the entire system. The calendar date (year, month, day), measured in microseconds from March 1, 1900 to midnight of the previous day, is stored in a doubleword called SYSYMD. March 1, 1900 is chosen as the starting date since, from that date, every year divisible by four is a leap year (366 days); century years are not leap years. A double word called SYSTOD contains the time of day measured in microseconds from midnight. By adding SYSYMD, SYSTOD, and PSAETM we can obtain a current value in microseconds from March 1, 1900.

Information concerning the elapsed time for individual tasks is kept in each task's extended task status index (XTSI). The total number of microseconds of time quanta a task has received since LOGON is stored in a word called XTSATI. A word called XTSCTI contains the number of microseconds of time quanta a task has received since it's last task-timer interrupt. The number of microseconds of time quanta that must elapse before the next task-timer interrupt is stored in a word called XTSUTI (see SETTU).

Appendix B describes a time conversion module that may be used by system programmers.

INITIAL VIRTUAL STORAGE

The task monitor and a number of programs (both privileged and nonprivileged) are collected into what is called initial virtual storage, (IVS). Startup establishes initial virtual storage by constructing a standard set of segment and page tables to be used by newly created virtual machines. Initial virtual storage programs are never dynamically loaded; we think of them as being permanently resident in virtual storage. Of course, IVS programs are paged in and out of main storage. All other privileged programs are brought into virtual storage, as required, by the dynamic loader (which must be part of IVS). Virtual storage is never "empty;" it always contains at least the programs that make up the IVS.

PRIVILEGED SUPERVISOR CALL INSTRUCTIONS

If a nonprivileged program being run by a user-programmer attempts to issue a privileged supervisor call, the resident supervisor will create a task program-interrupt (code hex 50 or hex 21 -- privileged operation). When the task monitor receives the interrupt, it calls DIAGNC. Generally, supervisor calls that can disrupt the operation of TSS/360 are privileged. Supervisor calls that allow access to private information are also privileged.

Usually, the operation requested by a privileged SVC is a synchronous one which is completed by the resident supervisor before it returns control to the task which issued the SVC. The principal exception to

this is IOCAL, an asynchronous SVC, which is processed concurrently with the issuing task. For simplicity of explanation, privileged supervisor calls are divided into nine groups (see Table 4). (Note: task program interrupts, which may result from improper use of these macro instructions, can be found in Appendix D.)

Table 4. Privileged Supervisor Calls (SVC 128-255) (Part 1 of 2)

MAINTENANCE OF TASK STATUS INDEX		
SVC 253	Create task status index	CRTSI
SVC 206	Special create task status index	SCRTSI
SVC 252	Delete task status index	DLTSI
SVC 235	Set up task status index field	SETUP
SVC 246	Extract task status index field	XTRCT
SVC 214	Set up extended task status index field	SETXTS
SVC 213	Extract extended task status index field	XTRXTS
SVC 230	Change task priority	CHAP
SVC 209	Extract accumulated CPU time	XTRTM
MAINTENANCE OF SYSTEM TABLE		
SVC 216	Set system table field	SETSYS
SVC 215	Extract system table field	XTRSYS
SVC 212	Reset system time	RSTTIM
SVC 216	Allow task initiation	ALLTI
SVC 216	Set year, month, and day	SETYMD
SVC 216	Set time of day	SETTOD
SVC 201	Reset Drum Interlock	RDI
TASK SYNCHRONIZATION/TASK TIMER MAINTENANCE		
SVC 251	Set user timer	SETTU
SVC 217	Set real time interval	SETTR
SVC 218	Read elapsed real time	REDTIM
SVC 243	Force time slice end	TSEND
SVC 248	Wait for an interrupt	AWAIT
SVC 229	Wait for terminal I/O interrupt	TWAIT
VIRTUAL STORAGE ALLOCATION		
SVC 250	Add virtual storage pages	ADDPG
SVC 236	Add shared virtual storage pages	ADSPG
SVC 249	Delete virtual storage pages	DELPG
SVC 238	Connect segment to shared page table	CNSEG
SVC 237	Disconnect shared page table from segment	DSSEG
SVC 247	List changed virtual storage pages	LSCHP
SVC 241	Check protection class	CKCLS
DEVICE MANAGEMENT		
SVC 234	Add device to task symbolic device list	ADDEV
SVC 233	Remove device from task symbolic device list	RMDEV
SVC 222	Purge I/O operations	PURGE
SVC 221	Reset device suppression flag	RESET
SVC 211	Set I/O device path	SPATH
SVC 210	Set asynchronous entry	SETAE

(Continued)

Table 4. Privileged Supervisor Calls (SVC 128-255) (Part 2 of 2)

I/O OPERATIONS		
SVC 231	I/C call	IOCAL
SVC 242	Write virtual storage pages to external storage	PGOUT
SVC 244	Set external page table entries	SETXP
SVC 245	Move page table entries	MCVXP
STATUS SWITCHING		
SVC 254	Load virtual program status word	LVPSW
INTERTASK COMMUNICATION		
SVC 240	Send message to another task	VSEND
ERROR RECOVERY and RECONFIGURATION		
SVC 254	Indicate supervisor detected error	ERROR
SVC 228	Indicate nonresident-program detected error	SYSER

CRTSI -- Create Task Status Index (R)

The CRTSI allocates storage for a TSI and initializes it for a new task if the system limit on TSIs has not been reached.

Name	Operation	Operand
[symbol]	CRTSI	None

EXECUTION: A new task status index is created if the system TSI limit has not been reached. The task identification is returned to the SVC issuing program in register 1; if the system TSI limit has been reached, register 1 is set to 0.

The TSI created is initialized like this:

1. XTSI page count set to 1 (TSINX)
2. XTSI pointer set to system skeleton (TSIXXI)
3. All task-interrupt mask bits are set to 1
4. The conversational bit is set on (TSICV)
5. The XTSI swapped out bit is set on (TSIIXT)
6. The TSI internal priority is set to the highest possible second-level priority (TSIUPR)
7. The task identification number in the system table (SYSTID) is incremented by 1 and is placed in the TSI as the task identification (TSITID)

EXAMPLE: If you want to create a TSI, you might write:

```
XYZ    CRTSI
```

This would generate,

```
XYZ    SVC 253
```

Note: This SVC must be used in conjunction with VSEND. See IBM System/360 Time Sharing System: Task Monitor (Form Y28-2041) for a description of the action of the external interrupt processors (XIP).

SCRTSI -- Special Create Task Status Index (R)

The SCRTSI macro instruction allocates storage for and initializes a TSI the same as does CRTSI, but does so regardless of the number of TSIs presently in existence.

Name	Operation	Operand
[symbol]	SCRTSI	None

EXECUTION: A new task status index is created regardless of the number of TSIs currently in existence. The task identification is returned to the SVC issuing program in register 1; if the system TSI limit has been reached, it is incremented until the new TSI will not exceed the limit. After the TSI has been created, the system limit value is restored.

The TSI created is initialized the same as in the CRTSI macro instruction discussed above.

EXAMPLE: If you want to create a TSI, you might write:

```
ABC     SCRTSI
```

This would generate,

```
ABC     SVC 206
```

DLTSI -- Delete Task Status Index (R)

The DLTSI macro instruction deletes the specified TSI and removes its associated task from the system.

Name	Operation	Operand
[symbol]	DLTSI	None

EXECUTION: The task issuing the SVC is eliminated from the system. All nonshared pages of main storage and paging storage used by the task are returned for reallocation. All storage required for table entries pertaining to the task in the resident supervisor is released.

EXAMPLE: If your program is completed and you wish to release all the resources it is using -- logically eliminating the task -- you might write:

```
RJG     DLTSI
```

This would generate,

```
RJG     SVC 252
```

Note: This SVC is the last step in a sequence required for removing a task. Other steps include closing data sets and releasing external storage.

SETUP -- Set Up Task Status Index Field (R)

The SETUP macro instruction permits you to alter or set the contents of a selected field in the TSI.

Name	Operation	Operand
[symbol]	SETUP	[field {code / (15)}][, register -{value / (1)}]

field
specifies the field you want to set or alter and may be specified as one of these codes:

USERID - set the user identification field
SYSIN - set the input data set location field
SYSCUT - set the output data set location field
BSN - set the batch sequence number field
CONV - set the intertask message flag
ITMFLG - set the intertask message flag
XPR - set the external priority flag
AUTH - set the privilege field

If you choose to write the instruction as register notation you must first select the proper value from the following list and place it in register 15 in the low-order byte.

Field	Value
USERID	1
SYSIN	3
SYSOUT	4
BSN	5
CONV	10
ITMFLG	12
XPR	13
AUTH	14

register
designates the even-odd register pair in which you have placed the information you want put into the specified TSI field. The register pair must be specified in terms of the odd register and may not be an external symbol or an expression containing an external symbol. If you wish to place this information in registers 12 and 13 you would write:

```
        SETUP   USERID,DATA
```

where:

```
DATA    EQU     13
```

or you could write:

```
        SETUP   (15),(1)
```

where register 15 contained the value one and registers 1 and 0 contained the appropriate values.

EXECUTION: From one to eight bytes of registers 0 and 1 are inserted into the task status index field specified by the low-order byte of register 15. The number of bytes to be inserted depends on the field specified.

Field	Code	Implied length (bytes)
USERID	1	8
SYSIN	3	2
SYSOUT	4	2
BSN	5	1

CONV	10	1
ITMFLG	12	1
XPR	13	2
AUTH	14	1

EXAMPLE: Assume that registers 12 and 13 contain an eight-character user identification. The macro instruction

```
TEST   SETUP    USERID,(13)
```

causes this code to be generated

```
TEST   DS       0H
       LA       15,1
       LR       0,13-1
       LR       1,13
       SVC      235
```

XTRCT -- Extract Task Status Index Field (R)

The XTRCT macro instruction permits you to extract and examine one of a selected number of fields from your TSI.

Name	Operation	Operand
[symbol]	XTRCT	[field -{code / (15)}]

field
designates the TSI field you want to extract and examine and may be any one of these codes

USERID - extract the user ID field
PRICRITY - extract the priority field
SYSIN - extract the input data set symbolic device address
SYSCUT - extract the output data set symbolic device address
BSN - extract the batch sequence number field
SOPRIV - operator privilege
SPRIV - system programmer, nonprivileged
SRPRIV - system programmer, privileged
UPRIV - user
CONV - extract the conversational task flag
TASKID - extract the task ID field
XPR - extract the external priority flag
ITMFLG - extract the intertask message flag
AUTH - extract the privilege field
PENDIO - extract the pending I/O operations count field

If you choose to write register notation, select the proper value from the following list and place it in register 15 before issuing the macro instruction.

Code	Value
USERID	1
PRIORITY	2
SYSIN	3
SYSOUT	4
BSN	5
SOPRIV	6
SPPRIV	7
SRPRIV	8
UPRIV	9
CONV	10

TASKID	11
ITMFLG	12
XPR	13
AUTH	14
PENDIO	15

EXECUTION: The task status index field indicated by the code contained in register 15 is extracted and returned to the program issuing the XTRCT. The extracted field is returned right-justified in registers 0 and 1. The number of bytes returned is:

Register 15	TSI Field	Implied Length (bytes)
1	TSIUID	8
2	TSIUPR	1
3	TSISIN	2
4	TSISOT	2
5	TSIBSN	1
6	TSIIOP (TSIF4)	1 (bit 0)
7	TSIIPP (TSIF4)	1 (bit 1)
8	TSIISP (TSIF4)	1 (bit 2)
9	TSIUP (TSIF4)	1 (bit 3)
10	TSICV (TSIF2)	1 (bit 5)
11	TSITID	2
12	TSIMB (TSIF4)	1 (bit 6)
13	TSIXPR	2
14	TSIF4	1
15	TSICIO 15	1

The smallest field size extracted is one byte. If you are interested in a particular bit within a byte, you must mask out the remaining bits.

EXAMPLE: Suppose you want to find out if your task is being run in the conversational mode; you might write:

```
EXAMP           XTRCT           CONV
```

This would generate,

```
EXAMP           DS              0H
                LA              15,10
                SVC             246
```

SETXTS -- Set Up Extended Task Status Index Field (R)

The SETXTS macro instruction enables you to set the estimated run time of your task in the XTSI.

Name	Operation	Operand
[symbol]	SETXTS	[field-{ESTIM / (15)}]

field
specifies the XTSI field to be set by this macro instruction and may be coded: ESTIM - indicates that the estimated run time field of the XTSI is to be set.

If you select the register notation, by design or by default, register 15 must contain the value 1.

EXECUTION: The value contained in registers 0 and 1 when SETXTS was issued is stored in the extended task status index field indicated by the code contained in register 15. Only this code is defined,

Code	Implied Length (bytes)
1	4

EXAMPLE: Suppose you wished to set the estimated run time field of the extended task status index. You could write:

```
        L       0,F'runtime'
NAME    SETXTS  ESTIM
```

This would be produced,

```
NAME    DS      0H
        LA      15,1
        SVC     214
```

XTRXTS -- Extract Extended Task Status Index Field (R)

The XTRXTS macro instruction enables you to extract and examine one of a selected set of XTSI fields.

Name	Operation	Operand
[symbol]	XTRXTS	[field-{UTIME / ATIME / (15)}]

field
designates the XTSI field you want to extract and examine and may be specified by one of these codes:

UTIME - extract the user time field
ATIME - extract the accumulated time field

If you choose to write register notation, select the proper value from the list below and place it in register 15 before issuing the macro instruction

Code	Value
UTIME	1
ATIME	2

EXECUTION: Register 0 is loaded with information extracted from the issuing task's extended task status index. The field extracted depends on the code contained in register 15. These codes are acceptable:

Code	Implied Length (bytes)
1	4
2	4

EXAMPLE: Suppose you want to find out how much time your task had used since LOGON; you might write:

```
NAME    XTRXTS  ATIME
```

This will generate,

```
NAME    DS      0H
        LA      15,2
        SVC     213
```

CHAP -- Change Task Priority (R)

The CHAP macro instruction lets you change the priority of your task.

Name	Operation	Operand
[symbol]	CHAP	[prior -{value / (0)}]

prior
designates the new priority you want assigned to your task and may be any value from 0 to 255. If you choose to write register notation, place the new priority in register 0 before issuing the macro instruction.

EXECUTION: The low-order byte of register 0 is stored in the task status index's priority field. The TSI is removed from the TSI list and restored to a position in the list corresponding to its new priority. If the priority supplied in register 0 is equal to 0, the system default priority is used.

EXAMPLE: Suppose you want to give your task maximum priority; you might write:

```
NEW     CHAP     1
```

This would generate,

```
NEW     DS       0H
        LA       0,1
        SVC      230
```

XTRTM -- Extract Accumulated CPU Time (nonstandard)

The XTRTM macro instruction enables you to extract and examine the total CPU time used by your task.

Name	Operation	Operand
[symbol]	XTRTM	None

EXECUTION: The total accumulated CPU time of the issuing task is computed and returned to the task in general register 1. The address of the task's XTSI is passed to the SVC processor. The accumulated time is computed by subtracting the current timer value from the last time slice value and adding the accumulated time value to the difference.

EXAMPLE: If you want to extract your task's accumulated time, you might write:

```
TIME    XTRTM
```

This would generate:

```
TIME    SVC 209
```

SETSYS -- Set System Table Field (R)

The SETSYS macro instruction allows you to set or alter one of a selected set of system table fields.

Name	Operation	Operand
[symbol]	SETSYS	[field-{TOD / YMD / TASKINIT / (15)}]

field
designates the system table field you wish to set or alter and may be written:

TOD - set time of day field
YMD - set year, month, day field
TASKINIT - set task initiation status field

If you choose to write register notation, you must select the proper value from the list below and place it in register 15 before you issue the macro instruction.

Code	Value
TOD	1
YMD	2
TASKINIT	3

EXECUTION: The contents of registers 0 and 1 are placed in the system table field corresponding to the code contained in register 15. The number of bytes to be inserted into the system table depends on the code:

Code	Implied Length (bytes)
1	8
2	8
3	1

EXAMPLE: Suppose you wish to inhibit the initiation of any further tasks. This is controlled by a flag byte in the system table called SYSTI; the appropriate bit mask (see dummy section usage) is called SYSTIM. The task initiation bit happens to be bit 2 of the SYSTI byte. SETSYS replaces the entire flag byte, so an XTRSYS should be used to extract the flag byte. Then you might write:

```
DOUG   SETSYS    TASKINIT
```

This would generate

```
DOUG   DS  0H
       LA  15,3
       SVC  216
```

XTRSYS -- Extract System Table Field (R)

The XTRSYS macro instruction enables you to extract and examine one of a selected set of system table fields.

Name	Operation	Operand
[symbol]	XTRSYS	[field-{TOD / YMD / TASKINIT / (15)}]

field
designates the system table field you wish to extract and examine and must be one of these codes:

TOD - extract the time of day field
YMD - extract the year, month, day field
TASKINIT - extract the task initiation status field

If you choose to write register notation, you must select the proper value from the list below and place it in register 15 before issuing the macro instruction.

Code	Value
TOD	1
YMD	2
TASKINIT	3

EXECUTION: A number of bytes are extracted from the system table and placed in registers zero and one. The number of bytes and the field to be extracted is determined by the code contained in register 15. Register 15 must contain one of these codes:

Code	Implied Length (bytes)
1	8
2	8
3	1

EXAMPLE: Suppose you want to learn the time of day (in microseconds from midnight to issuance of RSTTIM).

You might write:

```
NAME    XTRSYS    TOD
```

The expansion would produce,

```
NAME    LA        15,1
        SVC       215
```

RSTTIM -- Reset System Time (nonstandard)

The RSTTIM macro instruction enables you to update the time of day field, in the system table, to reflect additional elapsed time since the last setting of that field.

Name	Operation	Operand
[symbol]	RSTTIM	None

EXECUTION: The supervisor adds the elapsed time cell (PSAETM) to the system table field SYSTOD and then sets the elapsed time cells (PSAETM) in each prefixed storage area to zero. If the resulting time of day value exceeds 24 hours, one is added to the year-month-day cell (SYSYMD) in the system table and a value equivalent to 24 hours is subtracted from the time-of-day clock.

EXAMPLE: Suppose you want to reset all the elapsed time cells of all processing units in the system, possibly as a restart procedure. You might write:

```
BLAST   RSTTIM
```

This would produce,

```
BLAST   SVC     212
```

ALLTI -- Allow Task Initiation (R)

The ALLTI macro instruction enables you to allow or disallow task initiation for your task.

Name	Operation	Operand
[symbol]	ALLTI	action-{OFF / ON}

action
indicates whether you wish task initiation allowed or disallowed and must be one of these codes:

ON - task interrupts are to be enabled
OFF - task interrupts are to be disabled

EXECUTION: The ALLTI macro instruction examines the operand and generates either

```
SR   1,1   if it is OFF or
LA   1,1   if it is ON
```

after generating SR 0,0 in either case. AIITI then invokes SETSYS as an inner macro instruction, passing it the above code in registers 0 and 1 and TASKINIT as parameters.

EXAMPLE: If you want to enable task initiation you should write:

```
STOPME   ALLTI   OFF
```

This would generate:

```
STOPME   SR      0,0
         SR      1,1
         SETSYS  TOD
```

SETYMD -- Set Year, Month, and Day (nonstandard)

The SETYMD macro instruction enables you to set the year, month, and day field of the system table.

Name	Operation	Operand
[symbol]	SETYMD	None

EXECUTION: The SETYMD macro instruction invokes SETSYS as an inner macro instruction and passes it YMD as a parameter. You must preload registers 0 and 1 with the year, month, and day.

SETTOD -- Set Time of Day (nonstandard)

The SETTOD macro instruction enables you to set the time of day field in the system table.

Name	Operation	Operand
[symbol]	SETTOD	None

EXECUTION: The SETTOD macro instruction is used to set the time of day by issuing the SETSYS inner macro and specifying the TOD field. You must preload registers 0 and 1 with the time of day.

RDI -- Reset Drum Interlock (nonstandard)

The RDI macro instruction enables you to reset the task/task drum interlock byte contained in the system table.

Name	Operation	Operand
[symbol]	RDI	None

EXECUTION: The task ID of the caller is matched against that of the drum (TT) interlock and causes the PSW condition code (contained in the XTSI) to be set as follows:

0 (the drum interlock is cleared)
1 (the drum interlock was not cleared because the task ID of the issuing program did not match that of the TT interlock)
2 (the drum interlock was not found set)

The use of the RDI macro instruction results in the generation of SVC 201.

SETTU -- Set User Timer (R)

The SETTU macro instruction enables you to set the user timer field in the XTSI thereby limiting your task's execution time.

Name	Operation	Operand
[symbol]	SETTU	[time-{value / (1)}]

time
specifies the time duration, expressed in milliseconds, which you want placed in the user timer field. It may be any value from 0 to 55,364,812.

EXECUTION: The quantity contained in register 1 is converted to a multiple of 13-microsecond "ticks" and stored in the extended task status index field called user timer value (XTSUTI).

EXAMPLE: Assume register 5 contains the number of milliseconds to which you'd like to set the user timer. The macro instruction

```
NAME    SETTU     (5)
```

will produce

```
NAME    DS        0H
        LR        1,5
        SVC       251
```

SETTR -- Set Real Time Interval (nonstandard)

SETTR enables you to set a time limit, in terms of a real time, on the execution of your task.

Name	Operation	Operand
[symbol]	SETTR	None

EXECUTION: When the doubleword time-of-day cell in the system table equals or exceeds the time, in microseconds, supplied in registers 0 and 1, a task-timer interrupt is generated for the task issuing the SETTR. If the system limit for queuing real time interrupt requests has been reached, a condition code of X'10' is returned to the SVC issuing program.

EXAMPLE: Suppose you want to receive a task-timer interrupt at three o'clock (54,000,000,000 microseconds after midnight). You could write:

```
TIME    DS      0D
        DC      FL8E9'54'
PROC    LM      0,1,TIME
NAME    SETTR
```

SETTR will generate:

```
NAME    SVC     217
```

REDTIM -- Read Elapsed Real Time (nonstandard)

The REDTIM macro instruction enables you to read the system time in microseconds.

Name	Operation	Operand
[symbol]	REDTIM	None

EXECUTION: The year, month, day cell (SYSYMD) in the system table (CHASYS) is added to the time-of-day cell (SYSTOD) in the system table and to the elapsed-time cell (PSAETM) in the prefixed storage area giving the current instant in microseconds. The resulting double-precision fixed-point number is returned in registers 0 and 1.

EXAMPLE: Suppose you want to find the date and time. You might write:

```
NAME    REDTIM
```

This will generate:

```
NAME    SVC     218
```

TSEND -- Force Time Slice End (R)

The TSEND macro instruction enables you to impose a time slice end on your task prematurely.

Name	Operation	Operand
[symbol]	TSEND	None

EXECUTION: The current time slice of the task issuing the SVC is terminated. The task becomes a candidate for another time slice in the next operational cycle.

EXAMPLE: Suppose you want to cause your current time slice to come to an end. You might write:

```
XYZ     TSEND
```

This would generate:

```
XYZ     SVC     243
```

AWAIT -- Wait for an Interrupt (R)

The AWAIT macro instruction enables you to check for the completion of an event and to enter your task into the delay state to await completion.

Name	Operation	Operand
[symbol]	AWAIT	None

EXECUTION: The AWAIT routine checks to see if the SVC was the subject of an execute (ILC=2) and if the SVC lies on the second halfword of a fullword (implying an event control block). If both these conditions are satisfied, the event control block complete bit (bit 1 of the first byte) is checked. If this bit is on, the event is complete, no waiting is required, and control is returned to the issuing program. If this bit is off, a wait is required; the task is put into the delay state.

EXAMPLE: Suppose you want to place your task in the delay state (inactive TSI list) until an I/O operation associated with an event control block is completed. You might write:

```
WAIT    EX      0, ECB+2
        B       SOMEPLACE
ECB     DS      0F
        DC      H'0'            SECOND BIT IS COMPLETE BIT
        AWAIT                   AWAIT MUST BE SUBJECT OF EXECUTE
```

The AWAIT will generate the SVC 248.

TWAIT -- Wait for Terminal I/O Interrupt (R)

The TWAIT macro instruction enables you to check for a response to a message you have sent and, pending its arrival, to enter the delay state, which causes any pages for your task to be moved to auxiliary storage.

Name	Operation	Operand
[symbol]	TWAIT	None

EXECUTION: The SVC must be the subject of an execute instruction and must occupy the second halfword of a fullword control block called an event control block (ECB). The supervisor checks the second bit of the halfword preceding the supervisor call and interprets this bit as the event complete bit. If this bit is one, the supervisor returns control and the SVC has the effect of a NOP (no operation). If the bit is zero, the supervisor will set the TWAIT flag in the task's TSI to one and put

the task in the delay state; this will cause time slice end to occur for the task and cause any pages of the task occupying drum storage to be moved to paging disk storage. The task will be removed from the delay state when any task-interrupt -- if the task is enabled -- occurs.

EXAMPLE: Suppose you send a message to some terminal and are waiting a response. The posting routine associated with the IOCAL (see IOCAL) used to transmit the message to the terminal is responsible for setting the event-complete bit of an event control block to one. You have reached a point in your program requiring completion of the IOCAL activity; you do not wish to continue until the IOCAL posting routine has been entered. You might write:

```
        EX      0,TEST+2
        B       IOCOMPLETE
TEST    DS      0F              ALIGN
        DC      H'0'            POSTING FLAGS
        TWAIT
```

The TWAIT will generate an SVC 229.

ADDPG -- Add Virtual Storage Pages (R)

The ADDPG macro instruction enables you to add virtual storage pages to your task.

Name	Operation	Operand
[symbol]	ADDPG	[pgcnt-{value / (1)}][,{startad-{addrx / (0)} ,protcls-code}]

pgcnt
indicates the number of virtual storage pages you want added to your task.

startad
designates the address of the first page you want to add. This address must be a multiple of 4096.

protcls
specifies the protection class you want assigned to each halfpage and may be coded:

- A - both halfpages nonprivileged read/write.
- AB - first halfpage nonprivileged read/write, second halfpage nonprivileged read only.
- AC - first halfpage nonprivileged read, second halfpage privileged.
- BA - first halfpage nonprivileged read only, second halfpage privileged.
- B - both halfpages nonprivileged read only.
- BC - first halfpage nonprivileged read only, second halfpage privileged.
- CA - first halfpage privileged, second halfpage nonprivileged read/write.
- CB - first halfpage privileged, second halfpage nonprivileged read only.
- C - both halfpages privileged.

If you choose to write register notation, you should load the actual page count into register 1 and pack register 0 with the next two parameters.

The start address must be left aligned and the protection class code, selected from the list below, must be right aligned.

Code	Value
A	1
BA	2
CA	3
AB	4
B	5
CB	6
AC	7
BC	8
C	9

For example you might write:

```
            L     0,STARTAD
            SLL   0,8
            IC    0,CODE
             .
             .
             .
STARTAD     DC    XL4'4000'
CODE        DC    BL1'00000001'
```

EXECUTION: New page table entries and, if necessary, segment table entries are constructed, corresponding to the virtual storage address contained in register 0. The number of page table entries to be constructed is determined by the page count contained in register 1. The low-order byte of register 0 is used to determine the setting of the storage keys for all the pages being added.

EXAMPLE: Assume you want to add 100 pages of virtual storage, starting at location $FFFF_1$ with user read-only protection keys. You might code it this way:

```
LABEL   ADDPG   100,START,B
```

where START is a page boundary address.

This would generate,

```
LABEL   DS      0H
        LA      1,100
        LA      0,START
        O       0,=F'5'
        SVC     250
```

ADSPG -- Add Shared Virtual Storage Pages (R)

The ADSPG macro instruction enables you to add shared pages to your task's virtual storage.

Name	Operation	Operand
[symbol]	ADSPG	[startad-{addrx / (1)}][,pgcnt-{value / (0)}]
		[,{sptnbr- value ,protcls-code / (15)}]

startad
specifies the virtual storage address at which you want to start adding shared pages.

pgcnt
indicates the number of shared pages you want to add.

sptnbr
indicates the shared page table to which you are adding the shared pages.

protcls
indicates the protection class you want assigned to each pair of halfpages. The valid codes are the same as those for ADDPG.

CAUTION: Unlike the ADDPG macro instruction, the two parameters to be packed into register 15 each occupy a halfword.

EXECUTION: The number contained in bytes two and three of register 15 is used to determine if pages are to be added to an existing shared page table or if a new shared page table is to be constructed. If bytes two and three of register 15 are zero, or if there are not enough pages remaining in the shared page table indicated by bytes two and three, a new shared page table is constructed. Once the shared page table is selected or constructed, a number of page table entries corresponding to the number contained in bytes zero and one of register 0 is added to it. Storage protection keys are assigned as requested by the code in bytes zero and one of register 15.

The parameters contained in registers 0, 1, and 15 are returned intact to the program issuing the SVC unless a new shared page table had to be constructed; if that were the case, the new shared page table number and new starting virtual storage address replace the corresponding input parameters.

EXAMPLE: Suppose you want to add two shared pages with key B, starting at location RJG; assume the shared page table number is five. You might do it like this:

```
NAME   ADSPG    RJG,2,5,B
```

where RJG is a page boundary address.

This would generate,

```
NAME   DS         0H
       L          15,=F'5*65536'
       LA         0,2
       O          15,=F'5*65536'
       CHDINNRA   RJG,2,(236)        INNER MACRO INSTRUCTION
       L          1,=F'RJG'          GENERATED INNER MACRO
       SVC        236                INSTRUCTION GENERATED
```

DELPG -- Delete Virtual Storage Pages (R)

The DELPG macro instruction enables you to delete pages from your task's virtual storage.

Name	Operation	Operand
[symbol]	DELPG	[startad-{addrx / (0)}][,pgcnt-{value / (1)}]

startad
specifies the address of the first virtual storage page you want deleted.

pgcnt
specifies the number of contiguous virtual storage pages you want deleted.

EXECUTION: The contiguous pages, beginning at the address contained in register 0 and equal to the count in register 1, are deleted from the issuing task's virtual storage. Main storage and paging storage space in use for the released pages are freed for reallocation. If an entire segment is deleted, the auxiliary segment table entry is marked unassigned, the segment table entry is marked not available, and an indicator is set to represent the deleted segment. If the page table entries and external page table entries are not in the first XTSI page, the space they occupied is returned for reallocation. If the auxiliary segment table entry is marked shared, the entry corresponding to the segment in the resident shared page index is deleted.

Note: DELPG can be used for deleting both unshared and shared pages.

EXAMPLE: Suppose you want to delete three pages of virtual storage, starting at ABCXYZ. You might write:

```
TEST   DELPG    ABCXYZ,3
```

This would generate,

```
TEST       DS                0H
           LA                1,3
           CHDINNRA,ABCXYZ,(249)
           LA 0,ABCXYZ              INNER MACRO GENERATED
           SVC249                   INNER MACRO GENERATED
```

CNSEG -- Connect Segment to Shared Page Table (R)

The CNSEG macro instruction enables you to connect a new segment to the shared page table.

Name	Operation	Operand
[symbol]	CNSEG	[{segnbr-value,sptnbr-value / (1)}]

segnbr
specifies the segment that you want connected to the shared page table.

sptnbr
specifies the number of the shared page table to which the segment is to be connected.

If you choose to write register notation, the segment number should occupy the high order halfword of the register 1 and the SPT number should occupy the low order halfword.

EXECUTION: The shared page table number contained in the low-order halfword of register 1 is used to search the task's auxiliary segment table. If an auxiliary segment table entry is already connected to the shared page table, its segment number replaces the high-order halfword of register 1 and control is returned to the SVC-issuing program.

If no auxiliary segment table entry is already connected to the specified shared page table, the segment table entry indicated by the high-order halfword of register 1 is set not available; its auxiliary segment table entry is marked assigned and shared and the shared page table number in register 1 is inserted into the auxiliary segment table entry.

EXAMPLE: Suppose you want to connect shared page table number 3 to segment 12. You might write:

```
RJG   CNSEG    12,3
```

This would generate,

```
RJG   DS       0H
&A3   SETA     3+12*65536
      L        1,=F'&A3'
      SVC      238
```

Shared page table 3 is connected to segment 12 and the high-order halfword of register 1 is left unchanged (as 12) to indicate the actual segment to which the shared page table was connected.

DSSEG -- Disconnect Shared Page Table From Segment (R)

The DSSEG macro instruction enables you to disconnect a shared page table from its segment.

Name	Operation	Operand
[symbol]	DSSEG	[sptnbr -{value / (1)}]

sptnbr
specifies the shared page table you want to disconnect.

EXECUTION: The shared page table number in the low-order halfword of register 1 is used to search the auxiliary segment table. If a matching auxiliary segment table entry is found, it is set not assigned; the segment table entry is set not available.

EXAMPLE: Suppose you want to remove shared page table number 23 from the segment to which it is attached. You might write:

```
ANY   DSSEG    23
```

This would generate,

```
ANY   DS      0H
      LA      1,23
      SVC     237
```

LSCHP -- List Changed Virtual Storage Pages (R)

The LSCHP macro instruction enables you to obtain a listing of virtual storage pages which have been changed.

Name	Operation	Operand
[symbol]	LSCHP	[startad -{addrx / (1)}][,pgcnt-{value / (0)}]

startad
specifies the virtual storage address of the first page you want checked.

pgcnt
specifies the number of consecutive pages you want checked; the maximum number of pages is 16.

EXECUTION: The number of pages specified by register 0 and starting at the page address found in register 1 are checked. The results of this check are stored in register 0. The condition of a given page, page n, is found by checking bits 2n - 2 and 2n - 1 in register 0. The bit pair is interpreted as follows.

Bit Pair	Meaning
00	Page in core and changed
01	Page in core and unchanged
10	Page not in core and changed
11	Page not in core and unchanged

EXAMPLE: Suppose you wish to find out if three pages, beginning at location XYZ, have been changed. You might write:

```
NAME   LSCHP       XYZ,3
```

The macro expansion would produce,

```
NAME   DS          0H
       LA          0,3
       CHDINNRA    XYZ,,(,247)
       DS          0H              INNER MACRO GENERATED
       LS          1,=F'XYZ'       INNER MACRO GENERATED
       SVC         247             INNER MACRO GENERATED
```

CKCLS -- Check Protection Class (R)

The CKCLS macro instruction enables you to check the most restrictive protection class assigned to a group of halfpages.

Name	Operation	Operand
[symbol]	CKCLS	[startad-{addrx / (1)}][,hpgcnt-{value / (0)}]

startad
specifies the virtual storage address of the first halfpage you want to check.

hpgcnt
specifies the number of consecutive halfpages you want to check.

EXECUTION: A code indicating the most restrictive protection class of the pages checked is returned in the low-order byte of register 0. One of these codes will be returned:

Code	Protection Class
0	Page unassigned
1	User read/write (least restrictive)
3	User read only
7	User cannot read or write (most restrictive)

Consecutive halfpages starting at the address contained in register 1 and equal to the halfpage count contained in register 0 are checked.

EXAMPLE: Suppose you want to check the protection class of the five halfpages beginning at RJG. You might write:

```
CKCLS      RJG,5
```

This would generate,

```
DS         0H
LA         0,5
CHDINNRA   RJG,,(,241)
LA         1,RJG           INNER MACRC GENERATED
SVC        241             INNER MACRO GENERATED
```

ADDEV -- Add Device to Task Symbolic Device List (R)

The ADDEV macro instruction enables you to add additional I/O devices to your task. You may have up to 15 devices assigned to your task.

Name	Operation	Operand
[symbol]	ADDEV	[devnbr-{value / (0)}]

devnbr
specifies the symbolic device number of the I/O device you want added to your task's symbolic device list (TSDL).

EXECUTION: The supervisor adds an entry corresponding to the symbolic device number contained in register 0 to the task's symbolic device list. If the device is already in the task's symbolic device list, the count of the number of times the device has been added is increased by one; if this count exceeds 15 an error is indicated. Before returning, the supervisor will set the high-order bit of register 0 to one if the count of the number of times the device has been added exceeds 15.

EXAMPLE: Suppose you want to add symbolic device 17 to your symbolic device list. You might write:

```
ADD   ADDEV   17
```

This would generate,

```
ADD   DS      0H
      LA      0,17
      SVC     234
```

RMDEV -- Remove Device from Task Symbolic Device List (R)

The RMDEV macro instruction enables you to remove an I/O device from your task's list of available devices.

Name	Operation	Operand
[symbol]	RMDEV	[devnbr-{value / (0)}]

devnbr
specifies the symbolic device number of the I/O device you want removed from your task's symbolic device list (TSDL).

EXECUTION: The supervisor reduced the ADDEV count in the task's symbolic device list by one. If the count is reduced to zero, the device entry is removed from the task's symbolic device list.

If the symbolic device number is not found in the task's symbolic device list, the supervisor sets the high-order bit of register 0 to 1.

EXAMPLE: Suppose you want to remove symbolic device 46 from your symbolic device list; assuming no other part of your task had also added device 46, the ADDEV count for device 46 would be one. You might write:

```
GONE   RMDEV   46
```

This would generate,

```
GONE   DS      0H
       LA      0,46
       SVC     233
```

PURGE -- Purge I/O Operations (R)

The PURGE macro instruction allows you to suppress or to remove any or all I/O devices from your task's list of available devices.

Name	Operation	Operand
[symbol]	PURGE	[{action-code, / (0)}[devnbr-value]}][{, task-code / (1)} [,taskid-value]}]

action
specifies the purging action you want and may be any one of these codes:

AR - purge all devices immediately
AS - purge all devices but let the active ones quiesce
AL - purge all I/O requests immediately, leave TSDL alone
AD - remove the TSDL
SR - purge the specified device after it quiesces

devnbr
specifies the symbolic device number of the device you want purged.

task
specifies the combination of tasks from which you want, the device purged and you may write:

AT - the purge is for all tasks
ST - the purge is only for the task specified in the next operand

taskid
is actual ID of the task for which the purge is to be effective.

EXECUTION: The I/C devices to be purged are either suppressed or removed from the task symbolic device list (TSDL) of the task or tasks to which the purge is to apply. If a device is to be allowed to quiesce, its task symbolic device list entry is merely suppressed, if a device is to be purged immediately, its task symbolic device list entry is removed from the task symbolic device list. The TSDLs to be used depend on whether one or all tasks are to have their I/O devices purged.

General register 0 is returned to the calling program with bit zero containing an error flag, if applicable. Depending on the request, this bit can have these interpretations:

For one device, one task -- device not assigned to task

For all devices, one task -- no task symbolic device list exists

For all tasks, one or all devices -- devices not assigned to any task

If a task that does not have the system operator privilege issues SVC 222 for all devices and all tasks, a system error (SYSERR CODE RSC 6101*) is generated.

EXAMPLE: Suppose you wish to purge the I/O device assigned symbolic device number 357 for any tasks that might be using it, but you are willing to wait for the device to quiesce. You might write:

```
NAME   PURGE   SR,357,AT
```

The macro expansion would produce,

```
NAME   DS      0H
       L       0,=CL4'SS00'*
       O       0,=F'357'
       L       1,=CL4'AT00'*
       SVC     222
```

*The zero in this character string is only for illustration; a hexadecimal zero doesn't have an assigned graphic and must be punched as 12-0-9-8-1.

RESET -- Reset Device Suppression Flag (R)

The RESET macro instruction allows you to cancel a previous PURGE by resetting a device's suppression flag in the TSDL.

Name	Operation	Operand
[symbol]	RESET	[devnbr- {value / 'ALL' / (0)}]

devnbr
specifies the symbolic device address of the device whose flag you wish reset. 'ALL' indicates all device suppression flags are to be reset.

EXECUTION: The supervisor clears the device suppression flag in the task symbolic device list for the symbolic device number contained in register 0. This has the effect of cancelling a previous PURGE for the symbolic device.

Before control is returned, an error flag may be set in the high-order bits of register 0; if this bit is set to one, it means that the symbolic device is not contained in the task's symbolic device list.

EXAMPLE: Suppose you want to allow I/O operation to proceed on symbolic device 25. You might write:

```
GO   RESET   25
```

This would generate,

```
GO   DS      0H
     LA      0,25
     SVC     221
```

SPATH -- Set I/O Device Path (R)

The SPATH macro instruction enables you to set flags indicating units along a path are partitioned or malfunctioning.

Name	Operation	Operand
[symbol]	SPATH	[flag-{code / (0)}]
		[,{comp-integer,devad-hexinteger (1)}]

flgst
specifies the flag and the setting you desire and must be one of these codes:

POF - set unit's partitioned flag off (0)
PON - set unit's partitioned flag on (1)
SOF - set unit's malfunction flag off (0)
SON - set unit's malfunction flag on (1)

comp
indicates the component (channel, control unit, device) you want set and may be coded:

1 - I/O device only
2 - Control unit only
3 - Control unit and I/O device
4 - Channel only
5 - Channel and I/O device

6 - Channel and control unit
7 - Channel, control unit, and I/O device

devad
specifies the actual device address of the path you want set and must be a hexadecimal value less than X'2000'.

EXECUTION: The appropriate flag in the pathfinder's tables is set on or off according to the codes contained in registers 0 and 1 for the device, control unit, and/or channel represented by the actual device address contained in bits 19-31 of register 1.

EXAMPLE: Suppose you wish to partition device A01 and its control unit. You might write,

```
NAME   SPATH    PON,3,X'A01'
```

This would produce,

```
NAME   L        0,=X'40020000'
       L        1,=X'00006A01'
       SVC      211
```

SETAE -- Set Asynchronous Entry (F)

The SETAE macro instruction permits you to process your own asynchronous interruptions by moving the device capable of producing such an interruption to another task.

Name	Operation	Operand
[symbol]	SETAE	[devad{value / (1)}][,task-{value / (0)}]

device
specifies a symbolic device address whose value cannot be greater than $2^{14}-1$.

task
specifies a task identification number whose value must be less than $2^{16}-1$. If task is null, the entry is restored to its neutral state; otherwise, the entry is updated to point to the task specified.

EXECUTION: The entry in the asynchronous device group table (CHAADT) corresponding to the symbolic device address is set to point to the task status index of the task corresponding to the task ID supplied in register 0. If task is zero, the symbolic device is marked unassigned. An entry is also placed in the TSDL for the new task.

EXAMPLE: Suppose you want attention interrupts received from device 124 to be processed by the task, with task-identification 233. You might write:

```
ESTAB   SETAE   124,233
```

This would generate,

```
ESTAB   DS      0H
        LA      1,124
```

```
LA      0,233
SVC     210
```

IOCAL -- I/O Call (R)

The IOCAL macro instruction provides for the initiation and execution of an I/O operation.

Name	Operation	Operand
[symbol]	IOCAL	None

EXECUTION: An IOCAL macro instruction must always be the subject of an execute. The supervisor call is assumed to occupy the first halfword of a variable-length parameter list called an I/O request control block (IORCB). The IORCB supplies the information necessary for the supervisor to perform the requested I/O operation.

An IORCB consists of four parts: A fixed part of 10 doublewords; An optional I/O data buffer which cannot exceed 225 double words; An optional page list which cannot exceed eight doublewords; A channel command word (CCW) list. The entire IORCB cannot exceed 240 doublewords and must be wholly contained within a single page.

When an IOCAL is executed, the supervisor -- after some error checking -- obtains main storage space for the IORCB and copies it into that space. Based on the options selected by the user and indicated in the IORCB, the supervisor obtains a path to the requested I/O device, translates the virtual CCW addresses to real storage addresses, brings any required buffer pages into main storage, and starts the I/O operation. After the I/O operation is complete, the supervisor releases the device path, allows the data buffer pages to be paged out of storage -- if necessary -- and queues a pending task-I/O interrupt for the task associated with the IORCB.

When (and if) the pending interrupt is accepted by the task, the supervisor copies the IORCB into the task's interrupt storage area (ISA). After the IORCB has been copied, the main storage space it required is released for reallocation and the IOCAL operation is completed. A task may have more than one IOCAL in operation at one time and may operate asynchronously with an active IOCAL.

Figure 8 shows the format of the fixed area of the IORCB as it is viewed prior to issuing an IOCAL. Every IOCAL must have a 20 word fixed area regardless of the fields used. You may use space within the IORCB itself as a data buffer; you can do this if the data does not exceed 225 doublewords (or whatever space is left in the IORCB after the other items you need are included). You must include an IORCB data buffer if you are using a direct access device and if you wish record zero to be read into the IORCB by the supervisor if a unit check occurs. In this case, the IORCB data buffer (the first 56 bytes) are used to hold record zero. You may also use data buffers outside the IORCB; if you do this, you must include a page list. The page list contains one doubleword entry for each virtual storage buffer page; you may not have more than eight page-list entries. Figure 9 shows the format of a page-list entry.

Each page-list entry is associated with a channel command word (CCW) list entry; the CCW entry tells what operation is to be performed with the data buffer. A CCW entry does not need to point to a page-list entry; a page-list pointer of zero is assumed to mean the IORCB buffer is to be used. You can use any number of CCW entries as long as the size limits of the IORCB are not exceeded. Figure 10 illustrates the format of a CCW list entry. You always have at least one CCW entry, since the CCW represents the work you want the supervisor to do.

If the software command chain flag is one (see Figure 11), the supervisor will continue to reissue start-I/O instructions at the current point in the CCW list when a device-end interrupt is received. This has the effect of making the CCW list appear chained, even though the path to the I/O device may be free for certain periods during the operation. The most common use of software command chaining is to chain a seek to its read or write command.

If the IORCB chain flag is 1 (see Figure 8), the supervisor will change the last CCW entry to a transfer in channel (TIC) command if another (second) IOCAL for the same device is received before the final channel-end/device-end interrupt for the first IORCB is received by the supervisor. This TIC command will link the CCW lists of the two IORCBs. The supervisor will also set the program controlled interrupt (PCI) bit on, in the start CCW of the second IORCB. The receipt of this PCI signals the completion of activity for the first IORCB; the supervisor then enqueues a pending task-I/O interrupt for that IORCB.

The IORCB received by the task monitor as a result of the task-I/O interrupt has been changed by the supervisor; it is not identical to the IORCB originally received by the supervisor. Figure 11 shows the fields in the fixed area of the IORCB that may be set by the supervisor. Figure 12 shows the changes to the CCW list entry.

As part of its interrupt handling logic, the task monitor transfers control to the posting routine pointed to by the IORCB it receives as a by-product of the task-I/O interrupt. This posting routine informs the program originally issuing the IOCAL that the I/O operation has been completed.

0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
IOCAL (SVC 231)	USED BY ACCESS METHODS -- NOT SET OR INTERROGATED BY IOCAL

length of IORCB in 64-byte units (blocks)	length of page list in doublewords	relative origin of page list in doublewords	storage protection key (1 or 2)	start I/O failure count. (note 1).	length of CCW list in doublewords	relative origin of CCW list in doublewords	relative origin of starting CCW in doublewords
length of IORCB data buffer in doublewords	relative origin of IORCB data buffer in doublewords	actual I/O address to be used for this operation (note 2).		not used		system symbolic device address must be given if actual path not supplied	

USED BY ACCESS METHODS -- NOT SET OR INTERROGATED BY IOCAL	
V-type address constant of posting routine to be transferred to by the task monitor when the task-I/O interrupt associated with this IORCB occurs	R-type address constant of posting routine (see preceeding word)
USED BY ACCESS METHODS -- MAY BE SET BY IOCAL	
USED BY ACCESS METHODS -- NOT SET OR INTERROGATED BY IOCAL	
MAY BE SET BY IOCAL	USED BY ACCESS METHODS -- NOT SET OR CHECKED BY IOCAL

NOT USED	users' options							SET BY IOCAL.	options							
	S	I	R	C	H	U		P							G	
	(note 3)														note 4	

USED BY IOCAL FOR PERFORMING SENSE OPERATION
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7

Note 1. If flag R (note 3) is one, the start I/O instruction is reissued the number of times specified by this count, or until the start I/O instruction is successfully initiated.

Note 2. If flag S (note 3) is one, the I/O address contained in this halfword is used and the symbolic device address is ignored.

Note 3. S=specific I/O address; I=ignore device malfunctioning indicator in pathfinder; R=if start I/O not accepted because device is busy, reissue start I/O (see Note 1.); C=Software command chain; H=issue halt I/O on device before start I/O; U=if unit check occurs, read direct access device record zero into IORCB data buffer; P=treat PCI as channel end/device end.

Note 4. IORCB chaining flag; if another IOCAL is received for this device while the channel program for this IORCB is running, the last CCW of this list is TICed to the first CCW of the other IORCB.

Figure 8. Format of Fixed Area of Input/Output Request Control Block as Set Before IOCAL

0 12 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0	1	2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
high-order 20 bits of virtual storage address; the segment and page number of virtual buffer page	flages			UNUSED	set to actual main storage location used for this page -- before IORCB is returned to task monitor at task-I/O interrupt time
		A	(note 1)		

Note 1. A = (paging storage) copy of this page does not need to be used; use any core page and release paged copy

Figure 9. Organization of a Page List Entry

0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3	4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0	1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
CCW Operation code as in OS/360	position of page list entry 1,2...8 (note 1)	flags I (note 2)	displacement within page buffer or from start of IORCB buffer or from start of CCW list if T'C	CCW flags as in S/360 hardware	0s	BYTE COUNT

Note 1. If this field is 0, the IORCB data buffer is assumed

Note 2. I = do not relocate CCW addresses

Figure 10. Channel Command Word List Entry Before IOCAL is Issued

0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7						
UNCHANGED						
UNCHANGED						
UNCHANGED		set to actual I/O address used for this operation, or left unchanged if user supplied		UNCHANGED		
UNCHANGED						
UNCHANGED						
UNCHANGED			condition codes I I I C C S S Note 1	real main storage address used for IORCB data buffer. If no IORCB buffer used set to real address of IORCB itself		
UNCHANGED						
sense c codes I I C C S S (Note 2)	sense CSW status placed here if both requested operation and sense operation fail	sense failed fg 0 (Note 3)	halt I/O retry count (Note 4)	UNCHANGED		
UNCHANGED				flags S P I H R N W T x x C (Note 5)		NOT USED
CCW FOR PERFORMING SENSE OPERATION ON REQUESTED I/O DEVICE						
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7						

Note 1. If operation came to abnormal end, these condition codes are stored: I=test I/O condition code; C=test channel condition code; S=start I/O or halt I/O condition code.

Note 2. If sense operation failed, these condition codes are stored for I/O instructions used to attempt sense (see note 1).

Note 3. 0=a device other than the one requested has monopolized the control unit; sense data applies to that device.

Note 4. If user requested both a retry of start I/O and halt I/Os before each start I/O, this field is set equal to the user supplied start I/O count.

Note 5. S=CCW specification error; P=no path exists to requested device; I=start I/O failed; H=halt I/O failed; R=read record 0 (on direct access error) failed; N=sense failed; W=CCW addresses are relocated (changed to real addr) T=IORCB aborted because previous (pending) IORCB for same task had abnormal end; x=internal flag for IOCAL; C=interrupt code applies to device other than one requested monopolizing control unit.

Figure 11. Fixed Area of I/O Request Control Block as Set by IOCAL

0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
UNCHANGED	MAIN STORAGE ADDRESS USED FOR OPERATION	UNCHANGED

Figure 12. Channel Command Word List Entry After Task I/O Interrupt Occurs Occurs

EXAMPLE: Whenever you use an IOCAL, you should be sure to refer to the current version of the IORCB. The format of the IORCB is described by a dummy section in the system copy/macro library. You can get a copy by

assembling a program with this in it:

```
        COPY    CHAIOR
```

Suppose you want to read 120 bytes from symbolic device 85. You might write,

```
BGN     EX      0,TEST
        B       AWAY
TEST    IOCAL
        DC      3H                      WE DCN'T USE THIS
        DC      C'(TEST-IOREND)/64'
        DC      CL2'0'                  NO PAGE LIST
        DC      C'2'                    WE'RE PRIVILEGED
        DC      C'0'                    DEVICE SHOULDN'T BE BUSY
        DC      C'1'                    ONLY CNE CCW
        DC      C'(CCW-TEST)/8'         RELATIVE ORIGIN OF CCW LIST
        DC      C'0'                    START CCW IS FIRST
        DC      C'120/8'                IORCB BUFFER LENGTH
        DC      C'(BUF-TEST)/8'         RELATIVE ORIGIN OF IORCB
                                         BUFFER
        DC      H'0'                    NCT GIVING ACTUAL ADDRESS
        DC      H'0'                    NOT USED
        DC      H'85'                   SYMBOLIC DEVICE ADDRESS
        DC      D'0'                    WE WON'T USE THIS
        DC      A(POST)                 THE ADDRESS OF CUR
        DC      R(POST)                 POSTING PROGRAM
        DC      7F'0'                   WE WON'T USE THIS
        DC      3F'0'                   WE'LL ELECT "NO" CN THE
                                         CPTICNS
ENDFIX  DS      0D                      END CF FIXED AREA OF IORCB
BUF     DC      15D'0'                  120 BYTES OF BUFFER
CCW     CCW     READ,0,X'20',120
IOREND  DS      0D                      END OF IORCB
READ    EQU     194                     COMMAND FOR 2540 BCD READ
```

Notice that the CCW page-list entry and displacement fields are both zero; this causes the IORCB buffer to be used and the information to be read into the first byte of the buffer. After the operation is complete, the supervisor will cause a task-I/C interrupt which will store the IORCB in the interrupt storage area. Our posting program (POST) can, if it wishes, move the data out of ICRCB buffer at that time.

The macro instruction above would generate:

```
        CNOP    0,8     MAKE CERTAIN THAT THE EXECUTABLE
                        INSTRUCTICNS ARE DCUBLE-WCRD
                        ALIGNED.
TEST    SVC     231
```

PGOUT -- Write Virtual Storage Pages to External Storage

The PGOUT macro instruction enables you to write from one to eight virtual storage pages to one or more external storage devices.

Name	Operation	Operand
[symbol]	PGOUT	None

EXECUTION: The PGOUT macro instruction must be the subject of an execute instruction and must occupy the high-order halfword of the first word of a parameter list called an I/O paging control block (IOPCB). The IOPCB consists of a header and a number of external storage list entries (see Figure 13).

The supervisor reads into main storage any pages in the list that aren't already in main storage; when all pages are in, the supervisor writes them out at the external storage locations supplied in the external storage list. From one to eight consecutive virtual storage pages may be transmitted; the destination external storage locations need not be consecutive and may be on different devices.

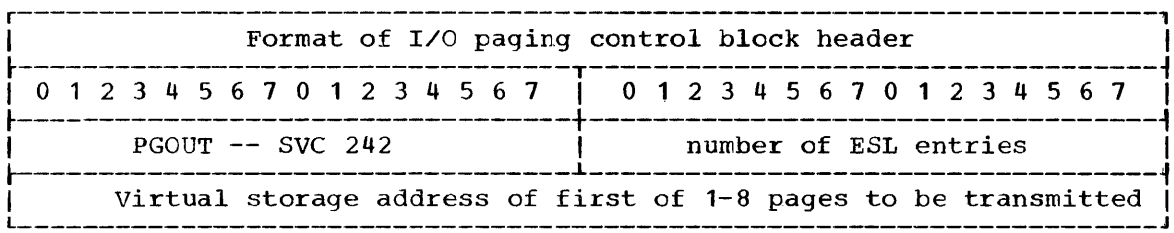
Before returning, the supervisor puts information in register zero to describe the action with each page in the external storage list. Four bits of register 0 are assigned to each page; bits 0-3 for the first page, bits 4-7 for the second, etc. The four bits are interpreted as follows:

Value	Meaning
0000	No error - page transmitted
0011	Virtual storage page not assigned to task
0100	Request for zero pages
0101	Symbolic device not assigned to task
0110	Page in -- bad device -- volume is movable
0111	Page in -- bad device -- volume is fixed
1000	Page in -- medium failure
1001	Page out -- bad device -- volume is movable
1010	Page out -- bad device -- volume is fixed
1011	Page out -- medium failure

EXAMPLE: Suppose you want to write virtual storage page RSLTS on the 127th page position of symbolic device 34. You might write,

```
OUT     EX      0,MOVE
        B       SOMEPLACE
MOVE    PGOUT
        DC      H'1'          1 PAGE TO BE TRANSMITTED
        DC      A(RSLTS)      EXTERNAL STORAGE LIST ENTRY
        DC      H'34'         SYMBOLIC DEVICE NUMBER
        DC      H'127'        RELATIVE PAGE NUMBER
```

The macro instruction above would generate an SVC 242 to make certain the executable instructions are full word aligned.



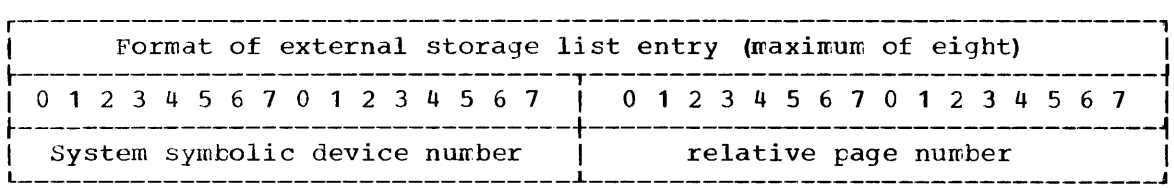


Figure 13. I/C Paging Control Block

SETXP -- Set External Page Table Entries (R)

The SETXP macro instruction enables you to flag external page table entries that you are currently setting up as "unprocessed by dynamic loader." The first reference to the page or pages indicated in those entries will then cause control to be given to the dynamic loader.

Name	Operation	Operand
[symbol]	SETXP	None

EXECUTION: The first bit of the halfword immediately following the SVC is interpreted as a bit string flag. If this bit is one, each unprocessed-by-loader bit for each entry modified in the external page table must be on. The page count maximum is 1022. The low-order 10 bits of the halfword following the SVC are interpreted as a page count. The first fullword following the SVC contains the virtual storage address at which the external page table entries are to be set. After this word -- and depending on the page count -- are a number of words; each word contains an external page table entry that is to be set. If a bit in the string is one, the corresponding page table entry is marked "unprocessed by loader." If a bit is zero, the unprocessed-by-loader bit is not set for the external page table entry. If the unprocessed-by-loader flag is set for a page, the first reference to that page by a program will cause control to be given to the dynamic loader via a task-program interrupt type 16 or 17.

The external page table entries supplied in the parameter list are set as indicated. The unprocessed-by-loader bit is set for each page whose bit string flag is a one.

EXAMPLE: Suppose you want to set external page table entries for three pages beginning at location NEW. You might write,

```
SAMPL   EX      SET
        B       SOMEPLACE
SET     DS      0F          SVC MUST BE ON FULL WORD BOUNDARY
        SETXP
        DC      H'3'        NO BIT STRING, THREE PAGES
        DC      A(NEW)      ADD EXTERNAL PAGE TABLE ENT AT NEW
        DC      F'1251'     EXTERNAL PAGE ADDRESS
        DC      F'356'      EXTERNAL PAGE ADDRESS
        DC      F'1234'     EXTERNAL PAGE ADDRESS
```

The SETXP macro instruction generates an SVC 244.

MOVXP -- Move Page Table Entries (R)

The MOVXP macro instruction enables you to move page table and external page table entries from one table to another or from one part of a table to another.

Name	Operation	Operand
[symbol]	MOVXP	[startad-{addrx | (0)}][,toad-{addrx | (1)}] [,pgcnt-{value | (15)}]

startad
specifies the address of the first page table or external page table entry you want moved and must be a multiple of 4096.

toad
specifies the address to which you want the first entry moved and must be a multiple of 4096.

pgcnt
specifies the number of consecutive entries you want moved.

EXECUTION: The page table and external page table entries beginning at the page address contained in register zero are moved to the page address contained in register 1; the number of entries to be moved is contained in register 15. Each "from" page table entry is marked assigned but unavailable; each from external page table entry is cleared to zero.

EXAMPLE: Suppose you want to move 300 pages located at IN to an area beginning at OUT; both IN and CUT must be page boundary addresses. You might write,

```
       MOVE   MOVXP    IN,OUT,300
```

This would generate,

```
       MOVE   DS       0H
              LA       15,300
              CHDINNRA OUT,IN,(,245)
              LA       1,OUT          INNER MACRO GENERATED
              LA       0,IN           INNER MACRO GENERATED
              SVC      245            INNER MACRO GENERATED
```

LVPSW -- Load Virtual Program Status Word (R)

The LVPSW macro instruction enables you to alter the flow of your program by changing its PSW in virtual storage.

Name	Operation	Operand
[symbol]	LVSPW	[pswad-{addrx / (1)}]

pswad
specifies the virtual storage address at which the new VPSW is presently stored.

EXECUTION: The virtual program status word whose address is in register one becomes the current virtual program status word. The previous contents of the virtual program status word are lost.

EXAMPLE: Assume location NEWVPSW contains a new virtual program status word that is to be loaded. The macro instruction

```
       NAME   LVPSW   NEWVPSW
```

causes this to be generated,

```
              CNOP    4,8
       NAME   TM      0(0),0
              TM      NEWVPSW,0
              SVC     254
```

VSEND -- Send Message to Another Task (R)

The VSEND macro instruction enables you to send information to another task.

Name	Operation	Operand
[symbol]	VSEND	None

EXECUTION: The SVC 240 resulting from a VSEND macro instruction must be imbedded in a message control block (MCB) and be the subject of an execute instruction. The format of a message control block can be found in System Control Blocks PLM.

The receiving task is alerted to the message by a task-external interrupt. When the external interrupt is accepted, the supervisor moves the MCB into the recipient task's ISA. No more than 1904 bytes can be transmitted. If the receiving task's intertask message flag (TSIMB) is one, it does not wish to receive messages. If the sending task's identification indicates that it is a system operator or the batch monitor, the receiving task gets the message (i.e., the pending task-external interrupt) in any event. If the sender is neither the batch monitor nor a system operator and the recipient's intertask message flag is one, register 1 is set to four, telling the sender that his message was not accepted. If the recipient task cannot be found, register 1 is set to zero. If the message is sent, register 1 is set to eight. If and when the message is accepted by the recipient task, and if the reply flag in the senders MCB is one, the complete bit in the event control block pointed to by the sender's MCB will be set to one.

EXAMPLE: Suppose you want to send the message, "This is a test." to a task whose task identification is 1273. You might write,

```
ANY   EX      0,MCB+4
      B       UPUPAWAY
MCB   DS      0D          DOUBLE WORD BOUNDARY
      DC      C'2'        NUMBER OF DOUBLE WORDS OF MESSAGE TEXT
      DC      CL3'0'      MESSAGE CODE FIELD
      VSEND
      DC      H'0'
      DC      H'1234'     OUR TASK ID
      DC      H'1273'     TASK ID OF RECIPIENT
      DC      A(ECB)      ADDR OF EVENT CONTROL BLOCK
      DC      CL15'THIS IS A TEST.'
```

The VSEND would generate an SVC 240. The remaining information composes a message control block (MCB).

ERROR -- Indicate Supervisor Detected Error (nonstandard)

The ERROR macro instruction provides the means by which the resident supervisor reports the occurrence of a major or minor software error or a hardware failure.

Name	Operation	Operand
[symbol]	ERROR	errtype-integer, dump-integer, module-integer, idno-integer [,tskint-{integer / 0}]

errtype
specifies the type of error which has occurred. The codes are given in Table 5.

dump
specifies the content of the dump you want supplied. The codes are given in Table 6.

Table 5. System Error Codes

Type of Error	Code
Minor software error	1
Major software error	2
Hardware failure	3
Hardware failure -- generate task-program interrupt	7
Minor software error -- generate task-program interrupt	9

Table 6. Dump Option Codes for System Error Processor

Dump to be Taken	Code
Basic output (error message, address of TSI, address of GQE, address of DCB, general-purpose and floating-point registers, storage locations 0-127)	00
Basic output and all storage from 4096	02
Basic output and all storage as specified by storage list pointed to by register 1	04
Basic output and TSI XTSI Task interrupt log ISA Of current processing unit task (PSATPT)	10
Basic output and all virtual storage of processing unit's current task (PSATPT)	20

module
designates the supervisor module issuing the ERROR macro instruction (see Table 7).

idno
designates a specific ERROR call in modules which issue multiple calls and must be in the range 0-99

taskint
task interrupt code to be used for error types 7 and 9 and must be in the range of 0-9999

Note: Both LVPSW and ERROR use the same SVC code (254); an SVC 254 occurring in the problem state is considered LVPSW; an SVC 254 occurring in the supervisor state is considered ERROR.

EXECUTION: ERROR is the only SVC that may be issued by the resident supervisor. The processing unit receiving the ERROR SVC will stop all other processing units in the system. The information to be dumped is converted to heaxadecimal format and transmitted to the system error output device.

Table 7. Resident Supervisor Module Codes

Code	Module Name
01	Dispatcher
02	Queue scanner and Enqueue-dequeue
03	Timer interrupt queue processor
04	Page turning
05	Core allocation and release
06	Program interrupt queue processor
07	Task initiation
08	XTSI overflow
09	Interrupt stacker (program, SVC)
10	Page posting
11	Activate and deactivate TSI
12	Supervisor call queue processor
13	Auxiliary storage allocation
14	Interrupt stacker (external, I/C, machine check)
15	Inter-CPU communication
16	TSS/360 recording
17	System inventory routine
18	Reconfiguration routine
19	Locate page
20	Real core diagnostic and error recovery
21	Data recording screen
22	Start recording SVC processor
23	Buffer packing for data recording
24-50	Unassigned
51	I/O call routine
52	I/O device queue processor
53	Pathfinding
54	Channel interrupt queue processor
55	Page drum interrupt queue processor
56	Page drum queue processor
57	Page direct access request and interrupt queue processor
58	Page out service routine
59	Task interrupt control
60	Device allocation and release
61	Purge
62	External page location address translator
63	Queue GQE on TSI
64	Dequeue I/O request
65	Start I/O
66	Paging I/O error recovery control
67	Task communication control
68	Suppress auxiliary allocation routine
69	Paging path analysis
70	Reinitialize operator task
71	Alternate path retry
72	Standard area retry
73	Scan on task ID routine
74	Same path retry
75	Standard area retry analysis
76	Rebuild DAIB/SYSDIC and restart I/O
77	External machine check interrupt processor
78	Locate outstanding CCU I/O operation
79	Set asynchronous entry
80	Data recording I/O
81	Data recording error recovery
82-99	Unassigned

If the error type is major (2, 3, or 7), the SVC 254 routine transfers control to the recovery nucleus; if the error type is minor (1 or 9), or if the recovery nucleus returns control to the SVC 254

routine, all other processing units in the system are restarted. If a task-interrupt has not been requested, control is returned to the instruction following the ERROR parameter list. Otherwise, a GQE is constructed and enqueued on the TSI pointed to by the prefixed storage area field PSATPT, subsequently causing a task-program interrupt.

The ERROR code transmitted as part of the basic output is of the form:

mmnn

Where mm is the two-digit module code and nn uniquely identifies multiple SVC 254s within the same module.

EXAMPLE: Suppose you detect a major error -- quantity A was neither less than, equal to, nor greater than quantity B. You might write,

```
BLAST   ERROR   3,01,23,02
```

This would generate,

```
BLAST   SVC     254
        DC      X'83',X'01'
        DC      X'23',X'02',XL2'00'
```

SYSER -- Indicate Nonresident-Program Detected Error (nonresident)

The SYSER macro instruction is the means by which a nonresident task reports errors it has detected.

Name	Operation	Operand
[symbol]	SYSER	errtype-integer, dump-integer,opt_1-integer, opt_2-integer,opt_3-integer,idno-integer

errtype
: specifies the type of error the task has detected. The codes are given in Table 5.

dump
: specifies the dump output you wish to receive. The codes are given in Table 6.

opt_1
: specifies the first of three unique identifiers, in the range 1-83; see Appendix E for values of this parameter for specific modules.

opt_2
: specifies the second of three unique identifiers, in the range 1-99; see Appendix E for values of this parameter for specific modules.

opt_3
: specifies the third of three unique identifiers, in the range 1-999; see Appendix E for values of this parameter for specific modules.

idno
: is a number from 1 to 99 which is used to uniquely identify one of several calls in a module.

EXECUTION: The processing unit receiving the SYSER SVC will stop all other processing units in the system. The information to be dumped is converted to hexadecimal format and transmitted to the system error output device.

If the error type is major (2, 3, or 7), the SVC 228 routine transfers control to the recovery nucleus; if the error type is minor (1 or 9), or if the recovery nucleus returns control to the SVC 228 routine, all other processing units in the system are restarted. If a task-interrupt has not been requested, control is returned to the instruction following the SYSER parameter list. Otherwise, a GQE is constructed and enqueued on the TSI pointed to by the prefixed storage area field PSATPT, subsequently causing a task-program interrupt.

The SYSER code transmitted as part of the basic output is of the form:

vvccsssnn

where vvccsss identify the opt_1 (vv), opt_2 (cc), and opt_3 (sss) codes, respectively, for the module issuing the SVC 228 and nn uniquely identifies multiple SVC 228s within a single module.

EXAMPLE: Suppose your task detects a minor software error and you want to get just the basic SYSER output. You might write,

```
BUG   SYSER    1,00,2,0,23,01
```

This will produce

```
      CNOP     0,8
BUG   SVC      228
      DC       X'81',X'00'
      DC       AL.1(1),AL.23(*100000+0*1000+23),X'01'
```

PRIVILEGED PROGRAM NAMING CONVENTIONS

As discussed in the section about resident supervisor naming conventions, all TSS/360 program module names begin with the letter C. All privileged program module names have the form:

CZxxx

where the characters xxx are used to uniquely identify all program module names beginning with CZ. All control section names and entry point names of privileged virtual programs add a character to the end of the module name to form a unique entry point or control section name. This is analogous to the way entry point and control section names are formed for resident supervisor modules. For example, an entry point of privileged module CZCJT might be CZCJTH.

Dummy sections for system control blocks are used by privileged virtual programs in the same way that they are used by resident supervisor programs. All system dummy section names begin with CHA; the location of the first byte of data described by a system dummy section is named by a label beginning with CHB. CHAXYZ is a dummy section describing data located at virtual storage address equivalent to CHBXYZ.

The dynamic loader treats all external names (ENTRY, EXTRN, V-type adcons, etc.) beginning with the characters SYS as system names. A control section without the attribute PRVLGD cannot define system names (externally). (See Table 1 for the effect of authority code in dynamic loader processsing.)

WRITING PRIVILEGED SYSTEM PROGRAMS

Virtual system programs are divided into two classes: programs that make up initial virtual storage and programs that are dynamically loaded. Broadly speaking, initial virtual storage (IVS) is composed of all those system programs (both privileged and nonprivileged) necessary to dynamically load a program. An attempt to dynamically load a program will not require or depend upon the prior dynamic loading of some other program. This is another way of saying that the dynamic loader is not a recursive program; it doesn't call itself. Privileged programs that are not part of IVS are brought into virtual storage, as required, by the dynamic loader and the miscellanous programs it uses for assistance.

In writing a system program, you must know whether it will be dynamically loaded or be part of IVS. Programs that are part of IVS must not attempt to dynamically load other programs that are part of IVS. If your program is not to be part of IVS, you needn't worry about whether the programs you call are or are not part of IVS. For example, if programs A and B are both in IVS, program A might call program B like this,

```
      NAME   CALL    B,DATA,I   IMPLICIT CALL
```

If program A were not in IVS, either this could have been written,

```
      NAME   CALL    B,DATA,E   EXPLICIT CALL
```

or this could have been written

```
      NAME   CALL    B,DATA,I   IMPLICIT CALL
```

regardless of where program B is.

The use of the E option in the CALL requires action by the dynamic loader; this is not allowed for programs that are part of IVS -- unless the program being called is outside of IVS.

Almost all TSS/360 programs can be shared by several users. When a program is shareable, or public, it must be put together in a special way. Each public program is thought of as consisting of two parts.

One part is made up of all the instructions and data in the program that never change because of relocation in virtual storage by the dynamic loader or because of execution by a processing unit (variables). This part of a public program is pure procedure; it is literally constant -- it never changes under any circumstances.

The second part of a public program consists of those parts of the program that may change because of relocation or execution -- the program's adcons and variables.

There is no requirement that each and every program have parts that change and parts that don't change. Indeed, some programs don't contain a single byte that ever changes; these programs keep all their variables in the general registers.

The parts of a public program that may change, the adcons and variables, are collected in a prototype control section (PSECT). All other control sections of a public program should be given the attribute, READONLY, since they can never be modified. As an exception to this, there are a few tables, such as the symbolic device allocation table (SDAT), that are protected with lock bytes and are shared nonread-only control sections. The devision of a public program into prototype control sections and read-only control sections allows a number of different tasks to share the same program without destroying

one another's results. This is accomplished by giving each task that is sharing the public program its own private copy of the prototype control section, while allowing each task to share a single copy of the public program's read-only control sections. In this way, each task has a private copy of those parts of the public program that may change, thus preventing tasks from destroying one another's variables and allowing each task to have its own adcon values.

You should take care not to confuse intertask program reenterability with intratask reenterability. The use of prototype and read-only control sections permits programs to be shared among many different tasks; this is intertask reenterability. The use of a prototype control section for storing variables does not automatically guarantee that, within a single task, a program can be reentered. All problem programs are freely interruptable by any real (not virtual) interrupt. When such an interrupt occurs, before control is returned to the interrupted program in virtual storage, the resident supervisor checks to see if there are any pending task-interrupts. If there are pending task-interrupts, the corresponding task-mask bit in the virtual program status word is set to 1 (enabling task-interrupts) and the ISA lock byte is zero; control is returned, not to the interrupted program, but to the task monitor. The task monitor, after some housekeeping, transfers control to the appropriate task-interrupt-handling routine. In some instances, the interrupt handler may have to use the interrupted program as a subroutine; GET, for example. When this happens, the interrupted program is being reentered. It is thus task-interrupt sensitive and it must be constructed to allow for this sensitivity. The prototype control section is no help in permitting intratask program reenterability, since, within this single task, there is only one prototype control section for each public program and only one copy of variables and adcons can be preserved in it.

Although address constants change as a result of program relocation and are placed in a public program's prototype control section, and may assume different values from task to task, they are not considered variables <u>within a task</u>. Once supplied by the dynamic loader (or by startup for IVS), an address constant within a given prototype control section will not change. (The equivalent address constant in other copies of the same prototype control section will, in all probability, be different. In other words, if the only thing in a prototype control section were a set of address constants, then such a PSECT would be read-only since it would never change after dynamic loading.

Within a single task, we are concerned about those parts of a program (public or otherwise) that change as a result of that program's execution by a processing unit. If a program that stores variables in fixed areas of virtual storage can be called by a number of other programs, it must protect itself against task-interrupts. If a program must be interruptable (by <u>task</u>-interrupts), it must use GETMAIN (or something equivalent) to dynamically allocate virtual storage and thus prevent the accidental destruction of variables. GET and PUT are examples of programs that can be in use by one program, interrupted, and reentered for use by another program within the same task.

If you wish to disable task-interrupts during some processing, you can use the macro instruction ITI (inhibit task interrupts); to enable task-interrupts, the macro instruction PTI (permit task interrupts) may be used (see Appendix A). For example,

```
        COPY    CHAISA
LOCK    ITI             DISABLE TASK INTERRUPTS
          .             MISCELLANEOUS INTERRUPT-SENSITIVE CODING
          .
          .
        PTI             ENABLE TASK INTERRUPTS
```

shows how task-interrupt might be disabled and restored in a program. The COPY statement must be included, since it is needed to define a field (ISALCK) used by the macro expansions of ITI and PTI.

Excluding dummy control sections, which are not true control sections (see discussion of dummy usage), you may have two kinds of control sections in your program: prototype (PSECT) and nonprototype (CSECT). From the standpoint of the dynamic loader, there is very little difference between a PSECT without qualifying attributes and a CSECT without qualifying attributes. Throughout TSS/360, however, PSECTs are used in public programs to contain address constants and variables; you should think of prototype control sections as the private part of shared program modules.

Be careful not to confuse the attributes PRIVLGD and SYSTEM. PRIVLGD automatically includes SYSTEM; every privileged program is a system program as far as the dynamic loader is concerned. SYSTEM does nct automatically include PRVLGD, however; every system program is not automatically privileged.

You might code a sample privileged program like this,

```
            TITLE    'SAMPLE PRIVILEGED PROGRAM'
            DCLASS   PRIVILEGED                    THIS ALLOWS PRIV
                                                   MACRC EXPANSICNS
            COPY     CHAISA                        GET FCRMAT OF
                                                   ISA
CZABP       PSECT    PRVLGD                        PUT ALL THE
                                                   ADCONS AND
                                                   VARIABLES HERE
            EXTRN    CHBXYZ                        LOCATION CF
                                                   TABLE XYZ'S DATA
CACABC      CSECT    READONLY,PUELIC,PRVLGD        PURE PROCEDURE
             .                                     SECTION ANYTHING
             .                                     HERE BUT ADCONS
             .                                     AND VARIABLES
             .
            END      CZABC
```

NONPRIVILEGED PROGRAMS

We're not going to say a great deal about writing nonprivileged system programs since most of the TSS/360 literature deals with writing nonprivileged programs; it would be redundant to repeat it here. Of particular interest to you, if you want to write nonprivileged system programs, is the information in Assembler User Macro Instructions and Assembler Programmer's Guide.

There isn't a great deal of difference between privileged and nonprivileged system program; almost everything we've said about privileged system programs applies to nonprivileged system programs. The most significant difference between them is that nonprivileged system programs operate with a program status word protection key of 1; they cannot read or write privileged control sections.

OPERATING ENVIRONMENT

A nonprivileged program operates in a virtual machine. The storage of this machine contains all the programs that make up IVS and any other programs that have been brought into virtual storage by the dynamic loader. A nonprivileged system program may be part of IVS; assembler is an example of nonprivileged initial virtual storage programs.

A nonprivileged program may use any System/360 problem state instruction and any of the nonprivileged supervisor call instructions. Nonprivileged system programs may not, in general, use the privileged supervisor call instructions. (Remember, if the logged-on user is a system programmer, any SVC that does not violate the privileged status of the issuing program can be used. SVCs cannot be used indiscriminately, however. For example, privileged or not, you can't issue an SVC 121 (ENTER) if your program is running in the privileged state; this is considered an error.

In essence, we're saying that a nonprivileged program cannot issue a privileged SVC and vice versa. The resident supervisor will find out from the task status index that such a program cannot issue privileged SVCs (whether or not the SVC was correctly used). On the other hand, you, as a system programmer, are allowed by the supervisor to issue any SVC, privileged or otherwise, but there is no guarantee that you'll do it correctly.

Since a number of SVC codes are used by the resident supervisor, a kind of substitute SVC, called ENTER, is used for most transfers of control from nonprivileged to privileged programs. A nonprivileged program can't transfer to a privileged program via a branch instruction since all privileged programs are fetch protected from all nonprivileged programs (both system and user). ENTER codes, analogous to SVC codes, are used by the task monitor to figure out where to transfer control. We'll have more to say about ENTER when we discuss it as a nonprivileged SVC.

PROGRAM DESIGN CONSIDERATIONS

In thinking about nonprivileged programs, be careful not to confuse the privilege of a program with the authority of the programmer who directed that the program be loaded. Despite any declarations at assembly time, you, as a system programmer, may always issue privileged SVCs. Therefore, any problem you write is implicitly a system program as long as you LOGON using your S or O authority code. Remember that all sections you load using your S or O authority code are private -- the dynamic loader ignores the PUBLIC attribute.

In the section about conventions, we talked about fence straddlers. A fence straddler should never be designed to issue privileged SVCs based on the authority code of the user. If this were done, and a programmer with a user authority code (U) attempted to use the fence straddler, he wouldn't succeed. To be on the safe side, when you write system programs, you should always give the control sections the attributes they need to be able to run; do not rely on your authority code unless all intended users will have an equivalent authority code.

Nonprivileged system programs accessible by user programs have module names that begin with SYS. Analogous to the resident supervisor and privileged programs, control section and entry point names are formed by adding a character to the end of the module name. For instance, SYSABC is an entry point in the nonprivileged system program SYSAB. Names beginning with SYS can be freely referenced by all programs, privileged or otherwise; SYS names can only be defined by control sections with the SYSTEM attribute.

Nonprivileged system programs not accessible by user programs generally use symbols beginning with CE.

NONPRIVILEGED SUPERVISOR CALL INSTRUCTIONS

Nonprivileged supervisor calls are those whose processing programs are in virtual storage; these SVCs use codes 64 through 127. When a nonprivileged supervisor call is issued, the supervisor simply passes it back to the task monitor as a task-SVC; no task-program interrupts are generated. The task monitor transfers to the appropriate privileged program for processing. Nonprivileged SVCs are used to pass control from a nonprivileged program to a privileged program. Since nonprivileged programs can neither read, write, nor transfer control to privileged programs directly, some form of interrupt is required. The nonprivileged SVCs described in this publication are listed in Table 8.

Table 8. Nonprivileged Supervisor Calls (SVC 64-127)

SVC 121	Enter privileged service routine	ENTER
SVC 127	Transfer to dynamic loader for external symbol resolution	DLINK
SVC 123	Enter delete program	DELET
SVC 125	Enter program checkout subsystem	PCSVC
SVC 119	Read command from SYSIN (conditional)	CLIC
SVC 118	Read command from SYSIN (unconditional)	CLIP
SVC 122	Enter command language director to end RUN	RTRN
SVC 120	Restore privilege	RSPRV

ENTER -- Enter Privileged Service Routine (R)

Name	Operation	Operands
[symbol]	ENTER	None

EXECUTION:

Supervisor

A task SVC interrupt is created to transfer control to the task monitor.

Task Monitor

The enter routine (part of the task monitor) transfers control to a privileged program using modified type-I linkage. The low-order byte of register 15 contains a code, the enter code, that is used by the enter routine to determine which privileged program is to receive control. Only the contents of registers 0 and 1 are passed to the privileged program; registers 0 and 1 are the only registers the privileged program can use to pass results back to the program issuing the ENTER. Registers 2 through 15 are saved and restored by the enter routine for the ENTER issuing program.

EXAMPLE: Suppose we want to get 256 bytes of working storage (without using the GETMAIN macro instruction). We might write,

```
        SR      1,1     CLEAR GP R1 TO SET OPTIONS
        LA      0,256   SET BYTE COUNT
        LA      15,48   SET ENTER CODE IN GP R15
NAME    ENTER
```

The ENTER would generate:

```
NAME   SVC     121
```

Note: For a list of the ENTER codes, see System Control Blocks PLM.

DLINK -- Transfer to Dynamic Loader for External Symbol Resolution (R)

Name	Operation	Operands
[symbol]	DLINK	None

EXECUTION:

Supervisor
The resident supervisor creates a task SVC for the task monitor.

Task Monitor

Control is transferred to the dynamic loader's dynamic-linkage routine. DLINK can be used for explicit linking (external symbol resolution and transfer of control to loaded program) or explicit loading (no transfer of control). DLINK must be the subject of an execute instruction. (For usage of DLINK, see CALL, LOAD, ARM, and ADCON in Assembler User Macro Instructions.)

EXAMPLE: Suppose you want to dynamically load a program called HELP and have control transferred to its entry point, BEGIN. You might write,

```
LOAD      EX      ADCNGRP
          B       AWAY
ADCNGRP   DS      OF
          DLINK
          DC      X'0100'        OPTIONS:  LOAD AND TRANSFER
          DC      CL8'BEGIN'
          DC      2F'0'
```

The DLINK would generate,

SVC 127

This will cause the dynamic loader to receive control from the task monitor via the supervisor; it will then load HELP and transfer to BEGIN.

DELET -- Enter Delete Program (nonstandard)

Name	Cperation	Operands
[symbol]	DELET	None

EXECUTION:

Supervisor
A task-SVC interrupt is created to transfer control to the task monitor.

Task Monitor
Control is transfered to the dynamic loader's delete routine (see Assembler User Macro Instructions for a description of DELETE.)

EXAMPLE: If you want to cause your program to enter the delete program within the dynamic loader, you would not use the ENTER mechanism; you could write,

```
          EX      0,NAME
          B       AWAY
NAME      DELET
          DC      CL8'DESTRYME'
          DC      X'0000'
```

DELET would expand as,

```
NAME      SVC     123
```

PCSVC -- Enter Program Checkout Subsystem (nonstandard)

Name	Operation	Operands
[symbol]	PCSVC	None

EXECUTION:

Supervisor
A task-SVC interrupt is created to transfer control to the task monitor.

Task Monitor
Control is transferred to the program checkout subsystem (PCS). This SVC is used by PCS to replace user instructions in response to the AT command (see Command Language User's Guide).

EXAMPLE: Suppose PCS wants to plant a transfer of control; it might be coded:

```
MOVE      MVC     NAME(2),PLANT
          B       AWAY
PLANT     PCSVC
```

The PCSVC would expand as an SVC 125.

CLIC -- Read Command From SYSIN (conditional) (nonstandard)

Name	Operation	Operand
[symbol]	CLIC	None

EXECUTION:

Supervisor
The supervisor creates a task-SVC interrupt for the task monitor.

Task Monitor
The task monitor transfers control to the command system, which checks to see if the task issuing the SVC is conversational. If a conversational task issued the SVC, the user at the SYSIN terminal is given an opportunity to enter a command (underscore, backspace, unlock keyboard). If a nonconversational task issued the SVC, nothing is done; i.e., the SVC has the effect of a NOP (no operation).

EXAMPLE: SVC 119 is used by the FORTRAN pause routine. Suppose you arrive at a point in your program where you want the terminal user running your program to be able to enter commands. You might write,

```
JFB   CLIC
```

This would generate,

```
JFB   SVC    119
```

If the SYSIN terminal user didn't want to enter any commands, he would enter RUN, followed by carriage return. If the task issuing the SVC 119 were being run nonconversationally, the SVC would be ignored.

CLIP -- Read Command From SYSIN (unconditional) (nonstandard)

Name	Operation	Operands
[symbol]	CLIP	None

EXECUTION:

Supervisor
The supervisor creates a task-SVC interrupt for the task monitor.

Task Monitor
The task monitor transfers control to the command system, which attempts to read a command from the SYSIN device.

EXAMPLE: SVC 118 is used by the FORTRAN halt routine. Suppose your program is finished and you want to return control to the terminal user or cause the command director to read the next command from a nonterminal input source. You might write,

```
DONE   CLIP
```

This would produce:

```
DONE   SVC      118
```

which would cause the command director to try to obtain a command from the SYSIN device.

Note: The CLIP macro instruction reads from the SYSIN data set and does not require a terminal; CLIC reads only from a terminal and must, therefore, only be used in a conversational task.

RTRN -- Enter Command Language Director to End RUN (R)

Name	Operation	Operand
[symbol]	RTRN	None

EXECUTION:

Supervisor
A task-SVC interrupt is created to transfer control to the task monitor.

Task Monitor
Control is transferred to the command language director (see Assembler User Macro Instructions, EXIT macro instructions).

EXAMPLE: If the program you caused to the run is finished and you want to return control to the command language director for end-of-run processing, you might write,

```
NAME  RTRN
```

This will produce,

```
NAME  SVC       122
```

RSPRV -- Restore Privilege (R)

Name	Operation	Operands
[symbol]	RSPRV	None

EXECUTION:

Supervisor
A task-SVC interrupt is created to transfer control to the task monitor.

Task Monitor
Control is transferred to the restore-privilege routine for the purpose of completing type-III linkage. The restore-privilege routine restores registers 2 through 14 to the values they contained when received from the privileged calling program. Registers 0, 1, and 15 are left unchanged (see the section on linkage conventions).

EXAMPLE: Suppose you have written a type-III program which has received control from the leave-privilege routine and is now ready to return control to the privileged calling program. You might write,

```
DEPART  RSPRV
```

which will expand as,

```
DEPART  SVC    120
```

You could also write,

```
BR      14
```

since register 14 is set to point to an SVC 120 by the leave-privilege routine.

As a system programmer you are well aware of the convenience and the power of the macro instruction. You are also familiar with the procedures for defining macro instructions that are outlined in Assembler Language. This section deals with the process of defining macro instruction, concentrating on precautions you should observe and limitations imposed by the various types of macro instructions.

This section has been organized around three basic types of macro definition: the R-type, the S-type, and the modified R- and S-types together with nonstandard.

R-TYPE MACRO DEFINITION

You may use the standard R-type macro instruction when all the subparameters can be contained in the two parameter registers 0 and 1. The R-type does not generate a parameter list but may generate constants or addresses. You are also limited in your choice of value mnemonics from the available set described in the introduction. We'll list the acceptable ones and some examples to illustrate the precautions you should observe.

addrx

You must remember to cover (with a base register) the addresses that may be written for an operand having this value mnemonic. Figure 14 shows a portion of the correct coding of the STORE macro definition. Notice the use of the LA instruction to provide an overriding base register for the STM instruction. You should not write

```
STM &REGS(1),&REGS(2),&AREA
```

The value mnemonic of &AREA is addrx which permits the coding of indexed addresses. But the STM instruction does not allow for indexing. In general, you must employ addrx-type operands only in instructions which are indexable. So, in the example, you would have used the operand &AREA in the LA instruction, which is indexable.

```
&NAME   STORE   &AREA,&REGS
          •
          •
          •
&NAME   LA      6,&AREA
        STM     &REGS(1),&REGS(2),0(6)
```

Figure 14. Coding addrx Operands

addx

The value mnemonic addx imposes the same restrictions as does addrx. This mnemonic, however, does not permit register notation.

integer

If you select integer as the mnemonic of the operand &INT several alternatives must be considered.

If the operand will always be less than 4096, you may write

```
LA 1, &INT
```

If the possibility exists that &INT will exceed 4095, you must first test its magnitude and, in the cases in which it does exceed this value, write

```
L 1,=F'&INT'
```

The F-type literal is chosen, rather than the R-type, for invarient data, to avoid organizing the literal in the user's first declared PSECT.

You may choose the mnemonic integer for an operand which is not a parameter but serves to indicate the proper path through the macro definition. This type of operand should be treated in conditional assembly instructions.

absexp

If the value of an absexp operand is less than 4096, you may use the LA instruction. If this value is greater than 4095, you must take into account the fact that the operand may be of the form '5280'. In this case, the instruction L 1,=F'&INT' would generate the instruction L 1,=F'C'5280". The apostrophes of the F-type constants argument, in the second operand, would be flagged as a syntax error.

You must first resolve the value of the operand by placing it in the operand field of a SETA instruction in this manner

```
          LCLA   &RIGHT          INITIALIZE THE SETA VALUE
&RIGHT    SETA   &INT            ASSIGN THE VALUE OF &INT TO &RIGHT
          L      1,=F'&RIGHT'
```

In this example the value of &INT is computed and the variable symbol &RIGHT is assigned that value. The literal may then be loaded into the parameter register.

The absexp operand may also be used as a path indicator and would be treated by conditional assembly instructions.

value

This type of operand may be written as an absolute expression or as register notation. In this case, you must test for the type of notation used as we have done in the general R-type example in Figure 16. Once you have determined the operand format, your processing should follow the appropriate rules.

code

A coded value may be enclosed in apostrophes or not. However, some macro instructions offer code or some other mnemonic as alternate choices for coding an operand. In these cases, it would not be possible to distinguish between the alternates without some kind of test. The simplest way to handle this possibility is to require the use of delimiting apostrophes and code your macro definition to test the operand for a leading apostrophe.

If the coded value is to be passed in a register as a parameter, restrict it to four characters; if two parameter registers can be used, restrict it to eight.

You may choose a code to indicate the path to be taken through the macro expansion or to be passed as a parameter in some form other than character string. In this latter case, you must provide a translation algorithm through the use of conditional assembly instructions.

text and characters

You will rarely use these mnemonics in an R-type macro instruction, but might choose to pass a character string parameter in one or a pair of registers. If you choose to do so, be sure to limit the size of the string to conform with the amount of available register space.

You may use a character, self-defining term as the displacement field of an LA instruction if the string consists of one character. If the string is longer than one character, your macro definition must employ the L instruction to load a literal.

symbol

You may specify this mnemonic if you want to force the writer of the macro instruction to specify a character string which conforms to assembly language conventions.

You may also permit the writer to provide a symbolic name for the first executable instruction in the expansion. If so, be sure to provide for the inclusion of the name with each model statement which may generate the first executable instruction. Figure 14 gives an example of this.

LINKAGE

Nearly all the routines called by macro instructions are privileged. If the module issuing the macro instruction is privileged, the macro instruction must generate a type-1 linkage; if the issuing module is nonprivileged a type-2 linkage must be generated. If a macro instruction may be issued by either type of module, then your macro definition must test for the privilege class.

The privilege class is set by the DCLASS macro instruction and is contained in the global SETB symbol &CHDCLS. If the DCLASS macro instruction specifies USER class or is omitted, &CHDCLS is given a value of 0; if PRIVILEGED is specified, &CHDCLS is given a value of 1.

Some macro instructions generate only type-1 linkage regardless of the issuing module's privilege class. If you write one of these so-called "fence-straddlers", be sure it's entry point name begins with SYS. These characters will be used to generate a type-1 linkage.

Finally, some macro definitions generate code without reference to parameters. That is, the same code is generated every time the macro defintion prototype name appears in a source program.

EXAMPLE: Here is an example of a typical R-type macro instruction and it's associated macro definition which illustrates some of the points just made. Your macro description would be

```
┌────────┬────────┬──────────────────────────────────────────────┐
│name    │operator│operand                                       │
├────────┼────────┼──────────────────────────────────────────────┤
│[symbol]│RTYPE   │loc-addrx,len-value                           │
└────────┴────────┴──────────────────────────────────────────────┘
```

and your macro defintion might look like this

```
(1)          MACRO                                    HEADER STATEMENT
(2)  &NAME   RTYPE    &LOC,&LEN                       PROTOTYPE STATEMENT
(3)          AIF      (T'&LOC EQ 'O').E1              IF 1ST OPERAND IS
                                                      MISSING
(4)  *                                                GENERATE AN ERROR
                                                      STATEMENT
(5)          AIF      ('LOC'(1,1) EQ'(').RNOT         IS FIRST OPERAND
                                                      REGISTER NOTATION
(6)  &NAME   LA       1,&LOC                          FIRST GENERATED
                                                      STATEMENT
(7)          AGO      .CP2
(8)  .RNOT   ANOP
(9)  &NAME   LR       1,&LOC                          FIRST STATEMENT IF
                                                      REGISTER NOTATION
(10.) .OP2   AIF      (T'&LEN EQ 'O').E2              IF 2ND OPERAND IS
                                                      MISSING
(11)         AIF      ('&LEN'(1, 1) EQ'(').RNOT2      IF 2ND OPERAND
                                                      REGISTER NOTATION
(12)         AIF      (&LEN GT 4095).LLIT
(13)         LA       0,&LEN
(14)         AGO      .LINK
(15) .LLIT   ANOP
(16)         LCLA     &A                              INITIALIZE SETA
                                                      SYMBOL
(17) &A      SETA     LEN&                            SET VALUE CF SETA
                                                      SYMBOL
(18)         L        0,=F'&A'
(19)         AGO      .LINK
(20) .RNOT2  LR       0,&LEN(1)
(21) .LINK   CHDINNRA ,,(CZCXYZ),X'FF'
(22)         MEXIT                                    TERMINATE PROCESSING
(23) .E1     ANOP                                     1ST OPERAND MISSING
(24) .E2     ANOP                                     2ND OPERAND MISSING
(25)         MEND                                     TRAILER STATEMENT
```

In this example, line 3 tests for the presence of a first operand and, if it is missing, branches to an ANOP statement in line 23. In practice you would want to place some error processing code at this

point. We'll discuss error processing and the CHDERMAC macro instruction later.

Line 5 tests for register notation by determining if the first character of the operand &LOC is a left parenthesis.

Line 6 is the model statement which generates the first executable instruction for nonregister notation and would also generate the name assigned to the macro instruction.

Line 9 would generate the first instruction in register notation and also contains the symbolic parameter &NAME in the name field. Line 6 and line 9 would never be generated together.

Notice the technique employed in lines 5 and 8. Line 5 determines if line 6 or line 9 should generate the first instruction and the symbolic name. In branching to line 9 the use of &NAME would be ambiguous, so an ANOP instruction, named .RNOT, is inserted in line 8 and a branch is taken to it.

The second operand is processed in much the same way. Notice that line 12 tests the magnitude of the operand &LEN and lines 15 through 18 cover the situation in which &LEN is greater than 4095.

Finally, line 21 generates the linkage by means of the CHDINNRA inner macro instruction, which we'll also discuss later. The third operand, (CZCXYZ), represents the type-1 linkage entry point and the fourth operand represents the ENTER code for type-2 linkage. You will see that CHDINNRA determines which type linkage to use.

S-TYPE MACRO DEFINTIONS

You should employ the S-type macro definition when you wish to generate a parameter list in storage because the parameters cannot be contained in two registers. When writing an S-type macro definition, bear in mind that, by convention, three forms of S-type definitions are required.

The standard form, indicated by the keyword operand MF=I or by the omission of the MF=operand, generates a parameter list and the required linkage to the called routine.

The L-form, indicated by MF=L, only generates a parameter list; it does not generate any executable code. For this reason, register notation is not allowed in the L-form.

The E-form, indicated by MF=(E,parloc- addrx (1)) generates the proper linkage and may also alter an existing parameter list.

This convention permits the programmer using your macro instructicn to conserve space in storage by generating a parameter list by means of the L-form and the altering the same list, in subsequent calls, by means of the E-Form.

The placement of the parameter list may be indirectly controlled by the user of your macro instruction and he should be advised about these precautions:

1. The S-type macro instruction places the parameter list in the first declared PSECT of the assembly module.

2. If this PSECT is declared by a macro instruction, then that instruction must appear in the user's program before any macro instructions which reference the list.

3. If rule 2 is violated, or if no PSECT exists at all, the standard form S-type macro instruction must place the parameter list in line with the code it generates and insert a branch around the list.

4. L-form macro instructions always generate the parameter list in line. Therefore, if the user is writing a reenterable body of code, he will want the parameter list generated in the area occupied by his working storage, presumably his PSECT. This is done for him by the standard form S-type, but the L-form may only be used in the PSECT.

STANDARD-FORM S-TYPE MACRO DEFINITION

As in the case of R-type macro definitions, the value mnemonics you choose will dictate certain steps in your macro definition. Here are some precautions for you to observe.

addr and relexp

Since relexp is implied by addr and, in turn, implies both relocatable and complex relocatable expressions, your macro definition must treat such operands by using them in the argument of an A-type address constant. This A-type address constant must be generated as a DC statement or as an A-type address constant literal. You must also include a test for register notation since this is allowed by the mnemonic addr.

integer, absexp, and value

If the operand specified by one of these mnemonics is an actual numeric value, it is only necessary to generate an A-type address constant bearing in mind any size constraints which might necessitate the use of length modifiers.

The mnemonic value permits register notation, and you would have to test for this. If register notation is used and the register contains the parameter, the register contents may be placed into the list. If the register points to a user supplied list, you must supply space and move the data in. You may also choose one of these mnemonics for an operand which is a path indicator.

code

If the coded value is to be passed as a parameter or is to be translated to a value to be passed as a parameter, you must pass it in the parameter list and not, as in the case of the R-type macro instruction, in a register. Since the coded value may only be one term, you may employ any type of constant to generate the parameter in the list. If the coded value is a character string which includes apostrophes, you must pass it as a character constant and adhere to the rules for writing such constants. Notice also that the TSS/360 Assembler will reduce all double apostrophes and double ampersands to single apostrophes and ampersands.

Again, you may choose to use the coded value as a path indicator. If you wish to pass a variable-length parameter list, you might use a coded value to indicate the length of the list being passed.

text and characters

These types of operands may be used in two ways. You may choose to pass the operands to the called routines as character strings in the parameter list or may choose to generate the character strings and then enter a pointer to them in the parameter list. Since the parameter list

produced by the S-type macro instruction normally is a list of pointers, you will, with few exceptions, use these operands in character constants or character literals.

You are responsible for verifying the presence of a leading apostrophe in a text operand and for providing error processing in the event that it is missing. The assembler program checks for the terminal apostrophe.

Two methods for checking the length of a character string are available to you. As you can see in Figure 15, you may test for either the K or the L attributes. The reason for subtracting two from the count of &text before placing it in the SETA cell is that the assembler will include the delimiting apostrophes in the count. If you choose to ascertain the length attribute of the character string, bear in mind that delimiting apostrophes will have been stripped and double apostrophes and ampersands will have been reduced. Thus, had the programmer written the operand &TEXT

```
'USE THIS SYMBOL &&'
```

you would find its K attribute to be 20 (including terminal apostrophes). Statement 3 of the example would yield the value 18 in the SETA cell. Statement 5 would yield a value of 17 in the SETA cell since statement 4 would have generated &TEXT stripped of its terminal apostrophes and only one of the two ampersands.

```
(1)              MACRO
(2)              MACX       &TEXT
                   .
                   .
                   .
(3)      &A1     SETA       K'&TEXT-2
                   .
                   .
                   .
(4)      CHDXX   DC         C&TEXT
                   .
                   .
                   .
(5)      &A1     SETA       L'CHDXX
                   .
                   .
                   .
                 MEND
```

Figure 15. Determining the Length of a Character String

symbol

You may use this type of operand in any of several ways: in the name field of a generated statement, as a character string to be passed as a parameter, or as an entry point or module name to be used as the argument of an address constant, usually R-type or V-type.

L-FORM S-TYPE MACRO DEFINITION

Register notation is not allowed in the L-form of the S-type macro instruction. The L-form is used to generate a parameter list only. Since register notation would require the generation of executable code to store the register contents, it is to be avoided.

EXAMPLE A: Coding an S-Type Macro Instruction

```
(1)          MACRO                                          HEADER STATEMENT
(2)  &NAME   STYPE  &LENLOC, &PROC, &SYM, &SYMLEN,          PRCTOTYPE
                      &MF=1
                     .
                     .
                     .
(3)  .LFORM  AIF     ('&NAME' EQ '') .E1                   IS NAME FIELD OK
(4)          AIF     (K'&LENLOC EQ 0) .OMIT1               IS FIRST FIELD OK
(5)  &NAME   DC      A (&LENLOC)                           ENTER FIRST OPERAND
(6)  .SYM    AIF     ('&SYM' EQ '') .E2                    3RD OPERAND OK
(7)          DC      CL8'&SYM'                             ENTER 3RD OPERAND
(8)          AIF     (K'&SYMLEN EQ 0) .E4                  4TH OPERAND OK
(9)          DC      AL1 (&SYMLEN)                         ENTER 4TH OPERAND
(10)         LCLB    &B                                    ESTABLISH SETB
(11) .PROC   AIF     (K'&PROC EQ 0) .OMIT3                 IS 2ND OPERAND PRESENT
(12)         AIF     ('&PROC' NE 'E' AND                   IS 2ND OPERAND VALID
                       '&PROC' NE 'P') .E3
(13) &b      SETB    ('&PROC' EQ 'F')                      SET CODE
(14) .OMIT3  DC      ALL (&B)                              DEFAULT 2ND OPERAND
(15)         MEXIT                                         DEFAULT 2ND OPERAND
(16) .OMIT1  ANOP                                          DEFAULT 1ST OPERAND
(17) &NAME   DC      A (0)                                 RESUME PROCESSING
(18)         AGO     .SYM
(19) .E1     ANOP
(20) .E2     ANOP
(21) .E3     ANOP
(22) .E4     ANOP
                     .
                     .
                     .
(23)         MEND
```

Example A points up this constraint in a subtle manner. If you intend to permit the user of your macro instruction to employ the L-type, you might want to highlight this point in your macro description. All other value mnemonics allowed in the standard form are also allowed in the L-form. Although operands in this form may be used as path indicator, they are generally used as the arguments of DC statements or are translated to values which are used as arguments.

Since the user has complete control of the placement of the parameter list, you needn't concern yourself with including a statement to generate a control section; specifically, don't attempt to locate the parameter list in the PSECT.

EXAMPLE: Notice that the L-form macro instruction shown in Figure 16 indicates the name field as being mandatory. This is a good general rule to follow because most users will generate the parameter list and later modify it with an E-form. The name assigned is the only safe way to identify the parameter list for later modification. Also take note of the subtle changes in value mnemonics that eliminate the use of register notation.

The coding shown in Example A would generate the parameter list shown in Figure 17. Statements 3, 4, 6, 8, 11, and 12 test for the existence and the validity of each parameter. Statements 5, 7, 9, 14, and 17 generate the parameter list. Notice that lines 3 and 6 employ a null

test in verifying the presence of operands with a value mnemonic of symbol.

Standard form

Name	Operation	Operand
[symbol]	STYPE	lenloc-addr,proc-{F|P}, sym-symbol,symlen-value [,MF=I]

L-form

Name	Operation	Operand
symbol	STYPE	[lenloc-relexp] [,proc{F|P},] sym-symbol,symlen-absexp, MF=L

Figure 16. Standard and L-form S-type Macro Description

Default options have been provided in lines 14 and 17. If the second operand is omitted, line 11 branches to line 14 which uses &B as the argument of the address constant. Since &B was initialized to zero by the LCLB instruction, line 14 defaults to zero (indicating P).

In your definition, lines 19 through 22 would be followed by error processing.

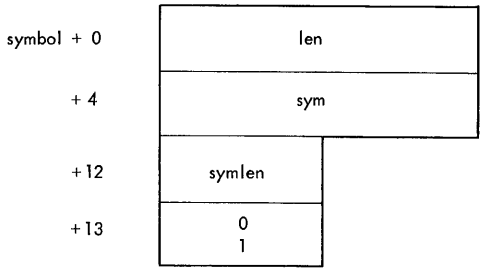


Figure 17. Parameter List Generated by L-form

E-FORM S-TYPE MACRO DEFINITIONS

The E-form macro instruction may modify a parameter list and may generate the linkage to the called routine. Since this requires the generation of executable instructions, some changes must be made in the value mnemonics.

addrx

This type of operand must be substituted wherever addr was specified in the standard form. The use of these operands will differ from their use in the standard form in that you will use them to compute the effective address and then store that address in the parameter list. You should alert the user to the fact that base register coverage must be provided.

By convention, general registers 14 and 15 are used as working registers in the macro definition because the linkage you generate will destroy their original contents anyway.

integer, absexp, and value

Operands specified by these value mnemonics are treated much the same as in the R-type. A Load or Load Address instruction is used to load register 14 with the operand value. The choice of instructions again depends on the magnitude of the operand value.

Because register notation must be allowed in the MF= operand, you must include a test for it and provide for loading register 1 from the register specified. If register notation is specified for other operands, utilize the register specified as a working register and generate an appropriate store instruction.

code and symbol

These operands may be used as path indicators and symbol may be used to name generated instructions. Although operands of this type are permitted in the E-form, you will seldom find them useful.

Linkage

When your macro definition is generating the linkage to the called routine, you should generate the entry point in a V-type address constant literal. Not only is this a convenient method, but the assembler program will place this constant in the proper control section -- a PSECT, if one exists.

When generating a Type I linkage, your E-form macro definition must generate both V-constant and R-constant literals. You can use the inner macro instruction CHDINNRA, which we will discuss later, for this purpose.

EXAMPLE B: Coding an E-form S-type Macro Instruction.

```
(1)          MACRO                                          HEADER STATEMENT
(2)  &NAME   STYPE    &LENLOC,&PROC,&MF=I                   PROTOTYPE
               .
               .
               .
(3)  .EFORM  ANOP                                           ENTRY POINT
(4)  &NAME   DS       0H                                    ALIGNMENT
(5)          CHDINNRA   &MF(2)                              LIST ADDRESS IN
                                                            REGISTER 1
(6)          AIF      (K'&LENLOC EQ 0).PROC                 IS THERE A 1ST
                                                            OPERAND
(7)          AIF      ('&LENLOC'(1,1) EQ '(').RNOT1         IS IT REGISTER
                                                            NOTATION
(8)          LA       14,&LENLOC
(9)          ST       14,0(0,1)                             1ST OPERAND TO LIST
(10)         AGO      .PROC
(11) .RNOT1  ST       &LENLOC(1),0(0,1)                     1ST OPERAND TO LIST
(12) .PROC   AIF      (K'&PROC EQ 0).LINK                   IS THERE A 2ND
                                                            OPERAND
(13)         AIF      ('&PROC' NE 'F' AND '&PROC' NOT EQ 'P').E1
                                                            IS IT VALID
(14)         LCLB     &B                                    ESTABLISH SETB CELL
(15) &B      SETB     ('&PROC' EQ 'F')                      SET SETB CELL
(16)         LA       14,&B
(17)         STC      14,13(0,1)                            STORE CODE IN PARA
                                                            LIST
(18) .LINK   CHDINNRA   ,,(CZCXYZ),X'FF'                    GENERATE LINKAGE
(19)         MEXIT
(20) .E1     ANOP
(21)         MEXIT
```

Figure 18 demonstrates the E-form of the macro instruction described in Figure 16 and the parameter list shown in Figure 17. The coding for a typical E-form S-type macro instruction is shown in example B.

Name	Operation	Operand
[symbol]	STYPE	[lenloc-addr] [,proc-{F / P}], MF=(E,parloc-{addrx / (1)})

Figure 18. E-form S-type Macro Description

The logic of this macro definition should be clear in the light of previous descriptions. Note that the only error test is for an invalid second operand. All other parameters may be omitted if no change is desired in that field of the parameter list.

MODIFIED R-TYPE MACRO DEFINITIONS

You may choose to pass parameters in registers other than 0 and 1; this makes the definition a modified R-type. If your macro instruction links to the called routine by means of an SVC, you may pass parameters in registers 14 and 15.

Should you choose to pass parameters in other registers (i.e., 2 through 13) you must save and restore these registers for the user.

No change in value mnemonics occurs between R-type and modified R-type macro instructions.

MODIFIED S-TYPE MACRO DEFINITIONS

An S-type macro instruction may have no standard form and an E-form that does not generate any linkage. These are modified S-type macro instructions and they serve only to generate and to alter a parameter list.

Another type of modified S-type macro instruction is the type that has only a standard form but neither an L-form nor an E-form.

NONSTANDARD MACRO DEFINITIONS

These macro instructions generate a parameter list and/or inline code but not a linkage. Your selection of value mnemonics is unlimited in this type.

TECHNIQUES USED IN WRITING MACRO DEFINITIONS

REGISTER NOTATION

Special register notation should be specified when you wish to allow the user to load parameters into the required registers before execution of the macro instruction. By convention, these registers are restricted to 0 and 1 for standard R-type macro instructions. Registers 14 and 15 may also be used if the R-type macro instruction can be of the modified type. Your method of linkage may place additional restrictions on the use of registers 14 and 15.

Register notation shall be limited to registers 2 through 12 in order to avoid the loss of parameters. Let us assume that you want to pass parameter P1 in register 0 and parameter P2 in register 1. Without this restriction on register usage, the user might issue the HAVOC macro instruction like this:

```
        L   R0,P2
        L   R1,P1
HAVOC   (0),(1)
```

Your macro expansion would then do this:

```
        LR  R0,R1   This is parameter P1
        LR  R1,R0   This is not parameter P2
```

If you use registers other than the conventional ones for working registers, and do not save and restore them, be sure to caution the user that those registers are volatile.

PACKING PARAMETERS

If you wish to pass two parameters in one register, you must pack the parameters. Figure 19 shows the methods for packing two parameters, each a half word long, into register 1.

	&OPA is register notation	&OPA is an absolute expression
&OPB is register notation	LR 1,&OPA(1) SLL 1,16 OR 1,&OPB(1)	LCL&A &A SET&OPA*65536 L 1,=F'&A' CR 1,&OPB(1)
&OPB is an absolute expression	LR 1,&OPA(1) SLL 1,16 LCLA &A &A SETA &OPB O 1,=H'&A'	LCL&A &A SET&OPA*65536 L 1,=F'&A' &A SET&OPB C 1,=F'&A'

Figure 19. Packing Two Halfword Parameters Into Register 1

Similar techniques can be used for other cases. Here are two examples:

EXAMPLE C:

Parameter P1 - three bytes left aligned
Parameter P2 - one byte right aligned
Both parameters given in register notation

Procedure

```
        LCLA  &A
&A      SETA  &P1*256
        L     1,=F'&A'
&A      SETA  &P2
        O     1,=F'&A'
```

EXAMPLE D:

Parameter P1 - one byte left aligned
Parameter P2 - three bytes right aligned
Both parameters given as absolute expressions.

Procedure

```
        LCLA  &A
&A      SETA  &P1
        L     0,=F'&A'
&A      SETA  &P2*256
        L     1,=F'&A'
        SRDL  0,8
```

We have already employed an inner macro instruction, CHDINNRA, in previous examples. You will find the use of this type of instruction not only convenient but also economical, in terms of lines of code written, lines of code generated, and time expended in assembly.

The assembler retains a copy of the inner macro instruction in virtual memory. Your first reference to the instruction will cause it to be read into main storage from the library but successive references will not require this delay.

You needn't write as much code in your outer macro definition since the inner macro instruction will supply it for you. You may also reduce the number of generated statements by the user of conditional calls. You might, for example, write

```
          AIF   ('&OP' GT'5').INR
           .
           .
           .
.INR      ANOP
           .
           .
           .
```

The inner macro instruction would only be called if &OP were greater than five. As you can see, the use of the inner macro instruction is somewhat analogous to the use of a subroutine.

Basically, the same criteria should govern your use of inner macro instructions as govern your use of subroutines. If time and space will be saved, define and use an inner macro instruction.

Don't nest more than three levels of macro definition. This technique will keep the definition clean and intelligible.

Don't define an inner macro instruction for only one outer macro instruction; use a conditional assembly subroutine instead. Let us assume you want to conditionally enter a subroutine from points A, B, and C. You must provide a means by which the subroutine can return to the correct point after each of the three calls. You can do this by establishing a SETC call and altering its contents prior to each conditional branch. Example E illustrates this technique.

EXAMPLE E: Branching To and Returning From a Conditional Subroutine

```
          MACRO
           .
           .
           .
          LCLC   &RTRN
           .
           .
           .
&RTRN     SETC   '.RTRN1'
          AIF    ('&OP1' GT 10).SR
.RTRN1    ANOP
           .
           .
           .
&RTRN     SETC'  .RTRN2'
          AIF    (K'&OP2' EQ 0).SR
.RTRN2    ANOP
           .
           .
           .
&RTRN     SETC   '.RTRN3'
          AIF    (L'&OP2' LT 1).SR
RTRN3     ANOP
           .
           .
           .
.SR       ANOP
           .
           .
           .
          AGO    &RTRN                          (END OF SUBROUTINE)
           .
           .
           .
          MEND
```

NAMING THE FIRST EXECUTABLE INSTRUCTION

If a given instruction may be conditionally assembled as the first or second executable instruction, you will find it convenient to generate the statement

```
NAME   DS   0H
```

this provides a vehicle for the symbol regardless of which instruction comes first.

SETTING THE SIGN BIT

If you define a macro instruction in which the user may specify the sign of operand two by the presence (negative) or the absence (positive) of operand one, you may find it difficult to properly set the sign bit. Examples F and G illustrate two techniques you might find helpful.

After establishing SETA and SETB cells, line 5 places the proper bit value in the SETB cell, based on the presence or the absence of OP1. Line 6 then generates the parameter in storage using the value in the SETB cell for the sign. Notice that the length modifiers in line 6 specify the length in bits. You must use one line for the DC statement. If you use two lines, the second line will be aligned on a byte boundary and the sign bit will be lost.

In the second example, line 7 tests for the absence of OP1. If it is absent, the sign is to be positive and lines 15 and 16 generate a positive value in register one.

If the sign is to be negative and OP2 is zero, line 12 generates a negative zero in register one.

If OP2 is to be a negative number other than zero, line 9 computes the two's compliment of OP2 and places it in the SETA cell. Line 10 loads register one with the negative SETA symbol. The assembler will convert the value in &A to its negative two's compliment. Since the value in &A is already the two's compliment of CP2, line 10 will load the absolute value of OP2 with the sign bit on.

EXAMPLE F:

```
(1)             MACRO
(2)    &NAME    MACEX   &OP1,&CP2
(3)             LCLA    &A
(4)             LCLB    &B
(5)    &B       SETB    (K'&OP1 NE 0)
                  .
                  .
                  .
(6)             DC      AL.1(&B),AL.31(&CP2)
                  .
                  .
                  .
```

EXAMPLE G:

```
(7)             AIF     (K'&OP1 EQ 0).CMIT
(8)             AIF     &OP2 EQ 0) .ZERC
(9)    &A       SETA    X'7FFFFFFF'-(&CP2-1)
(10)            L       1,=F'-&A'
(11)            AGO     .DONE
(12)   .ZERC    L       1,=X'80000000'
(13)            AGO     .DONE
(14)   .OMIT    ANOP
(15)   &A       SETA    &OP2
(16)            L       1,=F'&A'
(17)   .DONE    ANOP
                  .
                  .
                  .
```

PROCESSING A SINGLE APOSTROPHE

You must exercise caution in the treatment of operands which may validly contain single apostrophes. If, for example, a single apostrophe is found in a character relation in a character relation in an AIF instruction, it will produce invalid syntax.

There is a special technique you can employ to test for single apostrophes without violating syntax rules. You might write

```
AIF ('&OPND'(1,1).'&OPND'(1,1) EQ '''').TEXT
```

This use of substring notation concatenates the operand field you want to test with itself, thereby generating a pair of the tested character. Thus if the character tested is an apostrophe, paired apostrophes will be produced and no violation of the rules of syntax will result.

It's worth noting here that there are three methods available tc you to test for the presence of an operand

1. AIF (K'&OPERAND EQ 0).OMIT

2. AIF (T'&OPERAND EQ '0').OMIT

3. AIF ('&OPERAND' EQ '').CMIT

Method one tests for a count of zero, two tests for a type of "omitted", and three tests for a null character string. You should nct use this latter method if it is possible for a single apostrophe to appear in the operand. A test for the K attribute is your best course.

REFERENCING THE DCB

If the macro instruction you define must reference the user's DCB, express references to the various fields in terms of actual byte displacements from the origin of the DCB. The use of symbolic field names would require the user to have previously issued a DCBD macro instruction or the macro instruction currently being defined must issue a DCBD inner macro instruction. The former requires an unwarranted assumption on your part, while the latter could result in multiply defined terms if the user has issued a DCBD macro instruction.

SIZE LIMITATION

If the operand is not a sublist, it may contain no more than 255 characters. If the operand is a sublist and the only references are to individual members of the sublist, each member may be up to 255 characters long. You are not restricted in the number of operands or in the number of sublist elements.

ADDRESS CONSTANTS

If a R-type address constant refers to a symbol defined in a program which has no PSECT, then the R-value defined is the origin of the control section containing the ENTER statement whose operand field contains the argument of the R-type constant. Thus, given R(X), where X is defined in an assembly module having no PSECT, the R-value is the origin of the control section containing the statement:

ENTRY X

If there is a PSECT, all address constant literals will be located in it. If no PSECT exists, the constants will be placed in the proper literal pool.

V-type and R-type constants must have only a single relocatable symbol as an argument. If an operand is to become the argument of such an address constant, you should show its value mnemonic as "symbol".

A symbol X and a V-type constant with an argument of X may both be defined in one assembly module. Unless X is also defined as an ENTRY point to the module, the V-type constant will be resolved by searching for a definition of X outside the current module.

Testing the type attribute of a symbol for the value T will only indicate whether it is defined as the operand of an EXTRN statement in the assembly. If a symbol is externally defined as the argument of a V-type or R-type address constant, its attribute will be given as U fcr undefined. This cannot be considered as a conclusive test, however,

since U is also the attribute assigned to symbols internally defined by an EQU statement. The test for a type attribute of T can only be used to indicate a symbol externally defined by means of an EXTRN statement.

When establishing addresses for entry to a routine that may or may not be in the current assembly module, it is best to use a pair of A-type constants and to require the user to define them with EXTRN statements if the routine is not in the same module. The use of a V-type and an R-type would require the use of ENTRY statements in the defining module if the routine is internal to the current assembly module. Because this latter requirement is so unnatural, A-type constants are better.

If your macro instruction generates an implicit adcon group and may be called from a user written program, it is not safe to assume that he has defined its entry points with either EXTRN or ENTRY statements. The type of constants you generate should be determined by testing for the T-value of the type attribute. If it is present, you may generate a V-type and R-type constant pair. If it is not present, generate an A-type constant pair. Admittedly, the user may have defined the symbols with ENTRY statements but, since there is no way to test for them, this is your only safe course.

Conventionally, the R-value of the A-type constant is assigned from &SYSPSCT (i.e., the first PSECT). If this variable is null, indicating that no PSECT exists, the R-value is assigned (from &SYSECT) the origin of the CSECT from which the macro instruction has been issued.

An exception to this convention occurs in the ADCON macro instruction. Since the user controls the placement of the ADCON macro instruction and probably wishes the adcon group constructed in the same PSECT, the R-value is always assigned the value from &SYSECT. You should employ this same technique when you wish to allow the user the flexibility of declaring more than one PSECT and generating the adcon group in a PSECT other than the first.

TERMINAL APCSTROPHE AND SIZE LIMITATION

Assume that a user writes a text operand which is 258 characters long, including terminal apostrophes. After you have tested for the initial apostrophe, you seek to determine the K attribute. Since this attribute operates modulo 256, you would receive a character count of 2: the initial apostrophe and the first character. The terminal apostrophe would be missing and an error message would be generated by the assembler program.

KEYWORD OPERANDS AND STANDARD VALUES

If you write a macro definition and include a keyword operand to which you assign a standard value, then the type attribute of the standard value will be assigned to the operand if it is completely omitted by the user. If he writes KEYWORD=, and follows the equal sign by a blank or a comma, the type attribute of the operand will be O for omitted, and the standard value is overridden by the explicitly specified null string.

SUBSTRING NOTATION PROCESSING

If you employ substring notation to refer to a subset of characters in a character string, you must first ensure that the characters are present. Assume, for example, that you want to test the first four characters of an operand to see if they specified some specific acticn

to be taken. You should write something like this:

```
AIF  (K'&OP LT 4).ERROR
AIF  ('&OP'(1,4) EQ 'REG1').PROC
```

If you don't do this and the user codes some character string of less than four characters, the assembler will produce error diagnostics.

This technique must be employed where register notation is allowed. Since you will employ substring notation to test for the opening parenthesis, you must first determine that the operand has been coded. The user may have chosen a default option and omitted the operand.

Notice also that you can access a character subset in an element of a sublist by writing something like this:

```
'&OPERAND(2)' (1,1)
```

This refers to the first character of the second element in the sublist &OPERAND.

N ATTRIBUTE USAGE

The N attribute counts the number of operands or the number of elements in a sublist by counting the number of commas and adding one. As a result, the N attribute cannot be used to count the number of non-null operands or non-null elements in a sublist.

N'&SYSLIST HANDLING IN MIXED MODE MACRO INSTRUCTION

Keyword operands are not included in the value of the N attribute of &SYSLIST in mixed mode operands. If there are no positional operands, N'&SYSLIST is zero.

SUBSCRIPTS AND SUBLISTS

If a subscripted reference is made to an operand which is not a sublist, the whole operand will be used. Thus, if you write DC A(&OP(15)) and the operand is not a sublist, you will generate the operand as the argument of the A-type constant just as if you had written DC A(&OP).

SETC SYMBOL LENGTH

The maximum length of a SETC symbol is eight characters. As a result, you may not be able to use in its entirety the operand of a SETC statement written as a relocatable symbol, absolute expression, text, cr character string. Instead, it is better practice to use the operand in groups of eight, being careful to test for the presence of characters before attempting to use them.

LOGICAL TERMS IN RELATIONAL EXPRESSIONS

When a relational expression is used in the operand of an AIF or SETB statement, the terms on either side of the relational operator must both be arithmetic expressions or character expressions; neither of the terms can be logical expressions. This is illustrated in the samples below, only some of which are valid. &B(1), &B(2), and &B(3) are SETB variables.

```
valid     AIF   ((&B(1)+&B(2)+&B(3)) NE 0).ON
invalid   AIF   ((&B(1) OR &B(2) OR &B(3)) EQ 0).ON
invalid   AIF   (&B(1) EQ 0).CN
valid     AIF   ((&B(1)+0) EQ 0).ON
valid     AIF   (&B(1)).ON
valid     AIF   (&B(1) OR &B(2) OR &B(3)).ON
```

INNER MACRO INSTRUCTIONS

We have spoken previously of inner macro instructions that could be used as closed subroutines in a macro definition. Three such inner macro instructions exist and have been cited in previous examples. The following paragraphs are designed to describe these macro instructions so that you may make use of them.

CHDINNRA -- Generate Type-1 or Type-2 Linkage (nonstandard)

The primary function of the CHDINNRA macro instruction is to generate a type-1 or a type-2 linkage or to generate a linkage by means of an SVC. It may also be employed to furnish the limited services of loading general registers 0 and 1 with specific parameters or loading the second element of the MF sublist in the E-form of the S-type macro instruction.

Name	Operation	Operand
symbol	CHDINNRA	[paraone-{addrx / (1)}], [parazero-{addrx / (0)}], [([sublista-{symbol / integer}][,sublistb-integer])], [entrcd-absexp,] [mcrcd-code]

paraone
specifies a parameter to be loaded into register 1.

parazero
specifies a parameter to be loaded into register 0.

sublista
is the first element of a two element sublist. If specified by itself, it designates the entry point for a type-1 linkage. If specified together with the second element, it indicates the relative byte location within the DCB at which OPEN has placed the R-value. This parameter may be omitted. If it is, and sublistb is also omitted no linkage is generated.

sublistb
is the second element in the sublist. If it is specified by itself, it is interpreted as the integer specified in the operand field of an SVC instruction. If specified together with sublista, it is interpreted as the relative byte location of the V-value within the DCB.

entrcd
specifies the enter code to be used in generating a type-2 linkage.

mcrcd
specifies a code to be stored in the macro code field of the DCB.

CAUTION: Since the omission of both elements of the sublist in parameter three will result in no linkage being generated, you must not leave this field blank when using CHDINNRA to generate a type-2 linkage.

In addition to providing an enter code in parameter four, you must also provide a dummy entry point in parameter three.

CAUTION: The mcrcd parameter must only be used when the outer macro instruction has a DCB operand whose address is to be placed in register 1. Also, paraone and parazero may be used only when the value to be loaded into the appropriate register can be validly used as the second operand of an LA or LR instruction.

EXAMPLE A: The macro instruction

```
.LINK CHDINNRA, , (CZCXYZ),X'FF'
```

will result in the generation of either a type 1 linkage to CZCXYZ or a type-2 linkage to that routine with an enter code of 255 depending on the privilege class of the issuing module. This determination is made by testing the value in CHDCLS.

EXAMPLE B: The coding

```
EFORM CHDINNRA MF(2)
```

will result in the second element of the MF = operand being placed in register 1.

EXAMPLE C: The macro instruction

```
ERROR CHDINNRA , , (,254)
```

will result in the generation of SVC 254.

CHDERMAC -- Generate Error Message (nonstandard)

This inner macro instruction is used to generate error messages pertaining to errors encountered in macro expansions.

Name	Operation	Operand
	CHDERMAC	mesno-integer,[opnm-characters], [opva-characters] , [opvb-characters], [opvc-characters] [,S=integer]

mesno
: specifies a numerical code identifying the message to be generated. The codes and the messages issued by CHDERMAC can be found in Table 10.

opnm
: specifies the name of an outer macro instruction operand, or other information as desired.

opva, opvb, opvc
: specify operands of the outer macro instruction. A maximum of three operands can be specified in any one error message. These operands may also be used for such other purposes as the programmer may define.

S
: specifies the severity code associated with the error. A system default severity code exists and is shown in Table 9.

EXECUTION: For each value of the operand mesno, an MNOTE instruction is generated to produce an error message of this form

```
nnnnnn (B*sc+S) ***CHDmmm text
```

where (B*sc+S) is the severity code.

B	is set equal to zero if the S operand is present or to 1 if it is null.
sc	is the default severity code shown in Table 9.
S	is the severity code operand; it has a default of 0.
nnnnnn	is the six digit line number of the macro instruction for which the MNOTE is generated
mmm	is the error message number shown in Table 9.

The severity code algorithm, (B*sc S), is evaluated in the following manner. If you specify an S operand, B is set to 0 and the severity code is calculated as (0.sc) +S=S. If you omit the operand, B is assigned a value of 1 and S a value of 0. The algorithm then becomes (1.sc) +0=sc.

Table 9. Error Messages Issued by CHDERMAC (Part 1 of 2)

mesno	sc	mmm	Message Text
1	2	004	REQUIRED OPERAND(S) NOT SPECIFIED
2	2	001	FIRST OPERAND REQ'D-NOT SPECIFIED
3	2	001	SECOND OPERAND REQ'D-NOT SPECIFIED
4	2	001	THIRD OPERAND REQ'D-NOT SPECIFIED
5	2	001	FOURTH OPERAND REQ'D-NOT SPECIFIED
6	2	001	DCB OPERAND REQ'D-NOT SPECIFIED
7	2	001	DECB OPERAND REQ'D-NOT SPECIFIED
8	2	001	KEY OPERAND REQ'D-NOT SPECIFIED
9	2	001	FIFTH OPERAND REQ'D-NOT SPECIFIED
10	2	001	LOW. LIM. OPERAND REQ'D-NOT SPECIFIED
13	2	001	AREA OPERAND REQ'D-NOT SPECIFIED
14	2	001	LENGTH OPERAND REQ'D-NOT SPECIFIED
15	2	001	VALUE OPERAND REQ'D-NOT SPECIFIED
17	2	001	MODE OPERAND REQ'D-NOT SPECIFIED
18	2	001	REGISTER OPERAND REQ'D-NOT SPECIFIED
19	2	001	MESSAGE OPERAND REQ'D-NOT SPECIFIED
21	2	001	NAME OF DCB REQ'D-NOT SPECIFIED
22	2	001	NAME OF ADCON REQ'D-NOT SPECIFIED
23	2	001	NAME OF CSECT REQ'D-NOT SPECIFIED
24	2	001	NAME OF L FORM REQ'D-NOT SPECIFIED
25	2	001	TYPE OPERAND REQ'D-NOT SPECIFIED
28	2	001	CODE OPERAND REQ'D-NOT SPECIFIED
31	2	001	EP OR EPLOC OPERAND REQ'D-NOT SPECIFIED
35	2	002	INVALID MF OPERAND SPECIFIED-opva
36	2	002	INVALID FIRST OPERAND SPECIFIED-opva
37	2	002	INVALID SECOND OPERAND SPECIFIED-opva
38	2	002	INVALID THIRD OPERAND SPECIFIED-opva
39	2	002	INVALID FOURTH OPERAND SPECIFIED-opva
40	2	002	INVALID FIFTH OPERAND SPECIFIED-opva
42	2	002	INVALID EP OR EPLOC OPERAND SPECIFIED-opva
44	2	002	INVALID LENGTH OPERAND SPECIFIED-opva
45	2	002	INVALID MODE OPERAND SPECIFIED-opva
46	2	002	INVALID REG(S) OPERAND SPECIFIED-opva
47	2	002	INVALID AREA OPERAND SPECIFIED-opva
48	2	002	invalid type operand specified-opva
49	2	002	INVALID OPTION OPERAND SPECIFIED-opva
50	2	002	INVALID OPTION 1 OPERAND SPECIFIED-opva
51	2	002	INVALID OPTION 2 OPERAND SPECIFIED-opva
54	2	002	INVALID KEYWORD OPERAND SPECIFIED-opva
55	2	002	INVALID REGISTER NOTATION SPECIFIED-opva
56	1	025	PACK OPERAND NOT ALLOWED W/MODE=R
57	2	002	INVALID PR OPERAND SPECIFIED-opva
58	2	002	INVALID PACK OPERAND SPECIFIED-opva
59	2	002	LV OPERAND REQ'D-NOT SPECIFIED
62	1	067	ADCOND MACRO PREVIOUSLY SPECIFIED
63	2	002	INVALID TAM CHARACTER CODE OPERAND SPECIFIED-opva
69	2	006	REGISTER NOTATION INVALID W/MF=L
78	0	024	CSECT NAME BLANK. MACRO NAME OMITTED.
85	1	013	MESSAGE OPERAND NOT ALLOWED W/MF=E
86	1	013	OPLIST OPERAND NOT ALLOWED W/MF=E
87	2	014	DECB NOT SPECIFIED AS SYMBOL
88	1	015	MORE THAN ONE OF EP OR EPLOC PRESENT
89	0	050	opnm OPERAND INCONSISTENT WITH TYPE=opvaopvb
90	2	050	opnm INCONSISTENT W/opva OPERAND
147	0	050	opnm OPERAND INCONSISTENT-IGNORED
157	1	051	INVALID CODE FOR opnm-IGNORED-opva
159	1	053	INVALID CODE FOR DSORG-IGNORED-opva
162	1	056	MACRF INVALID WITH SPECIFIED DSORG-IGNORED-opva
163	1	056	EXLST INVALID WITH SPECIFIED DSORG-IGNORED-opva

(Continued)

Table 9. Error Messages Issued by CHDERMAC (Part 2 of 2)

mesno	sc	mmm	Message Text
166	1	060	INVALID CODE FOR DEVD WITH SPECIFIED DSORG-IGNORED-opva
167	1	065	MACRF INVALID-IGNORED-opva
169	1	067	DCBD MACRO PREVICUSLY USED
173	1	062	DDNAME LCNG-TRUNCATED TO 8 CHAR
174	1	070	devd=opvb IGNORES opnm=opva
175	1	071	INVALID opnm OPERAND SPECIFIED-IGNORED-opva
176	1	072	MULTIPLE DEVICE-DEP. PARAM. 1 SPECIFIED-IGNORED-opva=opvb
177	1	073	MULTIPLE DEVICE-DEP. PARAM. 2 SPECIFIED-IGNORED TRTCH=opva
178	0	074	PAD OPERAND GT 50-SPECIFIED VALUE USED-opva
179	0	101	CSECT ORIGIN USED FOR opnm RCCN
180	1		opnm OPERAND INVALID OR NOT SPECIFIED-SET TO opva
181	1	076	BPY CNTR INDICATES WRAP AROUND TC TCP CF CRT
182	1	077	BLC GREATER THAN OR EQUAL TO BLIM
183	1	002	OPNM INVALID-SET TC opva
184	*	078	* CURRENT BUFFER opnm=opva
185	*	079	* CURRENT BEAM POSITION CCUNTER IS X=opnm, Y-opva
186	1	080	opnm COUNTER EXCEEDS CRT LIMITS
187	1	081	LCAD VARIABLE SPACE ORDER MAY NCT HAVE BEEN SPECIFIED PRICR TO ENTERING STRCKE MODE
188	2	103	opnm MACRO NOT ALLOWED FOR PRIVILEGED USER
200	1	101	ZERO USED FOR opnm RCON
201	1	075	VAR OPERAND NOT ALLOWED W/MODE=R
210	2	001	opnm OPERAND REQ'D-NCT SPECIFIED
211	2	002	invalid opnm OPERAND SPECIFIED-opva

In general, you should attempt to continue processing a macro expansion after detecting an error and generating a message. However, although it is difficult to generalize, some errors should cause termination of processing. An example is an invalid MF operand in an S-type macro instruction, which makes further processing impossible. Another instance is the occurrance of an error that propogates other errors, thereby using up valuable time.

The termination of processing implies that you know the user has made a mistake and cannot continue. It is better to give him the benefit of the doubt and let him continue. When his program fails, it will fail in his code and won't attempt to blame his problem on you.

CHDPSECT -- Reserve Storage for Parameter List (nonstandard)

The CHDPSECT macro instruction establishes the next available location in the user's PSECT as the location at which the parameter list will be located. If no PSECT exists when the macro instruction is being assembled, then the next available location in the current control section is used, and CHDPSECT generates a branch around the list.

Name	Cperation	Operand
[symbol]	CHDPSECT	[loc-addr] , [align- {OF / OH}] [,string- text]

symbol
is the symbolic location of the first byte to be assigned to the parameter list.

loc
specifies the location to which the branch instruction is to transfer control. If this operand is omitted, CHDPSECT establishes its own branch address. All symbols generated by the macro instruction must be of the form CHD [X] &SYSNDX where x is an optional letter used to distinguish symbols when more than one must be generated. x must be unique for each symbol.

align
specifies the alignment you wish for the beginning of the parameter list.

0F - specifies alignment on a full word boundry.
0H - specifies alignment on a half word

string
specifies a character string, originally specified as an operand in the outer macro instruction, which is to be placed, as is, in the parameter list. When the character string is used as an operand, the CHDPSECT inner macro instruction generates X'27' to indicate end of string.

PROGRAMMING NOTES: When the string operand is not specified, you must specify the address for the branch instruction in the loc operand.

The use of a CNOP to force alignment will be generally ineffective since the parameter list may be generated in another control sectiCn (the PSECT). Placing the CNOP instruction before CHDPSECT will have no effect other than to align the macro instruction.

SECTION 5: GENERATING AND MAINTAINING TSS/360

System programmers are responsible for generating the specific version of TSS/360 used at each installation; and, they are responsible for troubleshooting and maintaining that system once it is generated. Maintenance involves analysis of any system problems that occur, design of installation required changes (addition, deletions, etc), and the incorporation of IBM-issued changes applicable to the installation.

SYSTEM GENERATION

The system generation process consists basically of reassembling and replacing system modules containing configuration-dependent tables and other installation-option parameters. System generation macro instructions are used to control this operation. These macro instructions, as well as the sysgen process, are described in detail in System Generation and Maintenance.

SERVICEABILITY AIDS

You have the following facilities for monitoring system performance and for analyzing sources of system errors once your system has been generated:

- SYSER Dump: A system output which provides you with information regarding system failure, and may enable you to pinpoint the source of trouble in the system.

- Program Checkout Subsystem (PCS): A set of commands that enable you to locate problem sources in nonprivileged, virtual storage programs. PCS also provides similar, but restricted, facilities for troubleshooting privileged virtual storage programs. (In the initial release, no PCS facilities are available for resident programs).

SYSER DUMP

The supervisor will provide you with a dump, whenever it encounters an error, by issuing the ERROR macro instruction. Dumps are also provided by privileged programs; these are supplied by means of the SYSER macro instruction. Both of these macro instructions produce basically the same output.

When an ERROR macro instruction is issued, a message is printed at the operator's console in this format:

```
CEAIS SYSERR CODE RSC mmnne t userid volid
```

where

```
     mm = module ID assigned to the module issuing the call (see Table
          7 for a list of these codes)
     nn = indicates the specific error which caused ERROR to be issued
      e = error type code
      t = time at which the error occurred
 userid = user ID of the task in which error occurred
  volid = volume identification of the output tape
```

ERROR also causes the construction of one of these header records in the output buffer:

RSC mmnnl t MINOR SYSTEM SOFTWARE ERROR userid (for error types 1 or 9)

RSC mmnn 2 t MAJOR SYSTEM SOFTWARE ERROR USERID=userid (for error type 2)

RSC mmnn 3 t SYSTEM HARDWARE ERROR USERID=userid (for error types 3 or 7)

where mm, nn, t, and userid are as defined above.

When a SYSER macro instruction is issued, a message is printed at the operator's console in this format:

CEAIS SYSERR CODE vvccsssnn t userid volid

where

vv
: a unique, two-digit identifier (see Appendix E, opt_1)

cc
: a unique, two-digit identifier (see Appendix E, opt_2)

sss
: a unique, three-digit identifier (see Appendix E, opt_3)

nn
: indicates a specific error condition in the module

t
: time at which the error occurred

userid
: user ID of the CPU's current task

volid
: volume ID of the output tape

SYSER also constructs a header record in one of these formats:

vvccsssnn t MINOR SYSTEM SOFTWARE ERROR userid (for error types 1 or 9)

vvccsssnn t MAJOR SYSTEM SOFTWARE ERROR USERID=userid (for error type 2)

vvccsssnn t SYSTEM HARDWARE ERROR userid (for error types 3 or 7)

After writing the message and constructing the header record, SYSER and ERROR proceed to fill the output buffers with the specified information. (Refer to Table 6 for a list of the options available.) As the buffers become filled, they will be written to the output tape to form the SYSER dump. This tape is nine track with standard labels.

Retrieving Your Dump: When it becomes necessary to diagnose system or task errors, you will want to retrieve the dump data set and print it on the system's high speed printer. You can do this by issuing the DEFINE DATA and PRINT commands described below.

Operation	Operand
DEFINE DATA DDEF	ddname-alphname,dsorg-PS,DSNAME=SYSTEM.ERROR.DUMP, UNIT=(TA,9),VOL=(,volserno-alphnum), LABEL=(,SL),DISP=(OLD)

ddname
specifies the symbolic data definition name associated with this data set definition. It provides the link between the DCB in the print program and the data set definition. It must contain from one to eight alphameric characters, the first of which must be alphabetic. The ddname may not begin with SYS since system reserved names are prefixed with those characters.

dsorg
specifies PS because the data set is on magnetic tape and has a BSAM organization.

DSNAME
specifies SYSTEM.ERROR.DUMP, the name of data set to be printed. Since the name has been specified by the system, no choices are available.

UNIT
specifies that the dump is on nine track tape

VOL
specifies the volume serial number of the tape on which the dump is recorded

LABEL
specifies that standard labels (SL) are used

DISP
specifies that the data set already exists (CLD). This parameter must be included since the default option is NEW.

After defining your data set, you can obtain its printout by issuing the following PRINT command.

Operation	Operand
{PRINT PR}	SYSTEM.ERROR.DUMP,, [{spacing-{1 2 3}, [H], [lines-integer], [P]}],, [error-{ACCEPT SKIP END}], [form-specsym]

SYSTEM.ERROR.DUMP
specifies the data set you want printed. You have no option in this case.

spacing
specifies the number of spaces you want between lines and may be one, two, or three. If you omit this parameter your listing will be single spaced. Since the system error processor constructs the data set without control characters, you cannot specify EDIT.

H
specifies that you want the first logical record in the data set used as a header line on each page. As you recall, this first logical record is supplied by SYSER or ERRCR processing and serves to identify the dump that follows. You might find this header misleading for all PAGES but the first; if so, omit the parameter.

lines
specifies the number of lines you want printed on each page. It may consist of one or two decimal digits, but two digits are preferred and 54 lines are standard.

P
specifies that you want the listing pages numbered.

error
specifies the action you want taken when an uncorrectable error occurs while reading the data set. You might choose to:

ACCEPT - the error record,
SKIP - the error record, or
END - the print operation.
This last option is not only the default option but possibly the least desirable. Since you are in the process of diagnosing past errors, it is doubtful that you would want your diagnostic tool destroyed by the occurrence of another error. If this is the case, don't omit this parameter.

form
specifies the form number of the paper you want used. Omission of this parameter indicates that you want the installation's standard form.

The output you receive will depend on the option that was specified in the macro instruction. In general, storage locations are printed eight words to a line. Each line is preceeded by the address of the first byte in the line and each word consists of eight hexadecimal digits.

PROGRAM CHECKOUT SUBSYSTEM (PCS)

The eight PCS commands and their function are discussed in detail in the Command Language User's Guide. The following discussion covers the precautions you must observe as a system programmer while using PCS commands because of the limitations imposed on their use by the action of the dynamic loader.

Each D class user is assigned an authority code at JOIN time. Code 'P' specifies system programmer, code 'C' specifies a master system programmer, and code 'U' specifies an ordinary user. When a user logs-on, this authority code is used to govern the operation of the dynamic loader and his use of PCS.

The dynamic loader ignores or overrides control section attributes depending on the programmer's authority code and the library from which the module is loaded.

If you are a system programmer with authority code P, you can checkout nonprivileged system programs. These programs can be dynamically loaded from any one of the three major libraries. If the program is loaded from either JOBLIB or USERLIB, they are assigned to private, read/write storage. The attributes of public, read-only, system, or privileged are overridden. If it is loaded from SYSLIB, only the public and read-only attributes are overridden. As a result, you will get a

private copy of any module dynamically loaded from SYSLIB. Nonprivileged modules so loaded will be assigned write/fetch protected. This provides continued protection for the privileged routine.

This action of the dynamic loader on modules you invoke as a user with authority code P has the following impact on your use of PCS commands:

1. You may utilize all PCS commands in testing your nonprivileged programs,

2. You may utilize symbolic addressing to display or dump any privileged CSECTs which have been dynamically loaded,

3. You may display or dump the contents of virtual memory.

If you are a master system programmer (authority code 0), any module you dynamically load will be assigned to private read/write storage. The attributes of public, read-only, system, and privileged are overridden by the dynamic loader.

Your PCS capabilities with respect to nonprivileged programs are the same as they are for a nonprivileged system program. In addition, you can display,dump, or set IVM. You must exercise extreme caution in setting IVM, particularly in a multi-CPU environment, since other CPUs may be accessing the code you are setting.

You cannot access other shared codes in the system for PCS testing, since it is not part of your virtual memory.

You will also find that your ability to check dynamically loaded privileged modules is quite limited due to several factors. Primarily, the LOAD and RUN commands will not accept module names beginning with CHB or CZ. These are the prefixes of the privileged routines.

You can, of course, load one of these modules under an alias. The RUN command, however, activates with a nonprivileged PSW key of 1. If you do manage to load the module under an alias, it will manage to run until it attempts to access or transfer control to another privileged module. At that point, the disparity in PSW keys will result in a program interrupt.

PCS operates with nonprivileged save area one. As a result you cannot display, dump, or set registers or PSW information relating to a program executing in privileged mode.

SYSTEM MAINTENANCE

You should not attempt modification of TSS/360 unless you are an experienced system programmer who has a thorough knowledge of the system's internal specifications (in particular of the interfaces involved in each modification). Detailed information about system modification may be found in System Generation and Maintenance.

In designing local changes to TSS/360 you will follow a procedure something like this:

1. Define the function to be accomplished.

2. Identify the modules to be added, amended, or deleted.

3. Define the interface of these modules with all other TSS/360 modules. The control section dictionaries of modules currently in the system provide you with a listing of all the module's REFs and

DEFs. This is a start in determining, for example, how an existing module (that is to be changed) is currently "plugged into" the system. However, care must be exercised, as this information may be deceptive. For example, an external address can be loaded into a register, and the register (instead of the external address) can subsequently be referenced in the program. You might see this,

```
BOLD     L       5,=V(CHBSYS)    SYSTEM TABLE ADDRESS
         USING   CHASYS,5        FORMAT OF SYSTEM TABLE
SNEAKY   L       6,SYSOCT        EXTERNAL REFERENCE
         L       7,60(5)         EXTERNAL REFERENCE
```

The symbol CHBSYS is an external symbol and would appear in the control section's dictionary as an external reference (REF). The reference to SYSOCT would not appear as an external reference, though, and the reference 60(5) isn't even a symbolic reference. The cross reference dictionary would show you that statement BOLD references the externally defined symbol CHBSYS; you have to figure out that its also referenced by SNEAKY.

Unfortunately, there's no convenient way to determine what programs actually do reference the external symbols defined in a given program. The instruction,

```
ENTRY    ABCRJG
```

allows other programs to reference the symbol ABCRJG. There is no guarantee, however, that any other programs will actually reference ABCRJG. Consequently, if you delete a program from TSS/360, you have no systematic way to determine which programs reference cr make use of the program you're deleting. You can determine such referencing only by carefully studying the function of the program being modified or replaced, and by understanding its role in the overall design you're trying to change.

You might be tempted to list all the external symbol dictionaries of all the program modules that make up TSS/360, as a way of determining their respective interdependencies. Although this might prove helpful, it is not foolproof. Some programs set up registers with external addresses for use by other programs that know what the registers are supposed to contain. A program using registers set up by another program might not contain a single explicit external reference. You might see this,

```
OBVIOUS  L       6,=V(CHBSYS)    LOAD EXTERNAL SYMBOL
         L       15,=V(SNEAKY)   LOAD ADDR OF SUBROUTINE
         BASR    14,15           TRANSFER
```

The subroutine might look like this,

```
SNEAKY   USING   *,15            DECLARE BASE
         L       8,12(6)         HIDDEN EXTERNAL REFERENCE
```

The external reference to CHBSYS would never show up in the external dictionary of the program module containing SNEAKY. You must, therefore, beware of this situation, as you will encounter it often.

4. Write the necessary assembler statements.

5. Assemble and test the new or amended modules and store them in the same library.

6. Update the TSS/360 system data sets using the system-edit control statements and procedures described in System Generation and Maintenance.

As a system programmer, you may be joined to the system with combined privilege classes D and E; each class is associated with a particular set of facilities that is available for your use.

The assignment of privilege class D (along with your authority code of P or S) designates you as a system programmer. This privilege class provides you with the facilities described in Assembler User Macro Instructions and Command Language User's Guide; in conjunction with your authority code, class D also provides you with the facilities discussed earlier in this publication.

The assignment of privilege class E, which designates you as a system monitor, extends the range of facilities available to you. Through certain options that only the privilege class E programmer can use in the DATA DEFINITION (DDEF) command and macro instruction, and in the DCB macro instruction, you can designate specific I/O devices and directly utilize unit record equipment. It also provides you with the ability to use (for system routines) the multiple sequential access method (MSAM) and the terminal access method (TAM), denied to ordinary users and to system programmers who have not been assigned privilege class E in addition to their privilege class D. It also provides you with the ability -- denied to privilege class D programmers -- to directly control unit record equipment when using the basic sequential access method (BSAM).

DESIGNATING I/O EQUIPMENT

When you have been joined with privilege class E, you have several options in the operand field of the DATA DEFINITION (DDEF) command and macro instruction, and DCB macro instruction, that are not shown in the detailed descriptions in Command Language User's Guide and Assembler User Macro Instructions. Except for these options, which will be described in detail here, the parameters you may use are those shown in Appendix G of each of those publications.

SYMBOLIC DEVICE ADDRESS

One of the options available to you, as a system programmer with privilege class E, is to designate the I/O device you want to use by its symbolic address. This can be accomplished by entering

```
,UNIT=(SDA=code)
```

in the operand field of the DATA DEFINITION (DDEF) command or macro instruction, where "code" is a one-to-four-hexadecimal-digit symbol (from 1 to 1FFF) assigned at system generation to the desired I/O unit as its symbolic address. By choosing this option, you can designate a particular terminal (for TAM programming), a particular unit record device (for MSAM, BSAM, or IOREQ programming), or a particular tape drive or direct access I/O device (for BSAM, IOREQ, or VAM programming).

DESIGNATING DEVICES FOR MSAM

In addition to SDA=integer, three other codes may be used with the UNIT keyword parameter of the DDEF command and macro instruction, when using MSAM. You may write

```
        UNIT= {SDA=code}
              {PC      }
              {PR      }
              {RD      }
```

where SDA= is followed by the symbolic device address of the desired unit record device, PC is a card punch, PR is a printer, and RD is a card reader. If you use the multiple sequential access method, one of these options must be specified.

When using MSAM, you must specify the code PS (physical sequential) for the positional data set organization (dsorg) parameter of the DDEF command and macro instruction, and the code MS (multiple sequential) for the keyword data set organization (DSORG=) parameter of the DCB macro instruction.

DESIGNATING DEVICES FOR TAM

When using TAM, you must denote the terminal with which you want to communicate, by means of its symbolic device address. Hence, the only permissible entry in the UNIT subparameter field of the DDEF command and macro instruction is

```
,UNIT=(SDA=code)
```

In addition, when using TAM, you must specify the code CX in the second positional parameter (dsorg) of the DDEF command and macro instruction. This code must also be used with the DSORG keyword parameter of the DCB subparameter list of the DDEF command and macro instruction and the DCB macro instruction.

CONTROLLING I/O DEVICES FOR BSAM

In addition to those detailed in Assembler User Macro Instructions, two macro instructions are available to you as a privilege class E programmer when you are using the basic sequential access method (BSAM). These macro instructions provide you with the ability to exercise greater control over the I/O devices you are using.

CNTRL -- Control On-Line Input/Output Devices (R)

The CNTRL macro instruction is used to perform operations on magnetic-tape drives and on-line card readers and printers in which data is not transferred. The following functions are provided: magnetic-tape positioning, card-reader stacker selection, and printer carriage control.

Name	Operation	Operand
[symbol]	CNTRL	{dcb-addrx | (1)}, {action-code [,number-value] | (0)}

dcb
: specifies the address of the data control block opened for the data set being processed. If you write (1), the address must be loaded into general register 1 before execution of this macro instruction.

action
: specifies, by a code, the service to be performed:

SS
causes a stacker to be selected for a card reader (stacker 1 or 2).

SP
causes a line space on a printer, space 1, 2, or 3 lines.

SK
causes a skip on the carriage-control tape for a printer skip to channels 1 through 12.

BSR
causes a backspace over a specified number of blocks on magnetic tape; one block is assumed if the number operand is omitted; BR is the abbreviated code.

BSM
causes a backward motion past a magnetic-tape mark and a forward space over the tape mark; a number value of 1 is assumed; BM is the abbreviated code.

FSR
causes a forward space over a specified number of blocks on magnetic tape; one block is assumed if the number operand is omitted; FR is the abbreviated code.

FSM
causes forward motion past a magnetic-tape mark and a backspace over the tape mark; a number value of 1 is assumed; FM is the abbreviated code.

FSF
causes forward motion past a magnetic-tape mark; a number value of 1 is assumed; FF is the abbreviated code.

BSF
causes backward motion past a magnetic-tape mark; a number value of 1 is assumed; BF is the abbreviated code.

WTM
causes a tape mark to be written on magnetic tape; a number value of 1 is assumed; WM is the abbreviated code.

REW
rewinds magnetic tape; RW is the abbreviated code.

RUN
rewinds and unloads magnetic tape; RU is the abbreviated code.

ERG
causes an erase gap to be executed for magnetic tape; ER is the abbreviated code.

If you write (0), the two-character action code must be placed in the two high-order bytes of general register 0 before execution of this macro instruction. In the case of three-character action codes, the abbreviated code must be placed in those bytes.

number
specifies a value for the stacker number, number of lines to be skipped on the printer, printer carriage-tape channel, or number of blocks on magnetic tape to qualify the action operand. The maximum value is 32,767. If you write (0), the value must be placed in the

two low-order bytes of general register 0; value is a binary integer.

CAUTION: If magnetic-tape positioning is performed, an uncorrectable tape-spacing error results in linkage to the user's SYNAD routine; this does not apply to action codes SS, SP, SK, REW, or RUN. See Appendix B of Assembler User Macro Instructions for a description of SYNAD.

Abnormal termination occurs if:

1. Action code is undefined or not applicable.
2. Number parameter is undefined for the action parameter.
3. A SYNAD-type error occurs and you have not provided a SYNAD address.
4. The specified data control block has not been validly opened.
5. The outstanding read or write operations have not been checked.

PROGRAMMING NOTES: For stacker selection, the DCBNCP field of the data control block must be 1. Each READ macro instruction directed to a card reader must be followed by a CHECK macro instruction and a stacker selection CNTR macro instruction directed to the same device. Stacker selection is not available for the card punch except through changing, in your program, the DCBSTA field in the data control block.

You must check READ and WRITE operations for completion before issuing the CNTRL macro instruction. If you are using the macro instruction to control stacker selection, you must issue it for each read operation except the last. The CNTRL macro instruction must not be issued for the last read operation (i.e., the READ macro instruction which, when checked, invokes EODAD) since no card was read.

For printers, a skip to a given channel results in no action if the device is already at that channel.

Control is returned to you if a tape mark or a load point is encountered while an attempt is being made to forward space or backspace blocks (control is not given to the SYNAD routine). Register 15 contains binary 0s if the operation is completed normally; otherwise, it contains a count of the remaining number of forward spaces or backspaces that were not completed in its low-order two bytes.

NOTE: The CNTRL macro instruction may also be used by a class D programmer for magnetic-tape positioning. However, use of the CNTRL macro instruction for card-reader stacker selection and for printer carriage control is restricted to class E programmers.

PRTOV -- Test for Printer Carriage Overflow (R)

The PRTOV macro instruction is used to control the page format for an on-line printer. As a privilege class E programmer, you can test channel 9 or 12 of the printer control tape to determine if an overflow condition exists.

Before testing overflow indicators, PRTCV waits for completion of all previously requested printing.

Name	Operation	Operand
[symbol]	PRTOV	{dcb-addrx / (1)}, number-{9 | 12} [, {userrtn-addrx / (0)}]

dcb
specifies the address of the data control block opened for the data set being processed. If you write (1), the data control block address must be loaded into general register 1 before execution of this macro instruction.

number
specifies either 9 or 12 as the channel to be tested for an overflow condition.

userrtn
specifies the address of a routine which is to be given control if the appropriate program indicator (for channel 9 or 12) is on when tested. If you write (0), the address must be loaded into general register 0 before execution of this macro instruction. If you omit this operand, and if the overflow condition exists, an automatic skip to channel 1 will be performed prior to the next WRITE operation.

CAUTION: Abnormal termination occurs if the data control block you have specified is not validly opened.

PROGRAMMING NOTES: This macro instruction causes no action if used for a device other than a printer.

If a WRITE macro instruction is directed to the printer and a CHECK macro instruction is not issued to verify its execution, the channel overflow indicator may not have been set to produce the desired results when the PRTOV macro instruction is issued.

If the user routine includes a PSECT, it must be the same PSECT as the routine that issues the PRTOV macro instruction. To continue processing at the point where the PRTOV macro instruction was issued, the user routine must branch to the address that was contained in general register 14 upon entry to the user routine. A RETURN macro instruction cannot be used for this purpose.

If no user routine is specified, execution of the problem program continues after a PRTOV macro instruction is issued. When the line associated with the first WRITE macro instruction issued after the PRTOV is to be printed, the appropriate program indicator is tested. An automatic skip to channel 1 is performed if an overflow has occurred.

If a user routine is specified, the control program waits after a PRTOV macro instruction is issued. When all prior print operations are complete, the appropriate program indicator is tested.

The contents of the general registers upon entry to the user's overflow routine are:

Register	Contents
0	Unspecified
1	Address of data control block
2 to 13	Same as before macro instruction was executed
14	Return address
15	Address of userrtn routine

EXAMPLES: In EX1, an overflow condition on channel 9 of the printer-control tape results in an automatic skip to channel 1 since the operand, userrtn, is omitted. In EX2, an overflow condition on channel 12 results in control being given to the user's overflow routine.

```
EX1     PRTOV   OUTDCB,9
EX2     PRTOV   PRINTDCB,12,OVERFLOW
```

MULTIPLE SEQUENTIAL ACCESS METHOD (MSAM)

The multiple sequential access method, MSAM, provides a fast and efficient mechanism for simultaneously driving, under the control of a single task, several card readers, card punches, and printers. The access method's macro instructions provide automatic buffering and automatic error retry options.

GENERAL DESCRIPTION

MSAM will support both fixed (F) and variable (V) format records. MSAM routines buffer logical records into system-provided buffers, each of which resides in a separate page of virtual memory. The basic user interface to MSAM is the GET, PUT, OPEN, CLOSE, SETUR, and FINISH macro instructions.

MSAM also provides you with the capability of efficiently processing data on multiple unit record devices within one task. While this is possible within other TSS/360 access methods, MSAM alone has defined its user-interface (macro instructions) in such a manner that the system service routines need not end the task's time-slice while waiting for the occurrence of an event, such as I/O completion. This efficient device utilization is accomplished by defining macro instructions which provide a return code to inform the invoking routine that a delay is necessary before the request, such as GET, PUT, or FINISH, can be completed. This transference of the responsibility of waiting, from the control program to the invoking routine, provides the ability for the task to process all its opened DCBs until all DCBs accessed require waiting. Then the task may wait for the first I/O interrupt for any DCB in the task.

MSAM also differs from other sequential access methods in that each MSAM I/O request of the system processes a buffer group of physical records. In the other methods, each I/O request of the system is for only one physical record. Considerable processing is required in IOS and the access mechanism for each I/O request of the system, regardless of buffer size. Usually, MSAM will invoke an I/O request only once for processing each buffered page. However, some buffers, such as the last buffer in a data set, may contain fewer records.

DCB OPTIONS

You can reference the information stored in the data control block by means of the DCBD macro instruction, which is described in Assembler User Macro Instructions. The options available to you may be selected by correctly filling in the DCB fields described below. The sources of this information are given in Table 10.

Table 10. Sources of DCB Information for MSAM

DCB Field	Alternate Sources			
	Your program prior to OPEN	DDEF command and macro instruction	DCB macro instruction	Your program after OPEN
DSORG	X			
MACRF	X	X	X	
DDNAME	X		X	
DEVD	X	X	X	
PRTSP[1]	X	X	X	X[2]
MODE[3]	X	X	X	X[2]
STACK[3]	X	X	X	X[2]
RECFM	X	X	X	X[2]
LRECL	X	X	X	X[4]
POCKET	X			X[2]
RETRY	X			X[2]
SUR				X[5]
INHMSG	X	X	X	X
FIP				X[6]
COMBINE	X			
FORMTYPE	X			X[2]

[1]Only checked if DEVD specifies a printer (PR).

[2]Only if a FINISH macro instruction has been executed and a return code other than 4 was provided, and if no GET or PUT macro instruction has been executed after the FINISH.

[3]Only checked if DEVD specifies a card punch (PC) or a card reader (RD).

[4]For format-F records, footnote 2 applies.

[5]Only if a SETUR macro instruction has been executed and a return code of 4 was provided.

[6]Only if a FINISH macro instruction has been executed and a return code of 4 was provided.

DSORG
must be set to indicate MS.

MACRF
must specify only GET or PUT macro instruction (all other combinations will cause abnormal termination of the task in OPEN).

DDNAME
must be three to eight alphameric characters.

DEVD
must be PR, PC, or RD.

PRTSP
specifies the line spacing as 0, 1, 2, or 3 after printing. The PRTSP field will be ignored if (A/M) is specified in the DCB RECFM field. PRTSP may be used but cannot vary between each PUT. If the field is not supplied in the DCB at OPEN time, one line of spacing is assumed. If neither A nor M was specified in the DCB, and channel 12 is sensed in the carriage control tape, an automatic skip to channel 1 is performed by the system.

MODE

must be C (column binary mode) or E (EBCDIC mode). The value of the MODE field may not be modified after the DCB is opened except between a completed FINISH and the next GET or PUT macro instruction. A binary value of 1 in this field specifies column binary and binary value of 0 specifies EBCDIC. If the mode is not specified, EBCDIC is assumed.

STACK

specifies the stacker bin (1, 2, or 3); stacker bin 3 may be specified only if the punch (PC) is specified. The STACK field is ignored if (A/M) was specified in RECFM at OPEN time. STACK may not be changed for each GET or PUT to a card punch so that the stacking of each card varies. If the field is not supplied, stacker bin 1 is assumed.

RECFM

specifies the characteristics of the records in the data set. Although any of the following order are valid

(F/V) [B] [A/M]

B will be ignored. Any other record format designations will cause OPEN to abnormally terminate the task.

Control Characters [A/M]: As an optional feature, records may include a control character in each logical record. This control character will be recognized and processed if a data set is being written to a printer or punch by MSAM. This character is provided by the user as the first byte of the logical record. Two options, A and M, are available. If either option is specified in the data control block, the character must appear in every record.

Machine Code (M) - The user may specify in the data control block that the machine code control character has been placed in each logical record. The byte supplied by the user must contain the bit configuration specifying a write and the desired carriage- or stacker-select operation (this permits independent carriage- and stacker-select operations).

Extended USASI Code (A) - The user may choose to specify this code rather than the machine code; the control byte must appear in each logical record if this option is chosen.

LRECL

For format-F records, LRECL specifies the length in bytes. This length must include the control character for an output data set if (A/M) is specified in RECFM. LRECL may not exceed 80 bytes for reading in EBCDIC mode, and 160 bytes for column binary mode. For an output data set, on a printer, the maximum is 133 bytes; on a card punch, 81 bytes for EBCDIC and 161 bytes for column binary - the additional byte for output data sets is for the control byte only if (A/M) is specified in the RECFM.

For format-V records, LRECL specifies the maximum length in bytes of a logical record. LRECL may be modified after the DCB is opened at any time. LRECL for an input data set may not exceed 84 bytes for reading in EBCDIC mode and 164 bytes for column binary mode. For an input data set, on a printer, the maximum is 137 bytes; for a card punch, 85 bytes for a EBCDIC and 165 bytes for column binary. The additional byte for output data sets is for the control byte only if (A/M) is specified in the RECFM. The four or five control bytes of a format-V logical record are not punched or printed. A format-V logical record will have five control bytes when a USASI or machine control character is specified.

BLKSIZE
this field is not referenced by MSAM routines, as only unit record equipment is supported. Unit records (card images or print lines) are considered to be unblocked. Unblocked is defined as one logical record recorded on one physical record.

EODAD
not referenced by MSAM routines.

SYNAD
not referenced by MSAM routines.

POCKET
a one-byte field to indicate error card stacker bin (for card reader on 2540 only). Options are CRG, 1, or 2. ORG means stack as if no error occurred, 1 means stacker bin 1, and 2 means stacker bin 2.

RETRY
a one-byte field to indicate data check error retry for 2540 reader. The options are N or U, where N indicates no retry and U indicates unlimited retry.

SUR
this bit will be set to 1 by the SETUR routine when the SETUR macro instruction is issued. The bit is set to 0 when the SETUR routine returns any return code other than 4. Thus the bit indicates that the unit record setup is in progress. If a problem program has issued a SETUR macro instruction and received a return code of 4 and if, for any reason (such as an asynchronous operator message), the problem program wants to suppress the completion of the SETUR, it may do so by setting this bit "DCBSUR" to 0 and reissuing the SETUR macro instruction.

INHMSG
this bit should be set to 1 if the problem program wants to inhibit messages to the operator, to remove the data group when a CLOSE or FINISH macro instruction is issued. If this bit is set to 0, and CLOSE or FINISH is issued, a message will be sent to the operator to remove the data group from the device. When the operation indicated by the WTO is complete, the operator will generate an asynchronous interruption by changing the status of the device from "not ready" to "ready". This bit "DCBINH" may be referenced by users of the DCBD macro instruction.

FIP
this bit will be set to 1 when the FINISH macro instruction is issued. The bit is set to 0 when the FINISH routine returns any return code other than 4. Thus, the bit indicates that FINISH is in progress. If a problem program has issued a FINISH macro instruction and received a return code of 4, and for any reason (such as an asynchronous operator message) the problem program wants to suppress the completion of FINISH, it may do so by setting this bit "DCBFIP" to 0 and reissuing the FINISH macro instruction.

COMBINE
if this bit is set to 1, both the card reader and the card punch on the same 2540 will be assigned to the output operation. Each time a FINISH or a CLOSE macro instruction is issued, a card will be read from the reader and stacked in pocket 3.

FORMTYPE
specifies the print error retry. Normally, this parameter is provided by SETUR's SYSURS specification. There are three options: D, F, and S. The D, or dump-type, option specifies that after

retry the line in error is to be struck out, one line space is to be skipped, and the line is to be rewritten.

The F, or form-sensitive, option specifies that after retry the entire page containing a line in error is to be reprinted; control characters must be in use and at least three buffers must be available. The S, or sequence-sensitive, option specifies that after retry printing is to be continued with the next line after the line in error. If SETUR is not called, the default value for the option is D.

DDEF COMMAND AND MACRO INSTRUCTION

The general format of the DEFINE DATA (DDEF) command and macro instruction are given in Appendix G of Command Language User's Guide and Appendix G of Assembler User Macro Instructions, respectively. When you are using MSAM, you must specify the code PS as the dsorg positional parameter. In addition, you can specify SDA=code, PC, PR, or RD with the UNIT keyword parameter, as described in detail earlier under "Designating I/O Equipment," and the DISP keyword parameter must be OLD when using the card reader, and NEW when using the card punch or printer.

GENERAL SERVICE MACRO INSTRUCTIONS

Two general service macro instructions, OPEN and CLOSE, are available for use with MSAM routines.

OPEN -- Prepare the Data Control Block for Processing (S)

The OPEN macro instruction initializes one or more data control blocks for processing of their associated data sets. During the execution of OPEN, the user's DCB exit routine is given control if supplied by the user.

Name	Operation	Operand
[symbol]	OPEN	({ dcb-addr,[(opt-code)],}...)

dcb
specifies the address of the data control block to be initialized.

opt
specifies the intended method of input/output processing of the associated data set. The codes and their meaning are:

INPUT Input data set; this value is assumed if opt is omitted.

OUTPUT Output data set.

CAUTION: The following errors cause the results indicated:

Error	Result
Opening data control block that is already open	No action
Specifying address of invalid data control block	Task terminated
Opening data control block when DDNAME has not been provided	Task terminated
Opening data control block when corresponding DDEF macro instruction or command has not been provided	Task terminated (prompting will be given if task is conversational)
Opening data control block containing invalid DSORG specification	Task terminated

PROGRAMMING NOTES: You may specify any number of data control block addresses and associated options in the OPEN macro instruction. This facility allows parallel opening of the data control blocks and their associated data sets.

OPEN initializes all the fields in the MSAM portion of the DCB, as well as obtains all the pages necessary for MSAM operations.

If the DCB COMBINE flag is set, the reader is assumed to be on the same 2540 frame as the punch and the symbolic device address of the reader must be one greater than that of the punch.

A violation of any of the following restrictions will cause the OPEN macro instruction to abnormally terminate the task.

1. The DCB MACRF field must specify only that GET or PUT macro instructions will be issued.
2. The DCB DEVD field must specify (possibly from the DDEF command) a card reader, card punch, or printer, and this device must correspond to the device specified in the DDEF command.
3. If the device is a card reader, the data set must be opened for input.
4. If the device is a card punch or printer, the data set must be opened for output.
5. The DCB RECFM field must indicate fixed-format records or variable-format records; A/M control characters may also be specified.
6. The DCB DEVD must specify the card punch, if the DCB COMBINE flag is set.

USE OF L- AND E-FORM: You may use the L- and E-form of this macro instruction. The E-form may specify any parameters. Furthermore, the parameters specified in the E-form will overlay parameters specified in the L-form. The E-form may not specify more DCB operands than are specified in the L-form.

CLOSE -- Disconnect Data Set from User's Problem Program (S)

The CLOSE macro instruction disconnects one or more data sets from the user's problem program.

Name	Operation	Operand
[symbol]	CLOSE	(dcb-addr,...)

dcb
specifies the address of the data control block opened for the data set whose processing is to terminate.

CAUTION: The following errors cause the results indicated:

Error	Result
Closing data control block that is already closed	No action
Closing when dcb operand does not specify address of data control block	Task terminated
Closing data control block containing invalid DSORG specification	Task terminated

PROGRAMMING NOTES: You may specify any number of data control block addresses and associated options in the CLOSE macro instruction. This facility makes it possible to close data control blocks and their associated data sets in parallel.

In most instances, the FINISH macro instruction should precede CLOSE (see the more detailed explanation of the FINISH macro instruction in "Macro Istructions for MSAM") since you cannot be informed from CLOSE of errors that may have occurred in processing the last output buffer page. Additionally, the use of CLOSE without a preceding FINISH that returned a normal completion code would cause the task to wait until the I/O operation for that DCB is complete.

The CLOSE macro instruction for MSAM releases all the storage area that was used for MSAM. The options of REREAD and LEAVE are ignored. If the FINISH macro instruction did not precede the CLOSE, and if the DCB INHMSG=0, a message will be written to the operator to remove the data set from the device. If the DCB COMBINE flag is set and if a FINISH did not precede the CLOSE, a card will be read from the reader on the same 2540 as the selected punch and stacked in pocket 3.

USE OF L- AND E- FORM: You may use the L- and E-forms of this macro instruction. The E-form of the macro instruction may specify any parameters. Furthermore, the parameters specified in the E-form will overlay those specified in the L-form. The E-form may not specify more dcb operands than are specified in the corresponding L-form.

MACRO INSTRUCTIONS FOR MSAM

There are four macro instructions that you may use in your MSAM programs. One of these, SETUR, enables you to specify the unit-record configuration you desire for on-line printers and punches. Two of the others, GET and PUT, access logical records and may be specified in either a move mode or a locate mode; the third, FINISH, informs the MSAM routines that a break point has been reached in processing a data set.

Interrupt Entry Handling

For each of the MSAM macro instructions (SETUR, GET, PUT, and FINISH), a return code of 4 indicates that the operation has not yet

been completed. In each case, the macro should be reissued, until a return code other than 4 is received. However, between repetitions of the macro instruction, you should interrogate DCBICB and, if it is non-zero, invoke the interrupt inquiry routine by issuing the INTINQ macro instruction (which is described in Assembler Users Macro Instructions) to determine whether an asynchronous interrupt is pending. If you should find that this is indeed the case, you should give control to the appropriate interrupt-handling routine and defer reissuing the MSAM macro instruction until you have control returned to your program.

SETUR - Unit Record Device Set Up (R)

The SETUR macro instruction enables you to specify the unit record configuration for on-line printers and card punches.

Name	Operation	Operand
symbol	SETUR	dcb-{addr / (1)}, param-{addr / (0)}

dcb
specifies the address of the data control block opened for processing a data set on a printer or card punch.

param
specifies the address of the unit record device-setup parameter. For card punches, the parameter is the desired form number and is 10 bytes in length. For printers, the parameter is a six-byte key used to refer to a VISAM system data set (SYSURS).

PROGRAMMING NOTES: The SETUR macro instruction should be issued before any I/O operations are directed to a printer or punch, to ensure a valid setup. This is done by issuing SETUR immediately after opening a data set or after the FINISH macro instruction is executed and the I/O operation completed.

Card Punch: The setup for a card punch is described by the form numbe of the card that the operator is to load into the punch-feed hopper o the 2540. This form number is an installation-defined constant. When the macro instruction is executed, the SETUR routine checks to see which form is mounted in the punch (the currently mounted form number -- or zeros if the DCB was just opened -- is stored for each device in the SDAT). If the desired form is already mounted, control is returned to the invoking routine with a return code of 0. If the form is not mounted, a message is written to the operator (WTO) to mount the desired form number (10-byte parameter), and to ready the 2540 punch. A return code of 4 is provided to the calling task. When the operator indicates the punch is properly loaded, by causing a not-ready to ready interrupt, the SDAT is changed to reflect the new form number and on the next call to SETUR control is returned to the invoking routine with a return code of 0.

Printer: A maximum of nine variables can be used to describe the setup of a printer. The six-byte parameter whose address is specified in the SETUR macro instruction is used as a key to read the VISAM system data set SYSURS. This data set, maintained by the installation, defines a setup configuration for each KEY (parameter). The following setup variables are specified in the SYSURS data set and are stored in the SDAT or DCB or DEB page.

1. Form Number: the number of the form to be mounted; this number is an installation-defined 10-byte EBCDIC constant.

2. Carriage Tape Number: the number of the carriage control tape to be mounted; this number is an installation-defined four-byte EBCDIC constant.

3. Chain/Train Type Number: the number of the print chain or print train to be mounted in the 1403; this number is an installation-defined four-byte EBCDIC constant.

4. Density Number: the density of lines per inch to be selected for the printer; this is a TSS/360-defined one-byte number, which must be an EBCDIC '8' or EBCDIC '6'.

5. Form Type Code: specifies the error retry to be used with this printer configuration; it is a TSS/360-defined one-byte code, which must be an EBCDIC 'D', an EBCDIC 'F', or an EBCDIC 'S'.

   D
   designates a storage DUMP-type retry including striking over the line in error, spacing one line, and rewriting the line. If three such strike-out lines should appear on one page, an eject to channel 1 is performed. The strike-over character for nonUCS (universal character set) printers is an EBCDIC 'X'.

   F
   designates a FORM-sensitive error retry including write-to-operator (WTO) to mark the error form, eject to channel 1, and rewrite the entire form (page). If a form-type F is specified, the USASI or machine-type control characters must be used to control the carriage and printer. Also, SDAMRB, the number of buffers specified in the SDAT, must be at least 3 to ensure that at least one page image is available for error retry.

   Note: Buffers required to store page images reduce I/O overlap.

   S
   designates a SEQUENCE-sensitive error retry including write-to-operator (WTO) to mark the error form. The system completes the error form and continues to the next form when a not-ready to ready interrupt is received. No attempt is made to reprint the error form. If a form-type S is specified, the USASI or machine-type control characters must be used to control the carriage and printer.

6. UCS Folding Code: specifies whether the folding option of the universal character set feature is desired. The code is not referenced if the printer is not equipped with the UCS special feature. This one-byte EBCDIC code is TSS/360-defined. The possible codes are F (folding) and U (unfolded).

7. UCS Strike-out Character Code: determines the UCS strike-out character to be used in connection with error retry. This code is never referenced if the printer is not equipped with the universal-character special feature, or if the form-type code is not D. The code is a two-byte EBCDIC field; each of the bytes may be set to a numeric 0-9 or a letter in the range A-F and represents one hexadecimal digit. The two combined hexadecimal digits represent the UCS strike-out character. For example, an EBCDIC two-byte code of 'D8', when translated into hexadecimal, becomes '1101 1000' in binary, which is the graphic "Q" in EBCDIC. This may be printed with UCS as another graphic.

8. UCS Buffer Load Key: a six-byte, installation-supplied constant key used to read the VISAM system data set SYSUCS, which specifies a UCS buffer configuration. This key is not referenced if the

printer is not equipped with the UCS special feature. SYSUCS also contains a 40-byte verification message.

9. Printer Alignment Message: a message occupying one print line of 132 bytes, to be written on the printer so that longitudinal form alignment may be accomplished.

EXECUTION: Upon completion of the SETUR macro instruction, a code indicating the manner in which the instruction was completed is returned in general register 15. All return codes are multiples of 4 and their meanings are given in Table 11.

When the SETUR macro instruction is executed, the routine determines if the present configuration of the printer, specified in SDAT, is the configuration specified by the SETUR parameter. If the desired form, carriage tape, chain/train, and/or density is already as required, the operator is notified of the current configuration; control is returned to the invoking routine. If the desired setup configuration is not present, the system initiates the action necessary to achieve the desired setup. This may include:

1. A write-to-operator message to mount the desired form or carriage tape or chain/train type; to select a different density; or any combination of the above. When the message to the operator has been written, a return to the invoking routine is made with a return code of 4.

2. After an asynchronous interrupt is received from the device indicating that the operator has hit the STOP button and then the START button, subsequent issuance of the SETUR macro instruction may cause the UCS buffer to be loaded.

3. For type-F and type-S forms, an eject to channel 1 is taken, and a printer alignment message is written on the printer. A WTO message is sent to the operator to align the printer, and control is returned to the invoking routine with a return code of 4. (See the section on "Interrupt Entry Handling," above, for a description of the proper procedure to be followed.) The operator indicates that alignment is complete by causing an asynchronous interruption. He does this by hitting the STOP button and the START button on the printer. When a SETUR is issued after the alignment is complete, a return code of 0 is returned. If an unrecoverable error has occured, a return code of 8 is returned.

If a routine wants to stop the SETUR procedures, the DCB field SUR should be set to 0, and the SETUR macro instruction should be issued. The SETUR routine sets the flag to ON when a code of 4 is returned, and sets the flag to OFF when any other code is issued. If a return code of 4 is provided and the invoking routine wants to purge or discontinue the SETUR procedures, the setting of DCBSUR flag should be changed from 1 to 0 before the next SETUR macro instruction is issued.

The following procedure is used when loading the UCS buffer. (The carriage control tape, print chain/train, and density will have already been selected and/or mounted at this time.)

1. Eject to channel 1.

2. Write first 120 bytes of the 240-byte buffer on the printer.

3. Skip one line.

4. Write second 120 bytes of the 240-byte buffer on the printer.

5. Skip two lines.

6. Write first 20 EBCDIC characters of verification message in SYSUCS on the printer.

7. Eject to channel 1.

8. Write second 20 EBCDIC characters of verification message in SYSUCS, on the console typewriter.

9. Write to operator (WTO) to verify that the same print line was printed on the on-line printer and on the console. When the operator has completed this action, he replies by pressing the STOP button followed by the START button.

NOTE: The printer output graphics must be the same as those written on the console typewriter. Therefore, the two 20-byte segments combined in the verification message in SYSUCS may be different in storage. The difference between these segments is determined by the UCS buffer load and the print chain/train mounted.

Table 11. Return Codes for SETUR Macro

Return Code	Meaning
0	Operation completed successfully
4	Operation not complete; SETUR macro instruction should be reissued
8	Unrecoverable error occurred in (1) reading SYSURS or SYSUCS, or (2) attempting to write UCS buffer load or UCS verification message, or (3) attempting to load UCS buffer
12	Parameter specified in SETUR macro instruction is not valid SYSURS KEY
16	SYSUCS buffer load key as specified in SYSURS is not valid

All messages written to the operator are by WTO. If UCS is involved, a command to unblock data check will always be issued.

SYSURS and SYSUCS are VISAM partitioned data sets, maintained and created by each installation. The formats are shown in Figures 20 and 21.

	0 7	8 27	28 47	67
Line 1	xxxxxx1	Verification	Message	Reserved
		(printer) First 20 bytes of verification message	(typewriter) Second 20 bytes of verification message	
Line 2	xxxxxx2	1st 60 bytes of 240-byte buffer load		
Line 3	xxxxxx3	2nd 60 bytes of 240-byte buffer load		
Line 4	xxxxxx4	3rd 60 bytes of 240-byte buffer load		
Line 5	xxxxxx5	4th 60 bytes of 240-byte buffer load		

xxxxxx=6-byte parameter key provided by SETUR routine from entry in SUSURS routine

Figure 20. Complete Entry in SYSUCS (5 line records, each 68 characters long, including KEYS)

	0 5	6	7	8 9	10 19	20 23	24 27	28 30	31	32 34	35 36	38 39	40 41	42 43	44 49	50 55	56 67
Line 1	XXXXXX Key from SETUR	1	R	←	XXXXXXXXX Form No.	XXXX Carriage tape No.	XXXX Chain train No.	R	X Density No.	R	form type code	R	X UCS folding code	R	XX UCS Strike out	R	XXXXXX UCS buffer load Key used to read SYSUCS	R
Line 2	XXXXXX Key from SETUR	2	R	1st 60 bytes of alignment message														
Line 3	XXXXXX	3	R	2nd 60 bytes of alignment message														
Line 4	XXXXXX	4	R	last 12 bytes of alignment message														

R = reserved
Byte 6 = line number

Figure 21. Complete Entry in SYSURS (4-line record, each 68 characters long, including KEY)

GET -- Get a Record (R)

The GET macro instruction may be specified in either the locate mode or the move mode. When you specify the macro instruction in the locate mode, GET locates the next sequential record in the specified input data set and places its address in general register 1. When you specify the macro instruction in the move mode, GET locates the next sequential record in the specified input data set and moves it to the work area you have specified in virtual storage.

Name	Operation	Operand
[symbol]	GET	dcb-{addrx / (1)}[,area-{addrx / (0)}]

dcb
specifies the address of the data control block opened for the data

set being processed. If (1) is written, you must load the address of the DCB into general register 1 before execution of the macro instruction.

area
specifies the address of the work area into which you want the record to be moved; hence, this operand is used only when the macro instruction is specified in the move mode. If (0) is written, you must load the address into general register 0 before execution of the macro instruction.

PROGRAMMING NOTES: You must place the address of a save area in general register 13 before executing the GET macro instruction.

When you are using MSAM, the GET macro instruction may only be employed to obtain records from a card reader. Hence, the RECFM field of the data control block must indicate format F, since format V is not supported for the card reader. At OPEN time, the LRECL field of the data control block should be set to a maximum of 80 bytes for EBCDIC, or 160 bytes for column binary. The mode field in the data control block must be set to a binary 0 for EBCDIC or to binary 1 for column binary.

CAUTION: Should any field of the DCB be altered by an improper source, the results are unpredictable when this macro instruction is executed.

EXECUTION: Upon completion of the GET macro instruction, a code indicating the manner in which the instruction was completed is returned in general register 15. All return codes are multiples of 4; their meanings are given in Table 12.

Table 12. Return Codes for MSAM GET Macro Instruction

Return Code	Meaning
0	Operation completed successfully.
4	I/O not complete; no record has been provided since next sequential buffer has not yet been filled; GET macro instruction should be reissued.
8	Unrecoverable I/O error occurred in connection with record being read; normally, CLOSE macro instruction should be issued; however, a record has been provided, content of which is buffer image. If I/O error was not permanent (DEBNF2 or DECG1 not on), you may accept record and continue processing, or you may skip record by issuing another GET macro instruction.
12	End-of-file; no record has been provided. The FINISH macro instruction should be issued.
16	Control card sensed; attempt to read an EBCDIC record resulted in validity check; first four columns of card contain same predetermined control mark, a 12-11-3-4 punch. Record provided is buffer image, control bytes of which should be tested for such installation-defined codes as (a) change of mode from EBCDIC to column binary or (b) change of data group without end-of-file indicator. Depending on installation assignment and control code usage, processing may continue. Control card will be stacked as if it were valid data card. If subsequent GET macro instruction is issued, it will refer to next card in reader, following control card. If any fields in data control block are to be changed, FINISH (or CLOSE and OPEN) macro instructions must be issued before the next GET.

EXAMPLE: In the following example, which illustrates the use of both the locate-mode and move-mode GET macro instructions, you want to read 65 EBCDIC bytes from the first 65 columns of the next sequential card, in a 2540 reader. Any cards with errors will be stacked in bin 2, with no attempt to reread the record; cards containing no errors will be stacked in bin 1. Since the return codes provided from the macro instruction are multiples of 4, it is possible for you to set up a branch table to provide proper control of processing.

```
ADL       DCB   ,MACRF=G,DDNAME=MYDD,      BUILD DCB
                DEVD=RD,MODE=E,
                STACK=1,RECFM=F,LRECL=65
          OPEN  (ADL,(INPUT))              OPEN DCB
          LA    3,RCTABLE                  SET UP BRANCH TABLE
          LA    1,ADL                      LOAD ADDR OF DCB
MOVE      GET   (1),WORK                   MOVE-MODE GET MACRO
          L     5,0(15,3)                  BRANCH ON RC INDEX AND RCTABLE
                                           AS BASE
          BR    5
LOCATE    GET   ADL                        LOCATE-MODE GET MACRO
          L     5,0(15,3)                  BRANCH ON RC INDEX AND
          BR    5                          RC TABLE AS BASE
          .
          .
          .
WORK      DC    CL65                       INPUT AREA FOR MOVE-MODE
                                           GET MACRO
```

```
RCTABLE     DC    A (NORM)          ADDR FOR PROCESSING AFTER
                                    RC OF 0
            DC    A (PAUSE)         ADDR FOR PROCESSING AFTER
                                    RC OF 4
            DC    A (ERROR)         ADDR FOR PROCESSING AFTER
                                    RC OF 8
            DC    A (END)           ADDR FOR PROCESSING AFTER
                                    RC OF 12
            DC    A (CONTROL)       ADDR FOR PROCESSING AFTER
                                    RC OF 16
```

Both the move-mode and locate-mode GET macro instructions result in a type-I linkage to the DOMSAM routine.

PUT -- Put a Record (R)

The PUT macro instruction may be specified in either the locate mode or the move mode. When you specify the macro instruction in the locate mode, PUT returns, in general register 1, the address within an output buffer of an area large enough to contain an output record; you should then construct, at that address, the next sequential logical record of the output data set. When you specify the macro instruction in the move mode, PUT moves the next sequential logical record of the output data set, from the location you have specified into an output buffer.

Name	Operation	Operand
[symbol]	PUT	dcb-{addrx | (1)} [,area-{addrx | (0)}]

dcb
: specifies the address of the data control block opened for the data set being processed. If (1) is written, you must load the address of the DCB into general register 1 before execution of the macro instruction.

area
: specifies the address of the next logical record to be moved into the output buffer; hence, this operand is used only when the macro instruction is specified in the move mode. If (0) is written, you must load the address into general register 0 before execution of the macro instruction.

PROGRAMMING NOTES: The length of the logical record is determined by the value of the LRECL field of the data control block for fixed-length (format-F) records and by the value of the control bytes for variable-length (format-V) records. If you write format-V records, the value of the LRECL field must be set equal to the maximum length of the logical record prior to the locate-mode PUT macro instruction. The value may be changed between executions of the PUT macro instruction and will be used to determine when to truncate the present buffer. The control program uses the current value of the LRECL field to determine the amount of buffer space needed for the record, even though the actual length is determined by the built by the user in the buffer area after the completion of the locate-mode PUT macro instruction.

If you do not specify USASI or machine code in the RECFM field of the data control block, the PRTSP and STACK fields of the DCB will be used to control line spacing and stacker selection, respectively.

Printer: The MODE field of the data control is not referenced for the printer. The LRECL field of the data control block must be set to a value not exceeding 133 bytes for format-F records, and 137 bytes for format-V records. These values are 132 bytes and 136 bytes, respectively, if the USASI or machine code was not specified in the RECEM field of the data control block. If you use control characters (A or M) for carriage control, channel 12 is ignored (greater efficiency is achieved by not having a channel 12 punched on the CC tape). However, if you do not use control characters and channel 12 is sensed, an immediate eject to channel 1 is performed.

Card Punch: For format-F records, you must set the LRECL field of the data control block to a value not exceeding 81 bytes for EBCDIC, and 161 bytes for column binary. These values are 80 bytes and 160 bytes, respectively, if the USASI or machine code was not specified in the RECFM field of the data control block. For format-V records, you must set (for locate mode only) the LRECL field to a value not exceeding 85 bytes for EBCDIC or 165 bytes for column binary; these values are 84 and 164 bytes, respectively, if the USASI or machine code was not specified in the RECFM field. The MODE field of the data control block must contain a binary 0 for EBCDIC or a binary 1 for column binary.

CAUTION: If any field of the DCB is altered by an improper source, the task may be abnormally terminated when a PUT macro instruction is executed.

EXECUTION: Upon completion of the PUT macro instruction, a code indicating the manner in which the instruction was completed is returned in general register 15. All codes are multiples of four, and their meanings are given in Table 13.

Table 13. Return Codes for MSAM PUT Macro Instruction

Return Code	Meaning
0	Operation completed successfully.
4	I/O not complete; the record has not been accepted as there is no room remaining in present buffer and next sequential buffer has not yet been released from prior I/O request. PUT macro instruction should be reissued. (See discussion of "Interrupt Entry Handling," above.)
8	Unrecoverable I/O error occurred and record was not accepted. General register 1 points to record on which I/O error occurred -- in the case of an equipment check on card punch, general register 1 points to record immediately following that on which error occurred -- and general register 0 points to associated DECB. FINISH and/or CLOSE macro instructions may be issued. However, if I/O error was not permanent (DEBNF2 or DECG1 not on), you may continue processing records beyond the one that failed by reissuing PUT.

EXAMPLE: In the following example, which illustrates the use of both the locate-mode and move-mode PUT macro instructions, you want to print a file of 132-byte EBCDIC records. After each line is printed, one line is spaced. Since the return codes provided by the macro instruction are multiples of 4, it is possible for you to set up a branch table to provide proper control of processing.

```
JHL       DCB   ,MACRF=P,                          BUILD DCB
                DDNAME=TODD,DEVD=PR,
                PRTSP=1,RECFM=F,LRECL=132
          OPEN  (JHL,(OUTPUT))                     OPEN DCB
                .
                .
                .
                .
          LA    3,RCTABLE                          SET UP BRANCH TABLE
          LA    1,JHL                              LOAD ADDR OF DCB
MOVE      PUT   (1),WORK                           MOVE-MODE PUT MACRO
          L     5,0(15,3)                          BRANCH ON RC INDEX AND
          BR    5                                  RCTABLE AS BASE
          LA    1,JHL
LOCATE    PUT   (1)                                LOCATE-MODE PUT MACRO
          L     5,0(15,3)                          BRANCH ON RC INDEX AND
                                                   RCTABLE AS BASE
          BR    5
                .
                .
                .
                .
WORK      DS    CL132                              OUTPUT AREA FOR MOVE-
                                                   MODE PUT MACRO
RCTABLE   DC    A(NORM)                            ADDR FOR PROCESSING
                                                   AFTER RC OF 0
          DC    A(PAUSE)                           ADDR FOR PROCESSING
                                                   AFTER RC OF 4
          DC    A(ERROR)                           ADDR FOR PROCESSING
                                                   AFTER RC OF 8
```

Both the move-mode and locate-mode PUT macro instructions result in a type-I linkage to the DOMSAM routine.

FINISH -- End of Data Set (R)

The FINISH macro instruction is used to inform the MSAM routines that processing of the current data group (subsection of a data set) is at an end. A task may process more than one data group (within the same data set) with the same data control block, without closing and reopening the DCB (and the assignment and release of the associated I/O device) between data groups.

Name	Operation	Operand
symbol	FINISH	dcb-{addr / (1)}

dcb
specifies the address of the data control block opened for the data set being processed.

PROGRAMMING NOTES: The FINISH macro instruction provides for

1. Initiating any necessary I/O activity so that an output data set may be closed;
2. Testing the results of all outstanding I/O on an output data set;
3. Awaiting completion of outstanding I/O requests on an input data set;

4. Notifying an operator to remove the data set from the device (under control of the INHMSG flag of the data control block);

5. Reading a card from the card reader and stacking it in pocket 3 of the same 2540 as the selected punch, if the COMBINE flag of the data control block is set.

You should precede the CLOSE macro instruction with the FINISH macro instruction, if you want to avoid an automatic WAIT condition, which may result from the access method CLOSE, or to receive notification of any I/O error. (CLOSE MSAM is the only MSAM routine to invoke AWAIT.)

CAUTION: The FINISH macro instruction will cause the operator to be notified to remove the data group from the device in use if the INHMSG flag of the data control block is not set to 1.

EXECUTION: Upon completion of the FINISH macro instruction, a code indicating the manner in which the instruction was completed is returned in general register 15. All return codes are multiples of 4; their meanings are given in Table 14.

Table 14. Return Codes for MSAM FINISH Macro Instruction

Return Code	Meaning
0	Operation completed successfully. CLOSE macro instruction may be issued or further processing may be initiated without reopening data control block; the DCB parameters LRECL, MODE, STACK, PRTSP, RECFM, POCKET, FORMTYPE, and RETRY may be altered at this time.
4	Operation was not completed since I/O was not finished. FINISH macro instruction should be reissued later, until a return code other than 4 is received. (See discussion of "Interrupt Entry Handling," above.)
8	Operation was completed with I/O error. If data control block was opened for input, description of GET macro instruction return code of 8 (Table 12) applies; if data control block was opened for output, description of PUT macro instruction return code of 8 (Table 13) applies. In order to flush remaining output buffers, if error was not permanent (DEBNF2 or DECG1 not on), FINISH may be reissued.

Example: The following example is based upon the coding in the example for the PUT macro instruction. It would follow the locate-mode PUT, L, BR:

```
         LA      7,FINTAB                SET UP BRANCH TABLE
REPEA    FINISH  JHL                     END OF DATA SET
         L       5,0(15,7)               BRANCH ON RC INDEX AND
         BR      5                       FINTAB AS BASE
HALT     CLOSE   JHL
         .
         .
         .
         .
FINTAB   DC      A(HALT)                 ADDR FOR PROCESSING AFTER
                                         RC OF 0
         DC      A(REPEAT)               ADDR FOR PROCESSING AFTER
                                         RC OF 4
         DC      A(ERROR)                ADDR FOR PROCESSING AFTER
                                         RC OF 8
```

TERMINAL ACCESS METHOD MACRO INSTRUCTIONS

This section provides you with a description of the macro instructions you will need to make use of the Terminal Access Method (TAM) routines. These routines comprise control unit and TERMINAL oriented functions. The access method provides macro instructions which enable a user to read messages into the computer and to write messages to terminals. The facilities provided include:

- Terminal polling
- Terminal addressing
- Answering
- Message receiving
- Dynamic buffering
- Message Transmitting
- Dialing
- Conversion
- Interrupt Control
- 2702 Control Orders

DCB -- Set Up Data Control Block (nonstandard)

The DCB macro instruction establishes a parameter list which provides the interface between the user's program and data set and the system I/O routines.

Name	Operation	Operand
symbol	DCB	[DDNAME=symbol] [,DSORG=CX]
		[,MACRF= { (R) / (W) / (R,W) }] [,BUFNO= absexp]
		[,BUFL=absexp] [,BFTEK=D]
		[,EXLST=relexp] [,SYNAD=relexp]

DDNAME
designates the name of the DD statement associated with this DCB.

DSORG
specifies the data set organization as being that of a communication line.
CX - specifies communication line.

MACRF
indicates that access to this data set is to be gained through this DCB only by the specified macro instructions.
(R) - access can be gained only by use of the READ macro instruction.
(W) - access can be gained only by use of the WRITE macro instruction.
(R,W) - access can be gained by use of either the READ or the WRITE macro instruction.

BUFNO
specifies the number of buffers to be provided for the buffer pool. The maximum you may specify is 255.

BUFL
specifies the byte length of each buffer in the pool or a standard length for user provided buffer. The maximum length you can specify is 32,767 bytes.

BFTEK
specifies that dynamic buffer allocation is to be provided. If you omit this parameter the system assumes that you have provided for your own buffer allocation.
D - dynamic allocation.

EXLST
specifies the symbolic address of the exit list.

SYNAD
specifies the address of the synchronous error routine.

PROGRAMMING NOTE: BUFNO, BUFL, BFTEK, and SYNAD may be supplied at execution time at any point up to OPEN time. BUFNC, BUFL, and BFTEK may be specified in the DD statement.

DCBD -- Specify DCB DSECT (nonstandard)

You may use the DCBD macro instruction to access the fields in the DCB. See Assembler User Macro Instructions for a description of DCBD.

OPEN -- Prepare DCB for Processing (S)

The OPEN macro instruction prepares the DCB for use with a communication line. Each DCB must be opened before message transmission can begin.

Name	Operation	Operand
[symbol]	OPEN	({dcb-addr,}.....) [MF={(E,list-{addrx / (1)})}]

dcb
specifies the address of the DCB associated with the communication line to be opened.

MF
specifies the form of the macro instruction. If this operand is

omitted, the macro instruction is executed with all the parameters specified in this issuance of the macro instruction.

L - L-form. No executable code is generated and the DCB is not actually opened. Only a parameter list is generated and is assigned the name of the OPEN macro instruction.
E - E-form. The OPEN function is actually executed.

list
specifies a previously built parameter list. The OPEN function is performed for each entry in the list.

CLOSE -- Remove Communication Lines From Use (S)

The CLOSE macro instruction is used to close a previously opened DCB associated with a communication line. The CLOSE macro instruction can be used in conjunction with the OPEN macro instruction to control data set organization for communication lines in the system.

Name	Operation	Operand
[symbol]	CLOSE	(dcb-addr,...) [MF={L | (E,list-{addrx | (1)})}]

dcb
specifies the DCB associated with a communication code to be closed.

MF
indicates the form in which this macro instruction is written. If this parameter is omitted, the function is executed using the parameters supplied in the macro instruction.
L - L-form. No executable code is produced and the close function is not actually performed. A parameter list is generated using the parameters supplied in the first operand.
E - E-form. The CLOSE function is performed.

list
designates a previously generated parameter list. The close function is performed for each entry in the list.

READ -- Read From Another Terminal (S)

The READ macro instruction causes contact to be established with a terminal. If that terminal has a message to transmit, contact is maintained until an EOT or FOB character is received. The message is decoded from line code to EBCDIC and posted as complete in the DECB and the READ operation is then considered complete. Although control returns to you immediately upon initiation of the channel program, you must test for the completion of the event before issuing another READ or WRITE. The CHECK macro instruction is provided for this purpose.

Name	Operation	Operand
[symbol]	READ	decb-{symbol / (1)},type-code, dcb-addr, area-{addr / 'S'},length-{value / 'S' / 'C'} , [arg1-addr] , [arg2-code][, MF={L / E}]

decb

specifies the address of the DECB in which you want completion information to be posted. If register notation is used, you must place this address in register 1.

type

specifies the type of transmission you require and may be one of six codes.

TID - read initial with dialing. This indicates that an automatic dial connection is to be made with the terminal. The dialing digits must be specified in the terminal entry list specified in "arg1". If the terminal type require polling, the necessary polling sequence characters are generated.

TIN - read initial. A previously established line connection is assumed. If the terminal type requires polling, the necessary polling sequence characters are generated.

TCN - read continue. When polling is not required, specify this option. You may use this option when contacting a terminal which was previously polled and in a transmit state.

The next three options apply when automatic retransmission of messages received in error is desired and the terminal is equipped with error correction facilities. A predetermined number of retries, specified by the terminal type, will be attempted for each message received in error. The posting of uncorrectable errors will include the appropriate error information. The basic types are as above.

TDR - read initial with dialing/repeat.

TNR - read initial/repeat.

TCR - read continue/repeat.

dcb

specifies the DCB associated with the line.

area

specifies the address of the first byte of your input area. If you write 'S' TAM will provide the buffer area based on the length parameter.

length

specifies the byte length of the input area which will receive the message.

'S' - tells TAM to use the buffer length you have specified in the DCB.

'C' - tells TAM to use a standard buffer length appropriate to the type of terminal. A mode of operation will be started in which one line or record will be read from the terminal.

arg1
specifies the address of the terminal entry list which contains the dialing digits. If this parameter is omitted, a standard station character is to be used.

arg2
indicates the character set to be used in decoding the message. If omitted the standard character set will be used. The permissable codes and their meaning can be found in Table 15.

MF
specifies the form in which the macro instruction has been written.

L - L-form. Only a parameter list is generated. The read function is not performed.

E - E-form. A previously generated parameter list is accessed, may be changed, and is then used in performing the read function. When either L or E is specified, the only parameters required are decb and type.

If the MF operand is omitted, a parameter list will be generated and the executable form of the read function will be generated.

Table 15. Character Set Codes

	TERMINAL KEYBOARD TYPE			
CODE	IBM 1050	IBM 1052-7	IBM 2741	TELETYPE MODE 35
A	PTTC/8	EBCDIC	PTTC/8	GSA
B	PTTC/6	-		-
C	-	-		-

WRITE -- Write a Message (S)

The WRITE macro instruction causes the transmission of a message to a terminal and generates a control order for the device control unit.

Name	Operation	Operand
[symbol]	WRITE	decb-{symbol}, type-code, dcb-addr, {(1)} [area-addr], length-value, [arg1-addr], [arg2-code] [,MF={L / E}]

decb
specifies the DECB in which you want completion information to be posted. You must CHECK for completion before issuing another READ or WRITE. If register notation is used, this address must first be placed in register 1.

type

specifies the type of transmission you require or the type control order you want generated. Any one of 18 codes may be specified in this operand.

TID - write initial with dial. An automatic connection is made with the terminal. The dialing digits should have been specified by the DFTRMENT macro instruction and located in terminal entry list specified by arg1. If the terminal requires addressing, the necessary addressing sequence characters will be generated.

TIN - write initial. This option assumes that you have already made the line connection. The necessary addressing sequence characters will be generated, if required.

TCN - write continue. Specify this option when addressing is not required; if you have previously addressed the terminal and it is in a receive state.

TIA - write with response. This option assumes that you have previously made a line connection. If the terminal requires polling and addressing, the necessary sequence characters are generated. This option provides the ability to transmit a message to a terminal and to receive its next output record or line as a response. The maximum size message you can transmit is 32,767 bytes; the maximum size response you can receive is one logical record or line as specified by terminal type.

The next four options apply to terminals equipped with error correction facilities and automatically retransmit messages sent in error. A predetermined number of retries, depending on the terminal type, will be attempted for each erroneous message. The posting of uncorrectable errors will include appropriate error information.

TDR - write initial with dialing/repeat.

TNR - write initial/repeat.

TCR - write continue/repeat.

TAR - write with response/repeat.

The remaining ten optional type parameters are used to issue control orders.

AUTOWRAP - causes the transmission control unit to wrap the output of the addressed line to the input of line zero. The command within the channel operates as a write.

DISABLE - causes the transmission control unit to reset the enable latch within the line adapter of the addressed communication line. No data transfer occurs.

ENBLASYN - causes the transmission control unit to set the enable latch within the line adapter of the addressed communication line, with software providing the necessary interrupt to invoke posting. No data transfer occurs.

ENBLSYN - causes the transmission control unit to set the enable latch within the line adapter of the addressed communication line with hardware providing the necessary interrupt to invoke posting.

PREPARE - sets up the communication line to detect attention signaling.

SADONE - On acceptance of this command, the 2702 will set the TC field within the addressed LCW to one. This has the effect of associating the terminal control with the line oscillator with internal address of one with the addressed communication line. No data transfer occurs.

SADTWO - Operates the same as SADONE except that terminal two is associated with the addressed line.

SADTHREE - terminal three is associated with the addressed line.

SADZER - terminal zero is associated with the addressed line.

BREAK - causes the addressed line to transmit a continuous space signal. Bytes transferred from the channel to the addressed unit must be all zeros. To provide control over the length of space signal, a byte count must be specified in the length field. If no count is given, a value of two is assumed.

dcb
specifies the address of the DCB for the line.

area
specifies the address of the first byte of your output area.

length
specifies the number of bytes in the output message.

arg1
indicates the address of the terminal entry list. If you omit this parameter, a standard status code will be used.

arg2
specifies the character set to be used to translate EBCDIC to the proper line. The codes to be used are the same as those used for READ and can be found in Table 8.

MF
specifies the form in which the macro instruction is written. If this parameter is omitted, standard form will be assumed. The valid form codes are:
L - L-form. A parameter list will be generated but the write function will not be performed.
E - E-form. A previously generated parameter list will be accessed and the write function performed. The parameter list may be altered with this form.

Note: When using the E-form, the only parameters required are decb and type.

CHECK -- Wait for and Test for Completion of Read or Write Operation (R)

The CHECK macro instruction tests for the completion of a read or write operation. If not complete, CHECK waits for completion. CHECK also detects errors and exceptional conditions.

Name	Operation	Operand
[symbol]	CHECK	decb-{addrx / (1)}

decb
specifies the DECB created as part of the expansion of a READ or WRITE macro instruction and in which completion information is posted. If you write (1), you must first load the address of the DECB into register 1.

PROGRAMMING NOTES: If an I/O error occurs and the read or write operation did not complete correctly, control will be given to your SYNAD routine, if one exists. If you have not specified one your task will be abnormally terminated. Your SYNAD routine may use the RETURN macro instruction to resume processing.

Figure 22 shows the format of the DECB, with the flags field detailed in Figure 23.

When the read or write function completes normally, byte zero of the DECB is set to X'7F'. The CSW and sense information are also stored but have no significance.

In posting the completion of a normal write operation (except Write with Response - TIA) the write bit in the DECB flag field is set; all other fields are undisturbed.

In posting the completion of a type TIA write operation, the data area address field of the DECB will be changed to contain the address of the input data and the length field will contain the input data length. The read bit in the DECB flag field is set.

In posting the normal completion of a read operation, the data area address field will contain the address of the edited data and the length field will contain the number of bytes in the data area and the read bit in the DECB flag field is set.

In posting a completion with exception conditions, byte zero of the DECB is set to X'41' and appropriate flags are set in the DECB. (See Figure 23.)

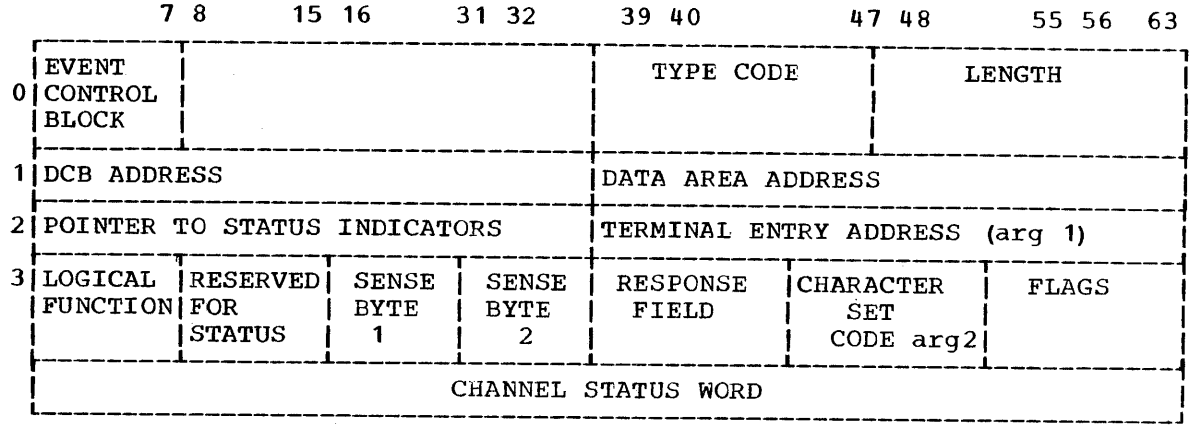


Figure 22. DECB Format

48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
FLAGS															
		A	U	I	B	S	I	A		R	I	W	R	R	A
		C	S	N	U	Y	N	B		E	N	R	E	E	A
		T	E	P	F	S	T	E		Q		I	A	S	T
		I	R	U	F	T	E	N		U	U	T	D	P	E
		V		T	E	E	V	D		E	S	E		O	N
		E	E		R	M	E			S	E			N	T
			R	A			N	R		T				S	I
			R	R	O	E	T	E						E	O
			O	E	V	R	I	Q		S					N
			R	A	F	R	O			Y					
					L	O	N			N					
				O	O	R				A					
				V	W		R			D					
				F			E								
				L			Q								
				O											
				W											

Figure 23. Flag Field of the DECB

DFTRMENT -- Define a Polling List (nonstandard)

The DFTRMENT macro instruction provides the capability for defining a polling or addressing list and entering the physical addresses and device specifications into the generated list.

Name	Operation	Operand
[symbol]	DFTRMENT	[DIAL=(integer, ...)]
		[,ADRID=(adrid-characters,...)]
		[,POLLID=(pollid-characters,...)]

DIAL
specifies the dial digits.

ADRID
specifies the addressing character sets. The number of addressing

character sets cannot exceed four, but the number of characters per set must be constant.

POLLID
specifies the polling character set. The number of polling character sets cannot exceed four, but the number of characters per set must be constant. The number of characters per polling character set need not be the same as the number of characters per addressing character set.

APPENDIX A: SYSTEM MACRO INSTRUCTIONS

There are a number of macro instructions that, if they are to be used, require system programmer privileges, but don't generate SVCs. These macro instructions are simply used as an assembly language vehicle to make it easier for you to produce code frequently used in system programs. Some of these macro instructions require you to define certain symbols in your program -- usually via a dummy section. We're only going to discuss the external appearance of these macro instructions in this appendix; if you want to see the macro definition, use the system macro/copy library. The following macro instructions are listed alphabetically by mnemonic:

Poll of pending attention interrupt	ATPOL
Locate JFCB corresponding to data set name	FINDDS
Locate JFCB and ensure volume mounting	FINDJFCB
Transfer control	INVOKE
Inhibit task interrupts	ITI
Permit task interrupts	PTI
Return to calling program	RESUME
Store register contents	STORE
Send message to task and await response	VSENDR

ATPOL -- Poll for Pending Attention Interrupt (nonstandard)

ATPOL is used to find out if there is a pending attention (task-asynchronous asynchronous I/O interrupt; if there is, control is transferred to the address specified by operand "pgmad." To use this macro instruction, you must define the symbols ISAAT and ISAATM unless you supply "switch"; these may be defined by copying the interrupt storage dummy srction (CHAISA) from the system macro/copy library.

Name	Operation	Operand
[symbol]	ATPOL	pgmad-addx [,switch-addx]

pgmad
: address of a program interested in a pending attention.

switch
: address of byte whose contents are to be tested.

FINDDS -- Locate JFCB Corresponding to Data Set Name (S)

The FINDDS macro instruction is used to obtain the location of the JFCB corresponding to a given data set name. If the data set name specified is not in the task definition table (TDT), but is in the catalog, the user can request that a JFCB be created.

Name	Operation	Operand
[symbol]	FINDDS	dsname-addr, byte-addr, area-addr

dsname
: specifies the address of a fully qualified data set name.

byte
specifies the address of a byte that the user has set to zero if he wants a JFCB created for a catalogued data set, or to non-zero if he does not want a JFCB created.

area
specifies the address of a word in which the pointer to the JFCB is to be placed.

FINDJFCB -- Locate JFCB and Ensure Volume Mounting (S)

The FINDJFCB macro instruction is used to locate the JFCB for a given data definition name and, optionally, to ensure the volumes specified in that JFCB are mounted.

Name	Operation	Operand
[symbol]	FINDJFCB	ddname-addr, byte-addr, area-addr

ddname
specifies the location of an 8-byte field containing the data definition name. If the ddname has fewer than 8 characters, it must be left-adjusted with trailing blanks.

byte
specifies the location of a 1-byte field containing a code.

area
specifies the location of a 4-byte field in which the address of the JFCB is to be placed.

INVOKE -- Transfer Control (nonstandard)

INVOKE causes transfer of control from one program or routine to another by means of the BASR instruction.

Name	Operation	Operand
[symbol]	INVOKE	address-addrx

address
specifies the address of a word that contains the address of the program to be invoked.

ITI -- Inhibit Task Interrupts (nonstandard)

ITI is used to prevent the occurrence of task interrupts; it does this by setting the interrupt storage area lock byte (ISALCK) to 1s. To use ITI, you must define the symbol ISAICK; you can do this by copying the interrupt storage area dummy section (CHAISA) from the system copy/macro library.

Name	Operation	Operands
[symbol]	ITI	None

PTI -- Permit Task Interrupts (nonstandard)

PTI is used to cancel the effect of an ITI macro instruction; it allows pending task interrupts to occur (if the task-mask bits in the VPSW are 1s). You must define the symbol ISALCK to use PTI; you can do this by copying the interrupt storage area dummy section (CHAISA) from the system macro-copy library.

Name	Operation	Operands
[symbol]	PTI	None

RESUME -- Return to Calling Program (nonstandard)

The RESUME macro instruction restores the specified registers from the specified area and returns control to the calling program.

Name	Operation	Operands
[symbol]	RESUME	[area-addrx,(reg1-integer[,reg2-integer])] [,RC=integer]

area
specifies the address at which the data to be restored is located.

reg1
specifies the first register to be restored from the specified area and must be greater than 7 but less than 16.

reg2
specifies the last register to be restored from the specified area. The restoration has the same wrap around feature as the STM or LM instructions. If this operand is omitted only the first register will be restored.

RC
specifies a return code to be sent back to the calling routine. This code must be less than 4092 and be a multiple of four.

STORE -- Store Register Contents (nonstandard)

The STORE macro instruction stores the specified register or registers in a specified area.

Name	Operation	Operand
[symbol]	STORE	area-addrx,(reg_1-integer [,reg_2-integer])

area
specifies the address of the storage area in which the the specified register or registers are to be saved.

reg_1 and reg_2
specify the range of registers to be stored. If reg_2 is not specified, only reg_1 is stored.

PROGRAMMING NOTES: reg_1 must be specified as equal to or greater than 8 and not greater than 15.

The specified area must be large enough to contain the specified range of registers.

VSENDR -- Send Message to Task and Await Response (nonstandard)

The VSENDR macro instruction is used to send a message to another task and to wait for a response from the receiving task.

Standard form

Name	Operation	Operand
[symbol]	VSENDR	msg-text, radd-addr, rleng-value, mcode-value, tid-addr

L-form

Name	Operation	Operand
[symbol]	VSENDR	msg-text, [radd-relexp] [rleng-absexp], [mcode-absexp] [,tid-relexp], MF = L

E-form

Name	Operation	Operand
[symbol]	VSENDR	,[radd-addrx],[rleng-value],[mcode-value] [,tid-addrx], MF=(E,list-{addrx / (1)})

msg
specifies the text of the message, and must be enclosed in single quotation marks.

radd
specifies the location into which the reply is to be placed.

specifies the length of the reply in bytes.

mcode
specifies the message code.

tid
specifies the ID of the sending task (i.e., the task to which the reply is to be sent).

list
specifies the location of the L-form of the macro instruction to which this E-form refers.

APPENDIX B: TIME CONVERSION ROUTINE

A number of privileged conversion routines are provided to enable you to convert time data, in any of several formats, into a form you can use with macro instructions SETTR and SETTU. Two types of conversions are performed: type-T; used for operations with the SETTU macro instruction, yields a 32-bit binary time interval in microseconds; type-R, used for operations with the SETTR macro instruction, yields a 64-bit binary time interval in microseconds elapsed since March 1, 1900 (see "Time-keeping"). Two different forms of input data may be used for type-T conversion (0 and 1); six forms (0-5) may be used for type-R. Table 16 summarizes the different input forms.

Table 16. Input Formats Accepted by Time Conversion Routine

Input data code	Input form
0	time interval in hours (h), minutes (m), seconds (s), tenths (t), and hundredths (h) of seconds; eight BCD characters: hhmmsstth
1	time interval in milliseconds; 32-bit binary number
2	time of day in hours (h), minutes (m), seconds (s), tenths (t), and hundredths (h) of seconds; eight BCD characters: hhmmsstth
3	day of week; four left-justified BCD characters: MOND, TUES, WEDN, THUR, FRID, SATU, SUND
4	day of month; two left-justified BCD characters: 00 through 31
5	day of year; eight left-justified packed decimal characters: 00yyddd+

To use the time conversion routine, you must put a pointer to a parameter list in register 1, the return address in register 14, and the address of the time conversion routine in register 15. It looks like this:

```
LA      1,PARAM          POINTER TO PARAMETER LIST IN
                         REGISTER 1
L       15,=V(CZCJXA)    ADDRESS OF CONVERSION ROUTINE
BASR    14,15            GO THERE
RETURN
```

```
PARAM   DC      C'c'        FORM OF INPUT DATA - 0,1,2,3,4,
                            or 5
        DC      C't'        TYPE OF CONVERSION - T OR R
        DC      H'0'        NOT USED
        DC      D'data'     INPUT DATA PLACED HERE - RESULTS
                            FOUND HERE
```

After completing the requested conversion, the time conversion routine returns control to the address found in register 14. The results are placed, right-justified, in the second and third words of the parameter list.

Note: The SETTU macro instruction expects a time value in milliseconds; if you use the time conversion routine to get a time interval (type-T), you must divide the result by 1000 to convert it to milliseconds.

Table 17 lists the meaning of the results obtained from the various conversions.

Table 17. Results of Time Conversion

Conversion	Result
T0	time interval in microseconds
T1	time interval in microseconds
R0	Current time + input time interval in microseconds from March 1, 1900
R1	Current time + input time interval in microseconds from March 1, 1900
R2	Next occurrence of input time in microseconds from March 1, 1900
R3	Next occurrence of day of week in microseconds from March 1, 1900
R4	Next occurrence of day of month in microseconds from March 1, 1900
R5	Next occurrence of day of year in microseconds from March 1, 1900

DRUM STORAGE FORMAT

Each IBM 2301 drum contains 900 pages of 4096 bytes. Dummy records of 246 bytes separate each data page on the drum to allow data channels to fetch and execute channel command words between pages. The 2301 contains 200 tracks; every even-odd pair of tracks is organized to contain 9 pages, with each track containing nine contiguous half pages. Figure 24 shows the organization of a typical even-odd track pair.

Address CCH H R	Record size	Gap size	Page number (within track pair)
*			
000 2n 1	4096	133	0
000 2n 2	246	133	dummy record
000 2n 3	4096	133	1
000 2n 4	246	133	dummy record
000 2n 5	4096	133	2
000 2n 6	246	133	dummy record
000 2n 7	4096	133	3
000 2n 8	246	133	dummy record
000 2n 9	2048	133	4 (first half)
			3 bytes are left over and unused
000 2n+1 1	2048	133	4 (second half)
000 2n+1 2	246	133	dummy record
000 2n+1 3	4096	133	5
000 2n+1 4	246	133	dummy record
000 2n+1 5	4096	133	6
000 2n+1 6	246	133	dummy record
000 2n+1 7	4096	133	7
000 2n+1 8	246	133	dummy record
000 2n+1 9	4096	133	8
			3 bytes, left over and unused

*Each track begins with IBM standard record zero
CCH is always zero for all 2301 tracks
n is an integer between zero and 99 inclusive

Figure 24. Organization of IBM 2301 Drum

The track pair (0-99) on which a drum page is contained may be obtained by dividing the page number by 4.5. The quotient is the track number; the remainder will be 0, 1, 2, 3, 4, .5, 1.5, 2.5, or 3.5; these remainder values correspond to the page number (0-8) within the track pair.

DISK STORAGE FORMATS

These restrictions apply to the use of the IBM 2314 or 2311, when formatted in pages:

1. Cylinder 199 is reserved for standard error-recovery retry.
2. Page 895 (2311) is not used because of overflow restriction.

Each IBM 2314 volume contains 3600 pages of 4096 bytes (each 2314 DASD contains 28,800 pages). Pages 0 and 1 contain three IPL records

(records one, two, and the last record on track 0 -- after the volume label and any user labels), and the IBM standard volume label (record three). Pages 0 and 1 are not available for VAM allocation.

Each 2314 disk pack has 200 cylinders with 18 tracks per cylinder, each cylinder is organized to contain 30 pages. Figure 25 shows a typical organization.

Record Address CC HH R	Record Size	Page Number
*		
nn 00 1	4096	0
nn 00 2	2790	1
nn 01 1	1306	1
nn 01 2	4096	2
nn 01 3	1302	3
nn 02 1	2794	3
nn 02 2	4096	4
nn 03 1	4096	5
nn 03 2	2790	6
nn 04 1	1306	6
nn 04 2	4096	7
nn 04 3	1302	8
nn 05 1	2794	8
nn 05 2	4096	9
nn 06 1	4096	10
nn 06 2	2790	11
nn 07 1	1306	11
nn 07 2	4096	12
nn 07 3	1302	13
nn 08 1	2794	13
nn 08 2	4096	14
nn 09 1	4096	15
nn 09 2	2790	16
nn 10 1	1306	16
nn 10 2	4096	17
nn 10 3	1302	18
nn 11 1	2794	18
nn 11 2	4096	19
nn 12 1	4096	20
nn 12 2	2790	21
nn 13 1	1306	21
nn 13 2	4096	22
nn 13 3	1302	23
nn 14 1	2794	23
nn 14 2	4096	24
nn 15 1	4096	25
nn 15 2	2790	26
nn 16 1	1306	26
nn 16 2	4096	27
nn 16 3	1302	28
nn 17 1	2794	28
nn 17 2	4096	29
*Each track begins with the IBM standard record zero		

Figure 25. Organization of IBM 2314 Volume for VAM

Each IBM 2311 volume contains 1616 pages of 4096 bytes. Pages 0 and 1 contain three IPL records (records one and two of track 0 and record one of track 1, and the IBM standard volume label (record three); they

are not available for allocation. A 2311 disk pack contains 202 cylinders of 10 tracks each; the cylinders are organized to contain 8 pages each. Figure 26 shows a typical cylinder organization.

Record Address CC HH R	Record Size	Page Number
nn 00 1	3625	0
nn 01 1	471	0
nn 01 2	3069	1
nn 02 1	1027	1
nn 02 2	2486	2
nn 03 1	1610	2
nn 03 2	1875	3
nn 04 1	2221	3
		1234 unused bytes, track 4
nn 05 1	3625	4
nn 06 1	471	4
nn 06 2	3069	5
nn 07 1	1027	5
nn 07 2	2486	6
nn 08 1	1610	6
nn 08 2	1875	7
nn 09 1	2221	7
		1234 unused bytes, track 9

Figure 26. Format of IBM 2311 Volume for VAM

The Supervisor must pass back to the virtual memory error processors a code identifying the type of software error perpetrated by the task and detected by the supervisor. To accomplish this, the Supervisor Processor must enqueue a GQE on the appropriate task's TSI program interrupt queue. The interrupt code in the GQE contains a value which uniquely identifies the cause of the program interrupt.

There are 17 interrupt codes used by the hardware. We reserve codes 18 thru 31 for future hardware interrupt expansion. This leaves codes 32 to 65535 for specifying software program interrupt errors. Further, codes 65280 thru 65535 are reserved for those errors which are temporary in nature.

The currently defined codes are:

Code Decimal	Hexadecimal	Error Type
0 through 31	0000 through 001F	Per Principles of Operation Manual
32	0020	Not assigned
33	0021	Nonprivileged program issued IOCAL, PGCUT
34	0022	IOPCB or IORCB page list too long
35	0023	Specified virtual address is not in user's virtual memory (IOCAL)
36	0024	Program has no I/O devices assigned to it (IOCAL)
37	0025	IORCB size of zero
40	0028	TSI Service Call interrupt counter overflow
41	0029	TSI External Interrupt counter overflow
42	002A	TSI Asynchronous interrupt counter overflow
43	002B	TSI Timer Interrupt counter overflow
44	002C	TSI Input/Output interrupt counter overflow
45	002D	GQE type code is in error
46	002E	IORCB size exceeds 1920 bytes
47	002F	IORCB or IOPCB crosses a page boundary
48	0030	Device not assigned to task (IOCAL,PGOUT)
50	0032	IOCAL or PGOUT SVC page address does not exist in virtual memory
51	0033	IOCAL or PGOUT SVC page is not in core
53	0035	Request to delete page from segment not previously assigned
54	0036	Request to delete page not previously assigned
61	003D	Invalid Segment number given to ADSPG SVC Processor
70	0046	User estimated time exceeded or user timer value not reset within quantum
71	0047	SYSERR detected while processing paging I/O error for this task

72	0048	Illegal code given to SETUP/XTRCT SVC Processor
73	0049	AWAIT SVC not executed remotely or else not on the last half word of an ECB
74	004A	Invalid Shared-Page Table number given to ADSPG SVC Processor
75	004B	Software has detected a possible hardware malfunction
76	004C	A VSEND message is too long or extends over a page boundary
80	0050	User's task not of sufficient priority to issue SVC
81	0051	SVC not on word boundary
82	0052	Count of external addresses is zero
83	0053	All parameters are not in one page
85	0055	Page unassigned
86	0056	Count exceeds 1022, bit string flag not set SETXP
93	005D	Illegal code given to SETSYS/XTRSYS SVC Processor
94	005E	Illegal code given to SETXTX/XTRXTS SVC Processor
96	0060	Enter SVC issued to interrupt table type routine while Type III linkage in effect and P1 flag on
97	0061	Enter SVC issued with invalid enter code-over 255_1 or not assigned
98	0062	SVC issued on nonprivileged state and no interrupt routine specified
99	0063	No Asynchronous Error Routine defined for device with error
100	0064	Asynchronous Interrupt received but no DE available for device
101	0065	SETTR not accepted because System Limit Reached in Table
102	0066	Program Interrupt received while in Type III linkage
103	0067	SVC Interrupt received while in Type III linkage
108	006C	PGOUT request for zero pages
109	006D	Attempt to add more than 256 shared pages to a segment
124	007C	Unsuccessful dequeue I/O request
125	007D	DRAM Flag illegally on
145	0091	Relocation page-in error (device defective) -- permanent volume
146	0092	Relocation page-in error (device defective)
147	0093	Relocation page-in error (medium defective)
148	0094	IOCAL page-in error (device defective) -- permanent volume
149	0095	IOCAL page-in error (device defective) -- moveable volume
150	0096	IOCAL page-in error (medium defective)
151	0097	Operator task has been reinitialized

Module	Name	Opt_1	Opt_2	Opt_3
CEAA0	I/O Call Routine	3	2	25
CEAA5	Pathfinding Subroutine	3	2	27
CEAA6	Page Direct Access Queue	3	2	31
CEAA7	Page Direct Access Interrupt	3	2	32
CEAAA	Command Word Relocator	3	2	35
CEAAB	Set Path SVC Routine	3	2	36
CEAAC	Queue Device on Task Routine	3	2	37
CEAAD	Remove Device from Task	3	2	38
CEAAH	Reset Device Supression Flag	3	2	42
CEAAI	Halt I/O	3	2	45
CEAAW	Data Recording I/O	3	2	55
CEAAX	Start Retry Operation	3	2	56
CEAAY	Data Recording Error Recovery	3	2	58
CEABE	External Machine Check Interrupt Processor	3	3	30
CEABQ	Generate and Enqueue Interrupt GQE	3	3	31
CEAH2	Setup TSI Field Subroutine	3	1	42
CEAH3	Extract TSI Field SVC	3	1	42
CEAH4	Delete TSI SVC	3	1	42
CEAH5	Add Pages	3	1	42
CEAH8	List Changed Pages	3	1	42
CEAI1	Sense Partitioning Switches	3	1	42
CEAIM	Machine Check New PSW	3	3	26
CEAIR	Recovery Nucleus	3	3	25
CEAIS	System Error Processor	3	3	29
CEAJE	Enqueue/Dequeue Routines	3	1	46
CEAJM	Move GQE Routine	3	1	50
CEAJS	Set Suppress Flag	3	1	54
CEAKR	Create Real Time Interrupt	3	1	57
CEAL2	Supervisor Core Release	3	1	33
CEAL4	User Core Release	3	1	34
CEAMA	Activate TSI Routine	3	1	40
CEAMC	Create TSI Routine	3	1	42
CEAMT	Task Initiation Routine	3	1	36
CEAMX	XTSI Overflow Routine	3	1	37
CEAP6	Add Shared Pages	3	1	42
CEAP8	Move Real Core SVC	3	1	42
CEAP9	Time Slice End SVC	3	1	42
CEAQ7	Connect Segment to Shared Page Table	3	1	42
CEAQ8	Disconnect Segment from Shared Page Table	3	1	42
CEAS2	Setup System Table Field	3	1	42
CEAS3	Extract System Table Field	3	1	42
CEAS4	Setup XTSI Field	3	1	42
CEAS5	Extract TSI Field	3	1	42
CEAS8	Restore Time Subroutine	3	1	42
CEAT0	Cancel Recording	3	1	42
CEAT1	Extract Accumulated Time	3	1	42
CEAT2	Special Create TSI Routine	3	1	42
CFADA	LPC MAIN	5	4	25
CFADB	LPC GETLINE	5	4	26
CFADC	LPC PUT DIAG	5	4	27
CFAMA	PCS INPUT PHASE 1	5	9	28
CFAMB	PHASE 1 RUN	5	9	28
CFAMC	PHASE 1 STOP	5	9	28
CFAMD	FORMLIST (Form Parameter List)	5	9	28

Part 1 of 8

Module	Name	Opt_1	Opt_2	Opt_3
CFAME	PHASE 1 IF	5	9	28
CFAMF	PHASE 1 AT	5	9	28
CFAMH	EXPSCAN (Expression Scan)	5	9	28
CFAMJ	SUBPOL (Subscript to Polish)	5	9	28
CFAMM	INSTLOC (Form Instruction Location Definition)	5	9	28
CFAMR	Qualify Directive	5	9	28
CFAMS	Remove Directive	5	9	28
CFANA	PCS Input PHASE-II	5	9	28
CFAND	PHASE II AT	5	9	28
CFANE	PHASE II LIST2	5	9	28
CFANF	CODEGEN (Code Generator)	5	9	28
CFANH	COMCON (Combine Constants)	5	9	28
CFANV	GETBASE (Base Register Assignment)	5	9	28
CFANW	DIAGNO (Issue Diagnostic)	5	9	28
CFANX	Prompt	5	9	28
CFAOA	VALMOD (Evaluate Module Name)	5	9	28
CFAOB	VALSYM (Evaluate Internal Symbol)	5	9	28
CFAOD	GETREG (Register Assignment)	5	9	28
CFAPA	PCS Output Overall	5	9	29
CFAPB	PCS Output Control	5	9	29
CFAPH	LINE (Output Line Routine)	5	9	29
CFAPK	SAVIX (Saved Instruction Execution)	5	9	29
CFAQA	Display/Dump Control	5	9	30
CFAQB	NEXTLIST (Process Parameter List)	5	9	30
CFAQC	NEXTITEM (Process Display List)	5	9	30
CFAQD	NEXTISD (Process Next ISD Entry)	5	9	30
CFAQF	DISREG (Display Registers)	5	9	30
CFAQG	SIMVAR (Display Simple Variable)	5	9	30
CFAQH	ADDITEM (Convert an Item by Data Type)	5	9	30
CFAQI	DISINST (Display an Instruction)	5	9	30
CFAQJ	DISARAY (Display an Array)	5	9	30
CFAQK	DISALINE (Display a Line of an Array)	5	9	30
CFAQM	DISHEX (Display a Range in Hexadecimal)	5	9	30
CFAQN	DISHLINE (Display a Hexadecimal Line)	5	9	30
CFAQR	DISYM (Display Symbol)	5	9	30
CFAQU	DISOUT (Output a Line)	5	9	30
CFAQV	REALCON (Real Number Conversion)	5	9	30
CFAQW	SUBERR (Output Subscipt Diagnostic)	5	9	30
CFAUC	Cancel Data Recording	8	14	27
CGCCA	Allocate Module	8	2	28
CGCCB	Select Hash	8	2	28
CGCCE	Resolve Symbol	8	2	28
CGCCH	Load PMD	8	2	28
CGCCJ	Fix PMD	8	8	28
CGCCK	Attach Text	8	2	28
CGCCL	Fix	8	2	28
CGCCN	Add PMD	8	2	28
CGCCO	Drop PMD	8	2	30
CGCCP	Reject Diag	8	2	28
CGCCR	Bisearch	8	2	28
CGCCT	PCSA	8	2	28
CGCCU	Check DEF Legal	8	2	28
CGCCV	Link DEFS	8	2	28
CGCCW	Get Storage	8	2	28
CGCCY	Define REF	8	2	28
CGCDA	Modify MUT Counts	8	2	30
CGCDB	Delete Caller Mutes	8	2	30
CGCDC	Delete Selected Mutes	8	2	30
CGCDD	Modify Use Counts	8	2	30
CGCDE	Test User Counts	8	2	30

Module	Name	Opt_1	Opt_2	Opt_3
CGCDG	Add Mute	8	2	28
CGCDPR	Loader Gate	8	2	28
CGCKA	Symbolic Library Indexing Routine	8	10	25
CGCKB	SYSXBLD (Build Symbolic Library Index)	8	10	26
CGCKC	SYSEARCH (Symbolic Library Search Routine)	8	10	27
CGCKZ	Control Section Store Routine	8	11	25
CGCMA	Reconfiguration	3	3	28
CHCAA	SQRT (Single-Precision Square Root Subroutine)	9	9	25
CHCAB	DSQRT (Double-Precision Square Root Subroutine)	9	9	26
CHCAC	EXP (Single-Precision Exponential Subroutine)	9	9	27
CHCAD	DEXP (Double-Precision Exponential Subroutine)	9	9	28
CHCAE	LOG and LOG10 (Single-Precision Logarithm Subroutine)	9	6	29
CHCAF	DLOG and DLOG10 (Double-Precision Logarithm Subroutine)	9	9	30
CHCAI	SIN and COS (Single-Precisicn Sine and Cosine Subroutine)	9	9	31
CHCAJ	DSIN and DCOS (Double-Precision Sine and Cosine Subroutine)	9	9	32
CHCAK	TANH (Single-Precision Hyperbolic Tangent Subroutine)	9	9	33
CHCAL	DTANH (Double-Precision Hyperbolic Tangent Subroutine)	9	9	34
CHCAM	CEXP (Single-Precision Complex Exponential Subroutine)	9	9	35
CHCAN	CDEXP (Double-Precision Complex Exponential Subroutine)	9	9	36
CHCAO	CLOG and CLOG10 (Single-Precision Complex Logarithm)	9	9	37
CHCAP	CDLOG and CDLOG10 (Double-Precision Complex Logarithm)	9	9	38
CHCAQ	CSIN and CCOS (Single-Precision Complex Sine and Cosine)	9	9	39
CHCAR	CDSIN and CDCOS (Double-Precision Complex Sine and Cosine)	9	9	40
CHCAS	CSQRT (Single-Precision Complex Square Root Subroutine)	9	9	41
CHCAT	CDSQRT (Double-Precision Complex Square Root Subroutine)	9	9	42
CHCAU	CABS (Single-Precision Complex Absolute Value Subroutine)	9	9	43
CHCAV	CDABS (Double-Precision Complex Absolute Value Subroutine)	9	9	44
CHCAW	ARCSIN and ARCCOS (Single-Precision Arcsine and Arccosine)	9	9	45
CHCAX	DARSIN and DARCOS (Double-Precision Arcsine and Arccosine)	9	9	46
CHCAY	TAN and COTAN (Single-Precision Tangent and Cotangent)	9	9	47
CHCAZ	DTAN and DCOTAN (Double-Precision Tangent and Cotangent)	9	9	48
CHCBA	SINH and COSH (Single-Precision Hyperbolic Sine and Cosine	9	9	49
CHCBB	DSINH and DCOSH (Single-Precision Hyperbolic Sine and Cosine	9	9	50
CHCBC	Eight-Byte Complex Number to Integer Power Exponentiation	9	9	51
CHCBD	Interrupt and Machine Indicator	9	9	52
CHCBE	Specification Interrupt Program	9	9	53

Part 3 of 8

Module	Name	Opt_1	Opt_2	Opt_3
CHCBG	FJXPJ,FJXPI,FIXPJ,FIXPI (Base to Integer Integer Power)	5	1	26
CZAAB	Gate Subroutine	9	9	55
CHCBH	FRXPJ,FRXPI (Real Four-Byte Base to Integer Power)	9	9	56
CHCBI	FDXPJ,FDXPI (Real Eight-Byte Base to Integer Power)	9	9	57
CHCBJ	FJXPR,FIXPR,FRXPR (Integer and Four-Byte Real Base to four-Byte Real Power	9	9	58
CHCBK	FJXPD,FICPD,FRXPD,FDXPR,FDXPD (Real Eight-Byte Base or Power to Real Power)	9	9	59
CHCBM	Sixteen-Byte Complex Number to Integer Power Exponentiation	9	9	60
CHCBQ	ATAN and ATAN2 (Single-Precision Arctangent)	9	9	61
CHCBR	DATAN and DATAN2 (Double-Precision Arctangent)	9	9	61
CHCBT	GAMMA and ALGAMA (Single-Precision Gamma Function)	9	9	61
CHCBU	ERF and ERFC (Single-Precision Error Integral Function)	9	9	65
CHCBV	DGAMMA and DLGAMA (Double-Precision Gamma Function)	9	9	61
CHCBW	DERF and DERFC (Double-Precision Error Integral Function)	9	9	62
CHCBZ	LIBER (Library Program Error Handling Routine)	9	8	25
CHCIA	Initialization	9	8	26
CHCIB	DCB Maintenance	9	8	27
CHCIC	I/O Control	9	8	28
CHCID	Namelist Processor	9	8	29
CHCIE	List Item Processor	9	8	30
CHCIF	Format Processor	9	8	31
CHCIG	Integer Input Conversion	9	8	32
CHCIH	Integer Output Conversion	9	8	33
CHCII	Real Input Conversion (No Exponent)	9	8	34
CHCIJ	Real Output Conversion (No Exponent)	9	8	35
CHCIK	Real Input Conversion (Exponent)	9	8	36
CHCIL	Real Output Conversion (Exponent)	9	8	37
CHCIM	Complex Input Conversion	9	8	38
CHCIN	Complex Output Conversion	9	8	39
CHCIO	Alphameric Input Conversion	9	8	40
CHCIP	Alphameric Output Conversion	9	8	41
CHCIQ	Logical Input Conversion	9	8	42
CHCIR	Logical Output Conversion	9	8	43
CHCIS	General Input Conversion	9	8	44
CHCIT	General Output Conversion	9	8	45
CHCIU	List Termination	9	8	46
CHCIV	Dump Module	9	8	47
CHCIW	Exit Module	9	8	48
CHCIX	I/O Error Message Control	9	8	49
CHCIY	I/O Psect Module	8	9	29
CMAGA	Option Selection Routine	3	3	27
CMASA	SERR Bootstrap	3	3	27
CMASB	Environment Recording	3	3	27
CMASC	Immediate Print	3	3	27
CMASD	Checker	3	3	27
CMASE	Pointer	3	3	27
CMASF	Restore and Validate	3	3	27
CMASG	Instruction Retry Execution	3	3	27
CMASH	CPU/Memory Checkout-1	3	3	27
CMASI	CPU/Memory Checkout-2			

Part 4 of 8

Module	Name	Opt_1	Opt_2	Opt_3
CMASJ	CPU/Memory Checkout-3	3	3	27
CMASK	DUM-1	3	3	27
CMASL	DUM-2	3	3	27
CMASM	DUM-3	3	3	27
CMASN	Environment Recording Edit and Print	3	3	27
CMATC	OLTS Print	8	9	34
CMATD	OLTS Utilities-Compare	8	9	35
CMATE	OLTS Conversion	8	9	36
CMATF	Setup Control Routine	8	9	25
CZAAA	Director	5	1	25
CZAAB	GATE Subroutine	5	1	26
CZAAC	Scan	5	1	27
CZAAD	MSGWR	5	1	29
CZAAE	Gatex	5	1	28
CZAAF	VMTI (Virtual Memory Task Initiation)	5	1	30
CZABA	Batch Monitor	5	1	25
CZABB	Execute Command Routine	5	2	26
CZABC	Background Command Routine	5	2	27
CZABD	Bulkio Preprocessor	5	2	28
CZABE	Batch (Read Cards)	5	2	29
CZABF	RTAPE (Read Tape)	5	2	30
CZABG	List (Print)	5	2	31
CZABH	Card (Punch Cards)	5	2	32
CZABI	Tape (Write Tape)	5	2	33
CZABJ	Cancel	5	2	34
CZABQ	XWTO	5	5	76
CZACA	MOCP (Main Operator Control Program)	5	3	25
CZACB	MOHR (Main Operator Housekeeping Routine)	5	3	26
CZACC	OXIP (Operator External Interrupt Processor)	5	3	27
CZACD	Reply	5	3	28
CZACE	Message/Announce	5	3	29
CZACF	Braodcast	5	4	29
CZACG	Force Command Routine	5	3	31
CZACN	Shutdown	5	3	38
CZACQ	ABEND Processor	5	5	67
CZACR	ABEND Processor	5	5	67
CZACP	ABEND	5	5	67
CZACS	Pair Table	5	5	66
CZADF	Data	5	5	45
CZAEA	Datadef	5	5	25
CZAEB	Findjfcb	5	5	26
CZAEC	Findds	5	5	27
CZAED	Load	5	5	28
CZAEG	Modify	5	5	31
CZAEH	LOCFQN	5	5	32
CZAEI	Catalog	5	5	33
CZAEJ	Erase/Uncatalog	5	5	34
CZAEK	Present Director	5	5	35
CZAEL	Present Data Set Status	5	5	36
CZAEM	Present Line	5	5	37
CZAEN	Present VTOC	5	5	38
CZAFG	Unload	5	5	56
CZAFH	Permit	5	5	57
CZAFI	Share	5	5	58
CZAFJ	Release	5	5	59
CZAFK	Join	5	5	60
CZAFL	Quit	5	5	61
CZAFM	Logon	5	5	62
CZAFN	Logoff	5	5	63
CZAFS	Ddcall	5	5	68

Part 5 of 8

Module	Name	Opt_1	Opt_2	Opt_3
CZAFU	Reserve	5	5	70
CZAFV	Dscopy	5	5	71
CZAGA	Accounting Routine	5	6	25
CZAHA	Diagno	5	7	25
CZAHB	IAIP (Initial Attention Interrupt Processor)	5	7	26
CZAHC	XIP/XIIS (External Interrupt Processor)	5	7	27
CZAHD	External Interrupt Subprocessor (message/Error)			
CZAMA	Dump	5	9	28
CZAMI	DATAFLD (Form Data Field Definition)	5	9	28
CZAML	DATALOC (Form Data Location Definition)	5	9	28
CZAMN	INTERNAL (Form Internal Symbol Definition)	5	9	28
CZAMO	EXTERNAL (Form External Symbolic Definition)	5	9	28
CZAMP	Offset	5	9	28
CZAMQ	SCANFLD (Scan Field to Delimiter)	5	9	28
CZAMT	PCS Unload	5	9	28
CZANG	SUBGEN (Subscript Computation)	5	9	28
CZANI	OPGEN (Operator Code Generation)	5	9	28
CZANT	LOADOP (Load Operand)	5	9	28
CZAPC	FINDLOC (Location Table Scan)	5	9	29
CZAPG	SYMGEN (Symbol Generator)	5	9	29
CZAPL	FINDREAL (Find Real Address)	5	9	29
CZAQA	PCS Debug	5	9	26
CZASE	VMEREP (Virtual Memory Environment Recording Edit and Print)	4	10	31
CZASX	Error Control	4	10	28
CZASY	Drum Access Module	4	10	27
CZATA	OLTAM Execute I/O	8	9	32
CZATB	OLTAM Posting	8	9	33
CZATG	Device Allocation	8	9	28
CZAUC	Data Recording Cancel	8	14	26
CZCAA	MTREQ Routine	8	5	25
CZCAB	Bump Routine	8	5	26
CZCAC	Pause Routine	8	5	28
CZCAD	Release Routine	8	5	27
CZCCD	Loader Logoff	8	2	31
CZCDH	LIBE Maintenance	8	2	32
CZCDL1	Explicit Linkage	8	2	28
CZCDL2	Hash Search	8	2	28
CZCDL3	LIBE Search	8	2	28
CZCDL4	Page Relocation	8	2	29
CZCDL5	Map Search	8	2	28
CZCDL6	Set Search Flags	8	2	28
CZCDU1	Explicit Unlinking	8	2	30
CZCDU2	Delete Module	8	2	30
CZCEA	Allocate	8	3	25
CZCEB	VAM Search	8	3	27
CZCEC	SAM Search	8	3	29
CZCED	VAM Merge	8	3	30
CZCEE	SAM Merge	8	3	33
CZCEG	Give Back SAM Storage (GIVBKVS)	8	3	31
CZCES	Scratch Data Set	8	3	32
CZCEV	Give Back VAM Storage (GIVBKV)	8	3	26
CXCEX	Extend	8	3	28
CXCFA	Addcat	8	1	27
CZCFD	Delcat	8	1	29
CZCFG	Get SBLOCK	8	1	35
CZCFH	Search SBLOCK	8	1	36
CZCFI	Index	8	1	26
CZCFL	Locate	8	1	25
CZCFO	Obtain	8	1	32

Module	Name	Opt_1	Opt_2	Opt_3
CZCFR	Retain	8	1	33
CZCFS	Share	8	1	30
CZCFU	Shareup	8	1	28
CZCFV	Unshare	8	1	31
CZCFZ	Rename	8	1	34
CZCGA	Virtual Memory Allocation	8	6	15
CZCJA	Stimer Routine	8	13	32
CZCJC	Cleanup	8	13	36
CZCJD	Delete Interrupt Routine (DIR)	8	13	30
CZCJI	Interrupt Inquiry (INTINQ)	8	13	31
CZCJK	Task Monitor Scanner and Dispatcher	3	13	28
CZCJL	Leave Privilage (LVPRV)	8	13	27
CZCJQ	Queue Linkage Entry	8	13	26
CZCJS	Specify Interrupt Routine (SIR)	8	13	29
CZCJT	Task Monitor Interrupt Processor	8	13	25
CZCJX	Time Conversion Package	8	13	33
CZCJY	Set Clock Routine	8	13	34
CZCJZ	Cancel Clock Routine	8	13	35
CZCLA	Open (Common)	4	1	25
CZCLB	Close (Common)	4	1	27
CZCLD	Force End of Volume	4	1	30
CZCMA	GETBUF	4	1	31
CZCMB	GETPOOL	4	1	34
CZCMO	Configuration Console System Sequence Control)			
CZCNA	Freebuf	4	1	32
CZCNB	FREEPOOL	4	1	35
CZCOA	VAM Open	4	5	28
CZCOB	VAM Close	4	5	29
CZCOC	VAM Move Page	4	5	30
CZCOD	VAM Insert/Delete Data Set Pages	4	5	31
CZCOE	VAM Reqpage (Assign External Entries in RESTBL)	4	5	32
CZCOF	VAM Insert (Insert Page Entries in RESTBL)	4	5	33
CZCOG	VAM RECLAIM (Delete Data Set Pages and Make External Pages Available)	4	5	34
CZCOH	VAM INTLK (Routine for Imposing Interlocks)	4	5	35
CZCOI	VAM RLINTLK (Routine for Releasing Interlock)	4	5	36
CZCOJ	VPAM Find Macro Instruction and Routine	4	8	27
CZCOK	VPAM Stow Macro Instruction and Routine	4	8	28
CZCOL	VPAM Search	4	8	29
CZCOM	VPAM Extpod (Extend POD)	4	8	30
CZCON	VPAM RELMBRS (Relocate Partitioned Data Set Members by POD)	4	8	31
CZCOO	VPAM GETNUMBER	4	8	32
CZCOP	VSAM OPEN	4	6	30
CZCOQ	VSAM CLOSE	4	6	31
CZCOR	GET LOC	4	6	26
CZCOS	VSAM PUT LOC	4	6	27
CZCOT	VSAM SETL	4	6	28
CZCOU	VSAM PUTX	4	6	29
CZCOV	VSAM FLUSHBUF	4	6	32
CZCPA	VISAM PUT MV	4	7	30
CZCPB	VISAM GET MV	4	7	32
CZCPC	VISAM SETL	4	7	33
CZCPE	VISAM Read/Write	4	7	37
CZCPI	VISAM Get Page	4	7	54
CZCPL	VISAM ADE (Add Directory Entry)	4	7	47
CZCPZ	VISAM Open	4	7	40
CZCQA	VISAM Close	4	7	41
CZCQE	SRCHSDST (Shared Data Set Table Manipulation Routine)	4	5	38

Part 7 of 8

Module	Name	Opt_1	Opt_2	Opt_3
CZCQF	GETSDST and RELSDST	4	5	39
CZCQI	EXP RESTBL	4	5	50
CZCQQ	VAM ABEND INTLK	4	5	40
CZCRA	BSAM Read/Write	4	2	25
CZCRB	BSAM Control Routine	4	2	32
CZCRC	BSAM Check	4	2	26
CZCRF	BSAM Prtov	4	2	31
CZCRG	BSAM Backspace a Block	4	2	30
CZCRH	SAM Direct Access Error Retry Routine	4	2	36
CZCRM	BSAM Point Position to a Block	4	2	28
CZCRN	BSAM Note Address of Last Block Processed	4	2	27
CZCRP	BSAM Posting and Error Retry	4	2	36
CZCRQ	BSAM Determines Records per Track	4	2	37
CZCRR	BSAM RELFUL	4	2	38
CZCRS	BSAM FULREL	4	2	29
CZCRX	VMER (Virtual Memory Error Recording)	4	10	26
CZCRY	VMSDR (Virtual Memory Statistical Data Recording)	4	10	25
CZCSB	IOREQ (I/O Request	4	11	25
CZCSC	IOREQ open	4	11	26
CZCSD	IOREQ close	4	11	27
CZCSE	IOREQ Posting	4	11	28
CZCWB	Build Common Portion of Data Extent Block	4	2	41
CZCWC	BSAM Close Mainline	4	2	34
CZCWD	Direct-Access Open	4	2	33
CZCWF	Tape Input Trailer Label Processor	4	2	43
CZCWH	Tape Input Header Label Processor	4	2	43
CZCWL	Build Direct-Access Data Extent Block	4	2	41
CZCWM	SAM Shared Routines Message Processor	4	2	42
CZCWO	BSAM Open Mainline	4	2	33
CZCWP	Tape Positioning	4	2	42
CZCWR	Read Format-3 DSCB	4	2	41
CZCWT	SAM Open Tape	4	2	33
CZCWV	SAM Shared Routines Volume Sequence Convert	4	2	42
CZCWX	Volume-Label Reader	4	2	40
CZCXC	Tape Output-Label Creator	4	2	43
CZCXD	Direct Access Output End-of-Volume Processor	4	2	35
CZCXE	SAM End-of-Volume Mainline	4	2	35
CZCXF	Tape Output Trailer-Label Processor	4	2	43
CZCXH	Tape Output Header-Label Processor	4	2	43
CZCXI	BSAM Direct-Access Input EOV	4	2	35
CZCXK	Check for Tape Read/Write	4	2	43
CZCXN	Direct-Access Input-Label Processor	4	2	43
CZCXO	Tape Output End-of-Volume	4	2	35
CZCXS	Set DSCB	4	2	35
CZCXT	Tape Input End-of-Volume	4	2	35
CZCXU	Direct-Access Output-Label Processor	4	2	43
CZCXX	Concatenation Processor	4	2	35
CZCYA	TAM Open	4	4	25
CZCYG	TAM Close	4	4	26
CZCYM	TAM Read/Write	4	4	27
CZCZA	Tam Posting	4	4	28
EAINV	System Inventory Program	3	1	53
SYSKA	Systime Conversion	8	15	25

Part 8 of 8

READER'S COMMENT FORM

IBM System/360 Time Sharing System
System Programmer's Guide

C28-2008-0

- Your comments, accompanied by answers to the following questions, help us produce better publications for your use. If your answer to a question is "No" or requires qualification, please explain in the space provided below. Comments and suggestions become the property of IBM.

	Yes	No
• Does this publication meet your needs?	☐	☐
• Did you find the material:		
Easy to read and understand?	☐	☐
Organized for convenient use?	☐	☐
Complete?	☐	☐
Well illustrated?	☐	☐
Written for your technical level?	☐	☐

- What is your occupation? _____
- How do you use this publication?

As an introduction to the subject?	☐	As an instructor in a class?	☐
For advanced knowledge of the subject?	☐	As a student in a class?	☐
For information about operating procedures?	☐	As a reference manual?	☐

Other _____

- Please give specific page and line references with your comments when appropriate. If you wish a reply, be sure to include your name and address.

COMMENTS:

- Thank you for your cooperation. No postage necessary if mailed in the U.S.A.

YOUR COMMENTS PLEASE . . .

This publication is one of a series which serves as reference for systems analysts, programmers and operators of IBM systems. Your answers to the questions on the back of this form, together with your comments, will help us produce better publications for your use. Each reply will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Please note: Requests for copies of publications and for assistance in utilizing your IBM system should be directed to your IBM representative or to the IBM sales office serving your locality.

Fold

Fold

FIRST CLASS
PERMIT NO. 34
YORKTOWN HTS., NY

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U. S. A.

IBM Corporation
PO Box 344
2651 Strang Boulevard
Yorktown Heights, N.Y. 10598

ATTN: Time Sharing System/360
Programming Publications Dept. 561

Fold

Fold

TSS/360 Printed in U.S.A. Form C28-2008-0

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
(USA Only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

C28-2008-0

TSS/360 Printed in U.S.A. Form C28-2008-0

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]